CONTENT ANALYSIS TOOLS IN ANDROID MEMORY FORENSICS

by

Andre V. Talley

A Capstone Project Submitted to the Faculty of

Utica College

April 2014

in Partial Fulfillment of the Requirements for the Degree of

Master of Science in
Cybersecurity

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Abstract

The purpose of this study is to examine and discuss the efforts underway to expand and improve the forensic community's ability to detect running malware in live Android devices. The need to conduct memory forensics on mobile devices has become more evident in recent years, as consumer use of tablets and smartphones grows. Various innovative methods have been proposed over time regarding Android malware detection. Commercial and open source vendors have created several reputable tools to conduct mobile malware analysis, but they largely focus on static and dynamic analysis of the APK files instead of examining live memory and program execution dynamically on the device itself. The forensic community at large has realized the importance of gathering and analyzing live memory dump data from an OS, as some system environment information is never stored statically on the system's storage media, and mobile devices are no exception. Android and its security construct present unique challenges to accomplish this, and traditional tools that work on other Operating Systems don't work properly with Android, driving the need for custom applications. The custom tools reviewed in this report, Dalvik Inspector (DI) and drozer pro, were developed specifically for use on Android devices. Many issues were encountered over the course of the research, including change of operating environment to conduct the testing, and problems with acquiring appropriate device kernel source code resulting in inability to collect device physical memory captures. Based on this outcome, DI could not be adequately assessed as intended. However, minimal problems occurred in assessing drozer pro and it proved to be a well-designed dynamic assessment tool that operated as intended. The need for future testing, collaboration, and providing assistance when possible to these product vendors and their developers will help bring these products to full maturity and acceptance. Keywords: Cybersecurity, Mr. Albert Orbinati, 504ensics Labs, MWR InfoSecurity, Volatility Framework, cross-compile, SQLite database.

Acknowledgments

This research project has been a great opportunity and has proved a humble introduction for me into the world of mobile forensics and its challenges. It was my intent to provide this as a resource for mobile forensic practitioners to learn about the usefulness of an innovative tool with great promise, and hopefully in the near future it will prove fruitful. I wish to thank Professor Albert Orbinati and Joseph T. Sylve for their invaluable guidance and constructive criticism in helping me craft and complete this Capstone project. Mr. Sylve and 504ensics Labs graciously provided Alpha versions of their Dalvik Inspector software for review, and they were an invaluable resource in helping me resolve issues I encountered over the course of the entire project. Likewise, Daniel Bradberry at MWR InfoSecurity provided a trial license code allowing me to assess drozer pro and graciously offered to assist me with any issues I may have had. I want to thank those who donated old unused phones as test hardware, and I especially want to thank my wife and family for their immeasurable and highly appreciated patience for the several months it took to conduct this research as I learned, struggled, and dealt with the demands of this work.

Table of Contents

List of Illustrative Materials vi
Statement of the Problem 1
Justification 1
Deficiencies 4
Audience 6
Literature Review 7
Malware Detection History 7
Collection Methods 10
Live memory and Dalvik analysis 13
Focus 16
Methodology 18
Dalvik Inspector 18
drozer pro 29
Recommendations and Conclusions 42
Challenges 42
Advantages 43
Conclusion 45
Recommendation 45
References 47
Appendices 51
Appendix A – LiME Cross-compile Instructions 52
Appendix B – Dalvik Inspector plugin creation sample images 54

List of Illustrative Materials

Figure 1. Santoku Community Environment v.0.4 18
Table 1. List of Android test subject devices 19
Figure 2. Nexus 4 virtual device running KitKat v.4.4.2 20
Figure 3. LiME web site 21
Table 2. Cross-compile and module insertion results 22
Figure 4. Volatility Framework LiME image parsing plugin output 25
Figure 5. Research output vs. expected sample output 25
Figure 6. Chuli malware sample database content list 26
Figure 7. Chuli malware sample PhoneService module content 27
Figure 8. Chuli malware sample SendInfo module content 27
Figure 9. Chuli malware sample plugin creation output 28
Figure 10. drozer Agent on devices and activated embedded server 30
Figure 11. drozer command-line console connected to virtual Nexus 4 device 30
Table 3. drozer commands 31
Table 4. drozer built-in module commands 32
Figure 12. Sample drozer app.package.list command output 34
Figure 13. Sample drozer app.package.list command output 34
Figure 14. Sample drozer app.package.attacksurface command output 35
Figure 15. Sample drozer app.activity.info command output 36
Figure 16. Sample drozer app.activity.start command 36
Figure 17. Sample drozer app.activity.start command output 37
Figure 18. Sample drozer app.provider.info command output 37
Figure 19. Sample drozer app.provider.info command output 38
Figure 20. Sample of drozer additional modules 39
Table 5. drozer built-in exploits and payloads 39
Figure 21. drozer pro interface 40
Figure 22. Sample drozer pro application structure display 41

Statement of the Problem

Google's Android mobile operating system (OS) has grown in popularity over the past several years. In 2010, it surpassed its closest rival, Apple iOS, to become the nation's fastest growing mobile platform, and held 81% market share of worldwide smartphone shipments for third quarter 2013 ("IDC," 2013, p. 1). The explosion of international vendors selling lightweight tablets and smartphones that run some version of Android has introduced a new dynamic in the digital forensic and law enforcement communities. History has shown that malware development grows for an OS as its demand grows, and Android is no exception. The purpose of this study is to examine and discuss two of the efforts underway to expand and improve the forensic community's ability to detect running malware in live Android devices.

Justification

The need to conduct memory forensics on mobile devices has become more evident in recent years, as consumers move away from using traditional desktop and laptop computing platforms as their primary means of casual computing and Internet access. Digital computing technology continues to progress at an astounding rate. Semiconductor vendors that create microprocessor and memory chips seem to have adopted a circuit development rate for themselves in accordance with what has been labeled Moore's Law, which predicted technological progress's ability to double the amount of components on an integrated circuit every two years as costs declined and demand grew (Moore, 2005). This progress has led to increased processing and storage speeds with lower power consumption, leading to reduced component heat and improved miniaturization. Today's mobile devices are lighter, much faster, and more powerful than their mainframe ancestors, and competition from multiple vendors has driven prices down to the benefit of all consumers.

Another technological advance that has spurred this industry is the explosive growth, usage, and content of today's Internet. What humbly began as ARPANET, a government project built to allow easier exchange of ideas and research between government and various collegiate research institutions, has grown beyond what any of those original scientists could have imagined. With that growth has come an explosion in content, capability, and applicability. The Internet has become a resource for education, news, entertainment, communication, information sharing, and more. Media outlets have altered their distribution methods to include legal digital downloads and content streaming via websites like Hulu, M-GO, and Netflix. Combined with modern wireless and mobile broadband capabilities, there are very few places in the world now where one cannot get at least minimal Internet access. This freedom from being tethered to a physical wired connection has likewise been a significant factor in the growth of mobile device usage around the world. Applications have been created that allow these devices to be used for various facets of computing – from gaming to books to movies to video conferencing. As more applications and content become available and consumer demand for them grows, so will the need for the forensic community to be able to safely and accurately acquire and parse information stored on these devices.

Acquiring and analyzing physical memory is seen by Digital Forensics and Incident Response (DFIR) professionals as critical to the success of an investigation, whether it be a criminal case, employee policy violation, or enterprise intrusion (FOR526: Memory Forensics In-Depth, 2014, p. 1). Several applications have been developed for public and Law Enforcement Only (LEO) use to forensically acquire memory content, such as KnTTools, Volatility, and MoonSols Windows Memory Toolkit. Unfortunately many of these tools are designed with the various versions of the Windows desktop/server OS in mind, and solutions available for other

OSes, like Linux and OS X, are far fewer. Volatility is one of those few that work on all 3 as long as Python is installed. This highlights the need for more vendor products to be designed and validated to address other popular OSes like Android.

Android's increase in popularity has positioned it as a new attack vector in the ongoing war between White Hat information security practitioners and Black Hat malicious code authors. Malware on Android devices was expected among leading industry experts to increase exponentially in 2013. Murphy stated, "For law enforcement and forensics professionals, mobile malware means dealing with potentially compromised devices that may help perpetrators cover their tracks, making it increasingly difficult for investigators to meet the threshold of reasonable doubt" ("Cellebrite," 2013, para. 10). Kaspersky Labs identified the Chuli malware as having been most likely designed by Chinese-speaking authors and is believed to be the first of this kind utilizing fully functional Android malware, specifically targeting mobile devices of Mongolian, Chinese, Tibetan, and Uyghur political activists. The malware stole private data from infected smartphones, including the address book and messaging history, and sent it to a command and control server. These attacks began in March 2013 with the hacking of an email account belonging to a high-profile Tibetan activist. The attackers used this account to send 'spear-phishing' emails to their contact list with an .APK file attached containing a malicious Android program ("Kaspersky Lab," 2013, p. 1).

Since Chuli's discovery just over one year ago, Android malware has skyrocketed. According to antivirus vendor F-Secure, "Android continues to be the most targeted mobile operating system, as threats against this platform accounted for 804 new families or variants, or 97% of the new threats we saw by the close of 2013" (F-Secure, 2014, p. 22). Recent antivirus vendor reports detail the expansion in capability and flexibility of Android Remote Access Tools

(RAT, also known as Remote Administration Tool). The concept of RAT has existed and been used for legitimate system administration purposes for many years, but has been implemented within the past few years by malicious authors as a way of clandestinely gathering user information and controlling device resources from a remote location, typically via the Internet. Released in November 2012, the first-such free Android RAT was aptly named AndroRAT. Shortly after its release came separate tools known as APK binders, which allow malicious users to repackage and Trojanize legitimate Android applications with AndroRAT (Lelli, 2013, p. 1). Now, Symantec has described a recent release, Dendroid, as a HTTP RAT that is marketed as being transparent to the user and firmware interface, having a sophisticated PHP panel, and an application APK binder package. Some of its many features on offer include delete call logs, call a phone number, open web pages, record calls and audio, intercept text messages, take and upload photos and videos, open an application, initiate a HTTP flood (DoS) for a period of time, and change the command-and-control (C&C) server. And according to reports on underground forums, the author of the Dendroid APK binder had assistance writing this APK binder from the author of the original AndroRAT APK binder (Coogan, 2014, p. 1-3).

Deficiencies

The effort to address this has been positive, as several reputable tools to conduct mobile malware analysis have been created by commercial vendors and open source, such as APKinspector, DroidBox, and Dexter. APKinspector and DroidBox are open source, freely distributable tools written using the Python scripting language. APKinspector statically disassembles a target APK and displays the configurations, permissions, call graphs, classes, and Dalvik codes within it (GitHub - APKinspector website, n.d., p. 1). DroidBox uses an entirely different approach to examining APK files by dynamically monitoring the application as it runs

within an Android virtual device. It monitors information until you stop the analysis, then the following information is shown in the results: hashes for the analyzed package, incoming/ outgoing network data, file read and write operations, started services and loaded classes through DexClassLoader, information leaks via the network, file and SMS, circumvented permissions, cryptography operations performed using Android API, listing broadcast receivers, and sent SMS and phone calls. Additionally, two graphical images are generated visualizing the behavior of the package: one showing the temporal order of the operations and the other one being a treemap that can be used to check similarity between analyzed packages (droidbox - Google Project Hosting website, n.d., p. 1). According to the Dexter website, its capabilities include static and dynamic modules, heuristic result enrichment, a flexible tagging system and an API for automated processing/extending. You must register for access to their engine via login. After you have submitted the target APK file on their website, it performs an analysis remotely and returns a web page displaying statistics, permissions, system activities, services used, broadcast receivers deployed, and content providers accessed (Dexter @ BlueBox Labs website, n.d., p. 1). However, each of these focused on static and dynamic analysis of the APK files instead of examining the memory and program execution dynamically on the target device itself.

Members of the DFIR and malware analysis communities have remarked on how much time is wasted searching and researching tools (About Us, 2013, p. 1). In response to this, there have been several well-intended attempts to create web-based tool repositories and listings of available applications, such as Malware-Analyzer.com and ForensicsWiki.org, but each web site's content is only as thorough as a tool creator's drive to publish and disseminate information throughout the community on their tool, its purpose, where it can be found, and documentation on how to use it. Too often, a great tool is listed on one site but not a second, simply because the

maintainer of the second web site has never heard of the tool and the creator of the tool has never heard of the web site. The web site maintainers have made a valiant effort to both stay abreast of modern tools and market their site as a resource, but much of the community-at-large has resorted to doing Google searches to find tools instead of using or demanding a single resource be created that everyone can use and contribute to for the betterment of the community-at-large.

Additionally, developers often rely on volunteers to beta-test and validate that their tools operate as intended. Forums such as forensics and hacker conferences often serve as good recruiting locations, but also inform those who may have malicious uses in mind about upcoming applications and their capabilities. Until the need for such tools are advertised and requested by the DFIR community, there will be neither developers to create them nor volunteers familiar with why they were created and how they should perform.

Audience

The DFIR community at large is the intended audience for this information and hopefully will benefit from this research. Highlighting the importance of applications that focus specifically on examining Android memory malware is sorely needed now. By researching, experimenting, and reporting on some of those newer and existing tools to highlight current developer progress, Android anti-malware vendors, examiners, and potential application developers can be better prepared to address expanding mobile device use among the criminal element.

Literature Review

As Android's popularity grows as an operating system there have been recent efforts within the open source community and by commercial vendors in different markets, e.g. Lenovo, Nikon, and Samsung, to expand its usage beyond mobile devices, appliances, and kiosks to a different, unexpected market: netbook-sized laptops. Actions such as these trumpet Android's acceptance in the consumer mainstream as more than just a mobile device OS and emphasize how important it is to expand our understanding of and ability to resolve malicious and questionable applications targeting Android and its users. The purpose of this study is to examine recent history surrounding methods to detect running malware, discuss collection methods, highlight memory and live analysis theories, and detail research regarding the effectiveness of a specific tool, Dalvik Inspector, which was recently introduced to the forensic community as a better alternative for gathering and examining data in live Android devices.

Malware Detection History

Various innovative methods have been proposed over time and developed in some cases regarding Android malware detection. Di Cerbo et al. described a methodology for mobile application analysis whereby they combine data of another author's tool, AppAware, with the output of their tool AForensics, that retrieves permission information of third-party applications installed on a device, to indicate malicious behavior. The AppAware system uses a client-server architecture that captures and shares installations, updates, and removals of Android programs in real-time, and offers statistics to discover the top installed, updated or removed applications. AppAware likewise records permissions requested by the most common applications. Users implicitly rate applications, and thus define their acceptance, by the approach that excellent/good applications are not removed once installed, whereas applications that are removed from the

device are not liked/considered useful according to users. The information retrieved from AForensics is compared to ideal statistics gathered from AppAware, and the result helps identify whether the suspected application exhibits malicious behavior. This approach has limitations, which its creators acknowledges, in its inability to specifically identify suspicious software that exploits Android vulnerabilities and its reliability is only as good as its assumption of how the suspect application will use or abuse its granted permissions. While this approach works well in systems where a light footprint is important, its weakness suggests the need for a stronger, more direct method. (Di Cerbo, Girardello, Michahelles, & Voronkova, 2011)

Behavior-based detection methods have gained a lot of interest in recent years. Isohara et al. described a kernel-based behavior analysis system for malware detection that consists of a log collector on the Android device that records system call application activity at the kernel layer and a log analyzer on a PC that keys on a process tree of a certain application's activity based on signature pattern matching. This method provides adequate analysis and generated 3 threat classification levels for the applications analyzed, and was a good proof-of-concept for the efficacy of behavior-based analysis. (Isohara, Takemori, & Kubota, 2011)

Abela et al. described AMDA, their behavior-based automated detection system which classifies applications to one of four malicious categories: Virus, Trojan, Spyware, Exploit or as Benign using 3 unique modules. The Application Acquisition module gathers applications from various Android Markets and stores them into the application repository inside the server. Benign test applications and known-malware applications are manually downloaded from the Official Android Market and from online Android malware providers such as VirusTotal and Contagio. A web-crawler submodule iterates alternative Android markets and selected sites from a predefined list of domains to search for free Android applications in .apk format. When an

Android application has been found, the web-crawler downloads and temporarily stores the file in the applications repository. When the web-crawler has finished all the URLs in the list, it forwards the applications to the VirusTotal VMS for classification. Applications deemed suitable for test and training purposes are manually downloaded and permanently stored in the repository to ensure their validity. The Feature Extraction module uses a virtualization submodule to run the test applications in an Android version 2.3.3 environment and records kernel-space and user input system calls generated by the application in an activity log for analysis. The Behavior-based Analysis module also employs a machine learning process similar to Sahs and Khan, which used Androguard and a One-Class Support Vector Machine, to generate behavior models and a separate training phase that identifies their behavior to classify the applications as either benign or malicious. (Sahs & Khan, 2012, p. 1) While the methods introduced in AMDA are not necessarily unique, their implementation and attempts to include test applications in their training phase from outside of the Official Android Market is an excellent approach and long overdue, as many of the malicious Android applications found internationally come from alternative markets. This system also reduces reliance on the integrity of the operating system in the accuracy of its classifications, since analysis is conducted separate from the potentially infected device, although there have been successful self-contained implementations worthy of note. (Abela, Angeles, Delas Alas, Tolentino, & Gomez, 2013)

The host-based intrusion detection concept has spawned various malware analysis tools. Burgueras et al. described Crowdroid, their hybrid proof-of-concept behavior analysis tool that can be installed from the Official Android Market. It employs a crowdsourcing philosophy where user devices send non-personal, behavior-related data of each application they use to a remote server for analysis and classification. These applications could have been downloaded from

either the Official or unofficial Markets or other sources. The remote server is in charge of parsing data and creating a system call vector per each interaction of the users within their applications. Thus, a dataset of behavior data is created for every application used, and the more users install Crowdroid, the more complete and accurate the server datasets and classifications will be. This use of aggregate data collection is a sound concept, as it has been used for many years in certain traditional antivirus engines. Crowdroid is in its infancy and needs to be expanded in my opinion to include a means for notifying users of its potential discovery of malicious behavior. Likewise its creators acknowledge and realize that API call analysis, information flow tracking or network monitoring techniques can contribute to a deeper analysis of the malware than just system calls, providing more useful information about malware behavior and more accurate results. On the other hand, more monitoring capability would place a higher demand on the amount of resources consumed in the user devices. The use of the crowdsourcing community concept to obtain real application traces of hundreds or even thousands of applications improves on the use of artificially generated user input implemented by AMDA and other detection tools. Its execution joins those emphasizing the importance of data collection in both identifying malicious behavior and gathering forensic evidence. (Burguera, Zurutuza, & Nadjm-Tehrani, 2011)

Collection Methods

The ever-expanding variety of devices running Android complicates forensic collection efforts. Each device contains both hardware common to Android devices, e.g. ARM processor and RAM, and hardware unique to the device based on its function – a tablet will contain different physical components than a refrigerator. Vidas et al. described a general collection method for Android based devices, based on the premise that despite the diversity they share a

common framework that the forensic community utilizes to perform collection, also known as acquisition. Their technique repurposed the recovery partition and associated recovery mode of an Android device for collection purposes. For any device, collection was a multi-step process that required flashing a collection recovery image to the device using device-specific instructions, rebooting the device, connecting the device to a computer that had Android Debug Bridge (adb) software installed on it, then remotely executing programs from the recovery image. The programs would port forward TCP ports from the device using adb, start a receiving process on the computer, and transfer data from the storage devices to the local device TCP port using a data dumping utility and a simple program that writes the output of the data dumping utility to a socket. Vidas properly stated that while the Android framework does provide some desirable common qualities, it is unlikely that a single all-encompassing bootable image (bootimg) that properly handles all devices can be created, since devices are still unique at the hardware level employing different connectors, processors, etc., making the creation of a universal bootimg improbable. To that end, it is desirable to create a small set of collection recovery images that together support a wide range of devices and should target the older processor types so that the resulting applications will correctly execute on all backward compatible devices. These techniques have become a defacto standard commonly taught and practiced by collection practitioners and, along with a few select keystroke combinations used to reboot the various devices into recovery mode, address the aforementioned device-specific flashing instructions. (Vidas, Zhang, & Christin, 2011, p. S14-S15)

Device-specific concerns have existed since well before the smartphone explosion. In one example, Murphy described a Fraternal Clone method that allowed the forensic examiner/analyst (hereafter, examiner) to transfer all user-created files and current settings from a CDMA cell

phone into a target CDMA Fraternal Clone phone for examination. With GSM cell phones, a common solution used during examination is to clone the SIM card from the evidentiary phone and to insert the cloned SIM card into another GSM phone to complete the examination. This method is not an option for CDMA phones because the data exists on internal storage chips within the phone and not on a SIM card. Likewise, there were times where the available phone forensics tools did not allow the examiner to extract the specific data needed from the device, or where data was still encoded in a proprietary manner and cannot be decoded using typical forensic tools. A common fall back method was to document the contents of the phone screen by screen, using a camera system, however when the phone's LCD screen is broken, the phone itself is broken, or the examiner wishes to avoid physical manipulation of the phone to the extent possible, other solutions were necessary. The CDMA Fraternal Clone was used as a means to view the user created data and settings from the original phone in their native format allowing the examiner to view and work with the extracted data in a way that emulates the original phone. The process consisted of four phases. Phase one consisted of preparation of the forensic machine and the target phone by gathering all necessary data cables, ensuring the BitPim open source data extraction tool and other relevant software was installed, and resetting the target phone to its factory default to ensure no residual data would corrupt the transfer process. In phase two BitPim was used to create a full copy of the file structure of the evidentiary phone, and in phase three BitPim was used to transfer the data extracted from the evidentiary phone to the Clone phone. Phase four verified the integrity of the data transferred by comparing hash values of the files found on the evidentiary phone with those on the Clone phone (Murphy, 2009, p. 1-4). The versatility and accuracy of the BitPim software was developed over time via trial-and-error with the support of nationwide Law Enforcement and collegiate forensic practitioners and the open-

source community. Law Enforcement needs and the legal process ultimately establish the importance of what is collected and how, with the understanding that in some cases it isn't always possible to examine the data in a pristine environment or collect the evidentiary data without causing potential change to the system. In terms of Android, this is especially true since its most important operating environment is based largely in memory like many modern operating systems.

Live memory and Dalvik analysis

The forensic discipline has realized the importance of gathering and analyzing live memory dump data from an OS, as several bits of information within a system's environment are almost never stored statically on the system's storage media. To experiment with this, Thing et al. described an automated system to perform a live memory forensic analysis for mobile phones. They investigated the dynamic behavior of the mobile phone's volatile memory, and the analysis is useful in real-time evidence acquisition analysis of communication based applications. Inspired by forensic investigations on the evidence data from the instant messaging (IM) services conducted on a Windows XP system and on Apple iPhone which showed certain data from the Google GTalk application was contained solely in volatile memory, their examination focused specifically in the ability to accurately retrieve both incoming and outgoing IM messages. Thing's experimentation showed "that the outgoing messages had a higher persistency than the incoming messages. With varying message lengths, we consistently achieved a 100% evidence acquisition rate with the outgoing messages in all scenarios. For the incoming messages, the acquisition rates were 95.6% and 97.8% for dump intervals of 40 and 60 (seconds), respectively, in the scenario where each party waited for the other before responding with a reply message. In the scenario where both parties rapidly continued to send messages without waiting for a reply,

the acquisition rates were 100%, 86.7%, 75.6% and 84.4% for dump intervals of 5, 10, 20 and 30 (seconds), respectively." (Thing, Ng, & Chang, 2010, p. 8) This exercise emphasized the volatility associated with various types of mobile applications that could have gone unnoticed were it not for innovations of forensic researchers working to expand the field of knowledge and improve law enforcement evidentiary collection capability.

Mobile device forensics encompasses those smart phones and tablets running operating systems that are traditionally considered for portable devices - Android, iOS, Windows Phone, Blackberry, and other smart OSes (Tizen, Firefox, etc.). The Android platform was created for devices with constrained processing power, memory, and storage. Only a core set of applications comes with it, and vendors and developers have created thousands of independent, custom-built applications that either come with a vendor's device or that users can download and run on their devices. To maintain maximum stability and security of both the individual applications and platform itself, every Android application runs in its own process with its own instance of the Dalvik virtual machine (Ehringer, 2010, p. 1-3). The Dalvik virtual machine (DVM) is a register-based virtual machine whose components convert Java instructions and classes into a unique set of Dalvik bytecode instructions. This design is optimized to use less space, run better on low-power, low-memory devices, and averages 47% less executed VM instructions than the Java VM it replaces (Huang, 2012, slide 49). Since every application runs in its own instance of Dalvik, DVM instances must be able to start quickly when a new application is launched and the memory footprint of the DVM must be minimal. Android uses a concept called the Zygote to enable both sharing of code across DVM instances and to provide fast startup time of new DVM instances. The Zygote is a VM process that starts at system boot time. When the Zygote process starts, it initializes a Dalvik VM, which preloads and preinitializes core library classes. Generally

these core library classes are read-only and are therefore a good candidate for preloading and sharing across processes. Once the Zygote has initialized, it will sit and wait for socket requests coming from the runtime process indicating that it should fork new VM instances based on the Zygote VM instance. Cold starting virtual machines notoriously takes a long time and can be an impediment to isolating each application in its own VM. By spawning new DVM processes from the Zygote, the startup time is minimized (Ehringer, 2010, p. 5-6) and the security architecture is maintained.

Android security is implemented across the operating system and framework levels of Android and not within the DVM. Android assigns each application its own user ID and stores each application's information in its own directory with permissions set so no other application can access it unless you, the owner, grant permission for it to do so when you install the application or during its runtime. Since it is not possible for applications to interfere with each other based on the OS level security and Dalvik VMs are confined to a single OS process, Dalvik itself is not concerned with runtime security (Ehringer, 2010, p. 7).

The forensic community has long identified that physical memory analysis is vital to investigations, since it contains a wealth of information that is otherwise unrecoverable, including but not limited to objects relating to both running and terminated processes, open files, network activity, memory mappings, and more. Android's mass adoption and its projected growth make it vital that the forensics community be able to properly acquire and analyze evidence gathered from it (Sylve, Case, Marziale, & Richard, 2012, p. 1). Android does not support a memory device that exposes physical memory (as has been implemented in other OSes, e.g. /dev/mem, /dev/kmem, and ||Device||PhysicalMemory) and furthermore does not provide APIs to support userland memory acquisition applications. Acquiring physical memory

requires gaining root privileges on the device so that code can be loaded into the OS kernel to read and export a copy of physical memory. At this time, loading code into the running kernel to dump memory is the only method available to access privileged memory and the memory of all running processes (Sylve et al., 2012, p. 2-3) which is a necessary evil to the forensic sanctity of changing as little as possible within the seized evidence. The ability to gather and analyze information directly from all application DVMs along with the device's memory can provide great information in understanding the running state of a seized device and the environment within it.

Unfortunately some of the tools used on Linux-based OSes do not operate properly with Android, so other solutions need to be implemented. One method to address safely acquiring device memory is proposed by the LiME tool. According to its creator, LiME (formerly DMD) is a Loadable Kernel Module (LKM), which allows the acquisition of volatile memory from Linux and Linux-based devices, like those powered by Android. The tool supports acquiring memory either to the file system of the device or over the network. LiME is unique in that it is the first tool that allows full memory captures from Android devices. It also minimizes its interaction between user and kernel space processes during acquisition, which allows it to produce memory captures that are more forensically sound than those of other tools designed for Linux memory acquisition (Linux Memory Extractor - Google Project Hosting website, n.d., para. 1). This tool used in conjunction with the focus of this report may offer the ideal Dalvik memory acquisition and analysis suite available to Android forensic examiners.

Focus

This report is focused on research and examination of two particular tools developed as part of the ongoing efforts to find better methods of collection/analysis and to develop a better

understanding of and method for Android malware detection. First, Sylve briefed the creation of Dalvik Inspector, a free, cross-platform GUI tool under Alpha testing as of this writing, designed to provide easy analysis of Dalvik-level objects from dumps of physical RAM from Android devices. It accomplishes this through locating an application's Dalvik instance, finding and enumerating loaded Class Names, parsing and filtering Fields, Strings, Arrays, Lists, and Objects, and displaying these articles via its GUI (J. Stormo, personal communication, January 15, 2014).

There are many developers dreaming and designing new tools to settle a problem, find a new way of doing old things, make a process faster, easier, smoother, less repetitive. MWR InfoSecurity has released an innovative tool called drozer pro that implements a unique approach to APK examination. The tool consists of two applications: drozer is the core application that allows you to assume the role of an Android app, and to interact with other apps, through Android's Inter-Process Communication (IPC) mechanism, and the underlying operating system, and; drozer pro clothes the raw security assessment and exploitation capabilities of drozer in a full graphical user-friendly interface, allowing you to point and click rather than memorize commands, to take screenshots, and to annotate and save your findings (MWR InfoSecurity, 2013, p. 4). While the application seems APK-focused, its capability is based on an agent APK which can be installed on the target mobile device to monitor live application interaction as easily as it can be installed on a virtual device to dynamically monitor a suspect APK. While such action may not be as forensically sound as we would like, it is no worse than running a program on a system to capture logical memory.

Methodology

Dalvik Inspector

To establish the environment for conducting the tool analysis of Dalvik Inspector alpha (DI), various attempts to build a test environment were assembled on a variety of operating systems and hardware. Each was configured with a minimum suite of software applications necessary to approach this research project: Eclipse Integrated Development Environment (IDE), Android Development Tools (ADT) Eclipse plugin which included the Software Development Kit (SDK) and Android Virtual Device manager (AVD), Android Native Development Kit (NDK), Git source code management tool, and LiME physical memory extractor. Unfortunately, the first three test environments were inadequate, unstable, or unable to successfully complete the various stages needed to conduct the analysis, each having its own hardware or software shortcoming that could not be easily resolved within the limited time allotted.

Figure 1. Santoku Community Environment v.0.4

Ultimately the environment used was the Santoku Community Edition v.0.4 distribution (figure 1) installed on an HP Pavilion TouchSmart Notebook PC running Lubuntu v.13.04 Linux OS. Santoku Community Edition is a free, open source suite of tools and applications dedicated to mobile forensics, mobile malware analysis, and mobile security packaged and distributed for public use by viaForensics. Once its distribution address was added to Lubuntu's Debian source library, Santoku was installed like a typical Debian software installation using apt-get. The installation includes a long list of tools and dependencies including Eclipse IDE, ADT/AVD/SDK, NDK, and many others. Git and LiME were added after the fact and additional steps were taken to locate, build, and install appropriate wireless kernel modules missing from Lubuntu to activate the built-in wireless hardware capability.

Table 1. List of Android test subject devices

Phone	Android OS	Distribution	Device Type
HTC One V	Ice Cream Sandwich v.4.0.3	Virgin Mobile	Physical
Samsung Galaxy S	Jelly Bean v.4.1.2	CyanogenMod	Physical
Nexus 4	KitKat v.4.4.2	Google	Virtual
HTC Evo 4G	?	?	Virtual

Android phones listed in Table 1 were chosen as test subjects to acquire physical memory images from, and a physical memory image of an HTC EVO 4G smartphone already in the LiME format was also procured from the Digital Forensics Research Workshop (DFRWS) 2012 archive (http://dfrws.org/2012/) as an additional sample test image. The virtual device was created using AVD (figure 2) and manually run via command line using the emulator command.

Figure 2. Nexus 4 virtual device running KitKat v.4.4.2

The ability to accurately capture content of physical memory from the Android devices is important. Sylve detailed the need for an Android-specific capture tool due to tool limitations and design differences between traditional Linux builds and Google's Android. For example, Kollar's *fmem* (Index of /~niekt0/foriana, 2011) is widely used for Linux memory acquisition on Intel-based machines. Its basic operation created character device /dev/fmem that supported read and seek operations backed by physical memory, allowing *dd* and other similar userland applications to read memory from the running operating system. Internally fmem obtained the starting offset specified by the read operation, checked that the page corresponding to this offset is physical RAM and not part of a hardware device's address space, obtained a pointer to the physical page associated with the offset, then wrote the contents of the acquired page to the userland output buffer. Unfortunately, the function used to perform the page check, *page_is_ram*, does not exist on the ARM architecture. Additionally, the dd application bundled

with common Android ROMs does not handle file offsets above 0x80000000 correctly. Traditional Android dd uses 32-bit signed integers for offsets and storing 0x80000000 causes a 32-bit signed integer overflow. It then uses a system call to interact with a kernel function that expects a 64-bit signed integer. This means the kernel function receives a sign-extended 64-bit integer, which will obviously produces incorrect results. In the case of 0x80000000, this transforms the address used by the kernel function into 0xFFFFFFFF80000000. These issues make both fmem and dd unusable for memory acquisition on a number of Android devices (Sylve, 2011, p. 9-10).

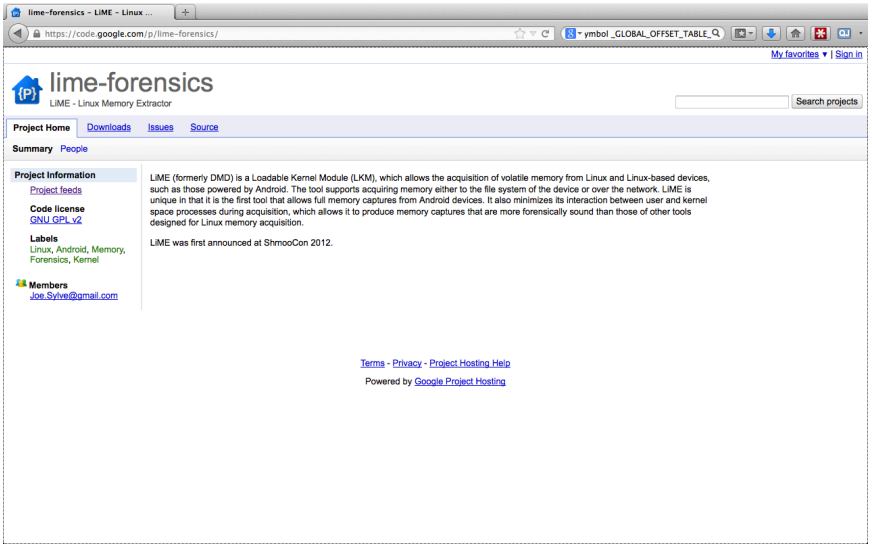


Figure 3. LiME web site

Linux Memory Extractor (LiME), originally known as Droid Memory Dumper (DMD), was designed as an Android-specific memory acquisition module to address the shortcomings of fmem and dd, but proved itself effective for all versions of Linux that employ a traditional Linux kernel and thus was renamed to reflect the expanded applicability. LiME consists of a Loadable Kernel Module (LKM) that must be uploaded into and run from the Android device's kernel memory. Every legitimate LKM is cross-compiled against the source code of the Android kernel running on a specific device, since almost all Android kernels perform module verification

checks and if any of the checks fail, the kernel will not load or run the LKM. Hence to use LiME, it must likewise be cross-compiled to the exact kernel source of the device you intend to perform the memory acquisition of.

While the cross-compile process is rather well documented with instructions provided by the product vendor (Appendix A), it is by no means a simple process as there is no guarantee the kernel source of the target device is available despite the fact that most major manufacturers do provide at least one version of it to comply with Android license requirements. Cross-compiling is basically a three-step process once a minimum environment detailed above has been assembled: acquire the kernel source for the target device; establish needed environmental variables and copy the device configuration from the target device; and, make the system and target device kernel modules. The target device kernel module is pushed to the device using Android Device Bridge (ADB) commands, and the module is installed using the Install Module (insmod) command with options tailored for a LiME output. Efforts to cross-compile LiME and install the generated LKMs in the test device kernel met with mixed results as detailed in Table 2.

Table 2. Cross-compile and module insertion results

Device	**Cross-compile**	**Insmod**	**Comments**
HTC One V	Success	Fail	Insmod output: insmod: init_module '/sdcard/lime.ko' failed (Exec format error) Dmesg log: lime: version magic '3.0.16 preempt mod_unload ARMv7 ' should be '3.0.16-g8455ee9 preempt mod_unload ARMv7 '

Samsung Galaxy S	Success	Fail	Insmod output: insmod: init_module '/scdard/lime.ko' failed (No such file or directory) Dmesg log: lime: Unknown symbol _GLOBAL_OFFSET_TABLE_ (err 0)
Nexus 4	Success	Fail	Insmod output: Insmod: init_module '/sdcard/lime.ko' failed (Function not implemented)

The LiME LKM was developed to require minimal interaction and took a simple, less invasive approach. The module parsed the kernel's *iomem_resource* structure to learn the physical memory address ranges of system RAM, calculated and translated physical to virtual address for each page of memory, then read and wrote all pages in each range to either a file (typically on the device's SD card) or a TCP socket based on the option chosen by the user when loading the module. The memory dump is written directly from the kernel to minimize userspace interaction and to eliminate the need for data copying programs like dd. The module also attempted to avoid using kernel file system buffers and network buffers to minimize volatile memory contamination during acquisition (Sylve, 2011, p. 14-15).

The ability to extract the physical memory content using LiME required the Android device under test to be rooted. This introduced several issues since the rooting process counters the traditional forensic approach of leaving a minimal footprint or causing minimal change to the target device. However, industry-leading forensic vendors, such as viaForensics and Oxygen Forensics, have asserted that rooting is an unavoidable necessity if you wish to gather important Android user content from physical memory. A critical aspect of the rooting method or application of choice concerned ensuring the process employed did not cause the subject device to reboot. This would be highly unacceptable since all relevant user data would be lost in the case of a reboot. Additionally, the techniques for root privileges differ not only for each

manufacturer and device, but also for each version of Android and even the Linux kernel. Just based on the Android devices and versions developed to date, there can be literally thousands of possible permutations. However, various open source and commercial applications, such as Kramer's open-source *Rage Against the Cage* application (http://c-skills.blogspot.com/2010/08/droid2.html) and Oxygen Forensics's suite Analyst (http://www.oxygen-forensic.com/en/features/analyst/), have shown themselves effective in causing temporary device rooting which disappears after memory acquisition when the device is rebooted, restoring it to its original state.

Before the LiME memory image contents can be examined with DI, it must be processed and converted into a format suitable for review. This was accomplished via Volatility Framework v.2.3_alpha using profile information and plug-in Python scripts provided in the DFRWS 2012 Forensics Rodeo TAR archive that held the LiME image and the Dalvik Inspector alpha TAR archive acquired from 504ensics Labs. Volatility Framework properly recognized the Evo 4G LiME image using both the LinuxEvo4Gx86 and Linuxemulatorx86 profiles as expected (figure 4), and created a SQLite database file containing the data it parsed from the image. Unfortunately when the database was opened for review using DI, it displayed no content. This was not the expected output, as is demonstrated by screenshots from 504ensics Labs (figure 5). I subsequently learned from the company that I needed to recreate the Volatility profile for this virtual device memory image, and my future attempts will include this crucial process with help from the 504ensics Labs programmers.

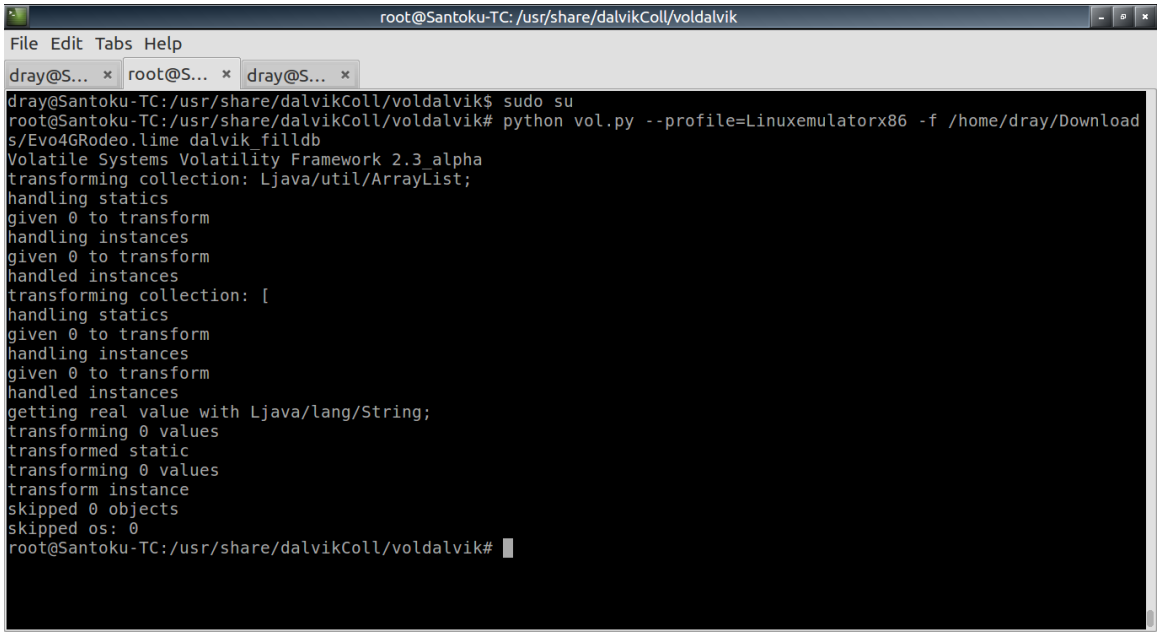


Figure 4. Volatility Framework LiME image parsing plugin output

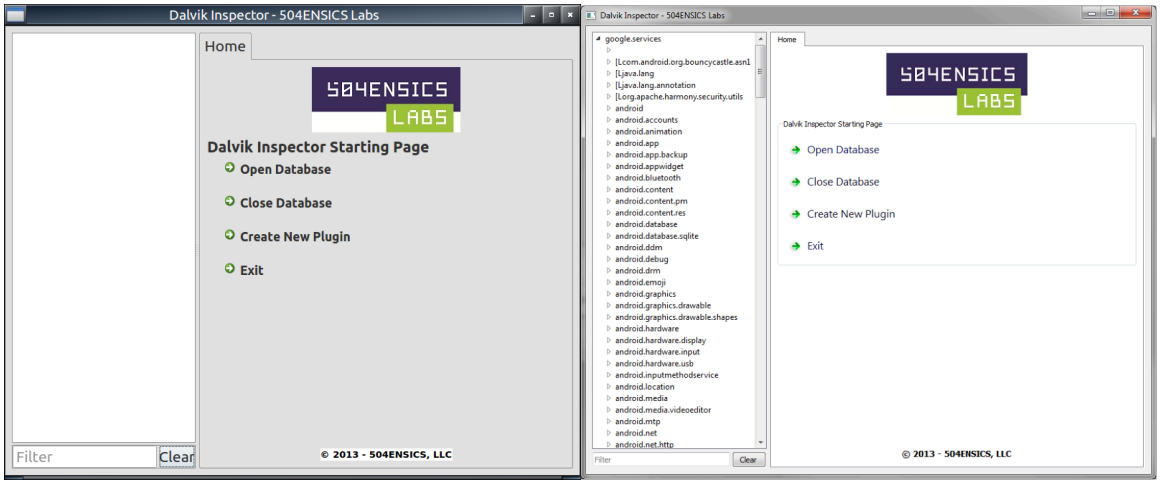


Figure 5. Research output vs. expected sample output

While no formal documentation had been written detailing DI's features, various slides showing snapshots of its functions were made available to the examiner from 504ensics Labs for review and better understanding of the tool's capabilities (J. Stormo, personal communication, March 18, 2014). The sample images describe a forensic examination looking for remnants associated with the Chuli malware.

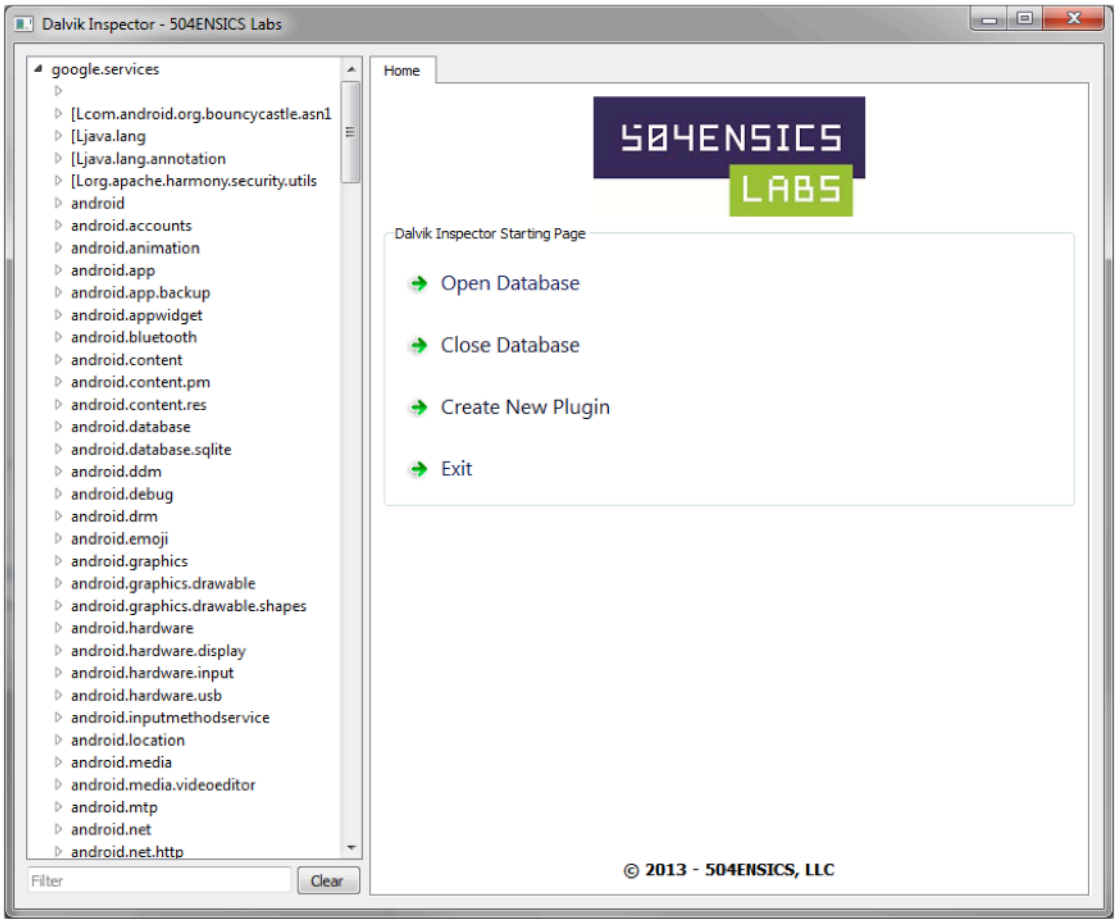


Figure 6. Chuli malware sample database content list

When a populated database is opened using DI, it should contain a listing of each of the application classes associated with the Dalvik instances running on the device (figure 6). Within each application class, its static and dynamic instance variables and their content are listed by name. In this example, information related to the device's PhoneService module are displayed and one specific record contains an Internet Protocol (IP) address (figure 7).

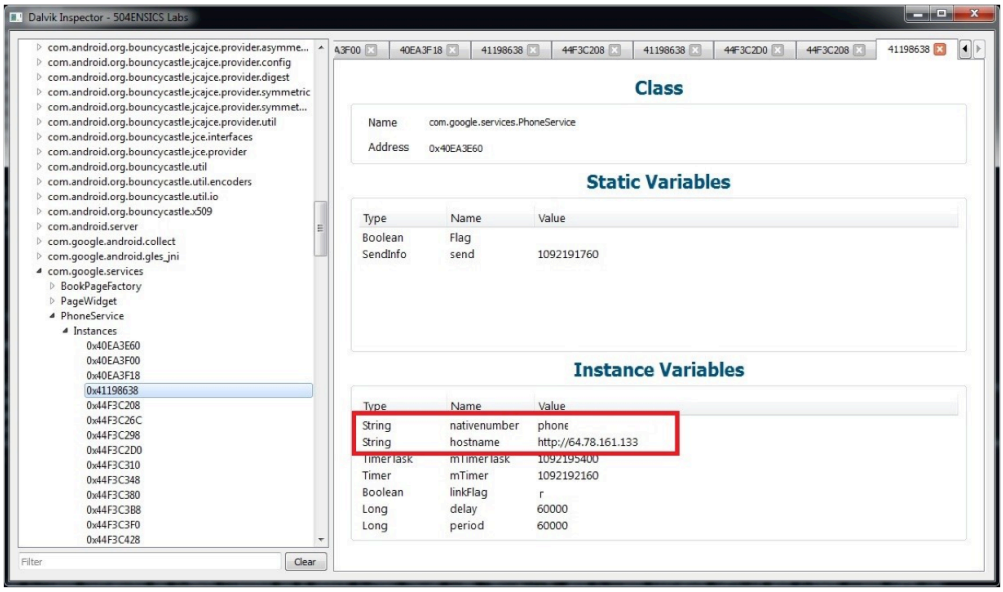


Figure 7. Chuli malware sample PhoneService module content

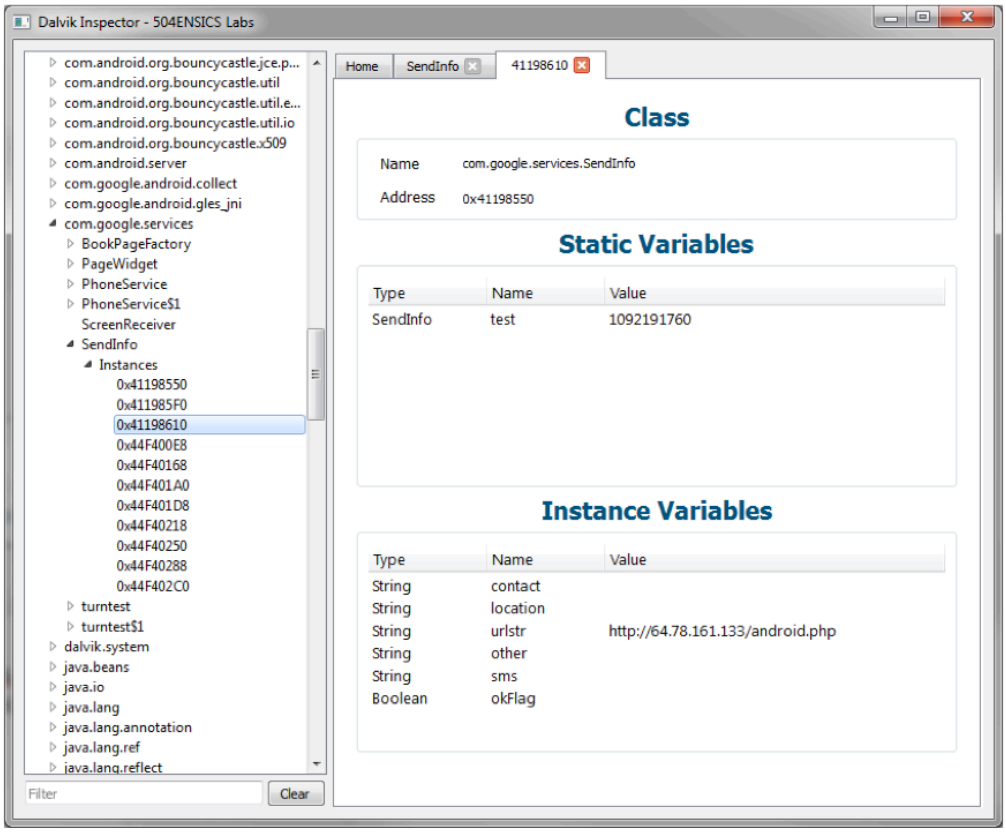


Figure 8. Chuli malware sample SendInfo module content

While such information may be associated with the device's Short Message Service (SMS)/Multimedia Messaging Service (MMS) functionality depending on the service provider, in this case further examination shows the hard-coded static IP address 64.78.161.133 which is known and documented to be the Command and Control server for Chuli malware (figure 8).

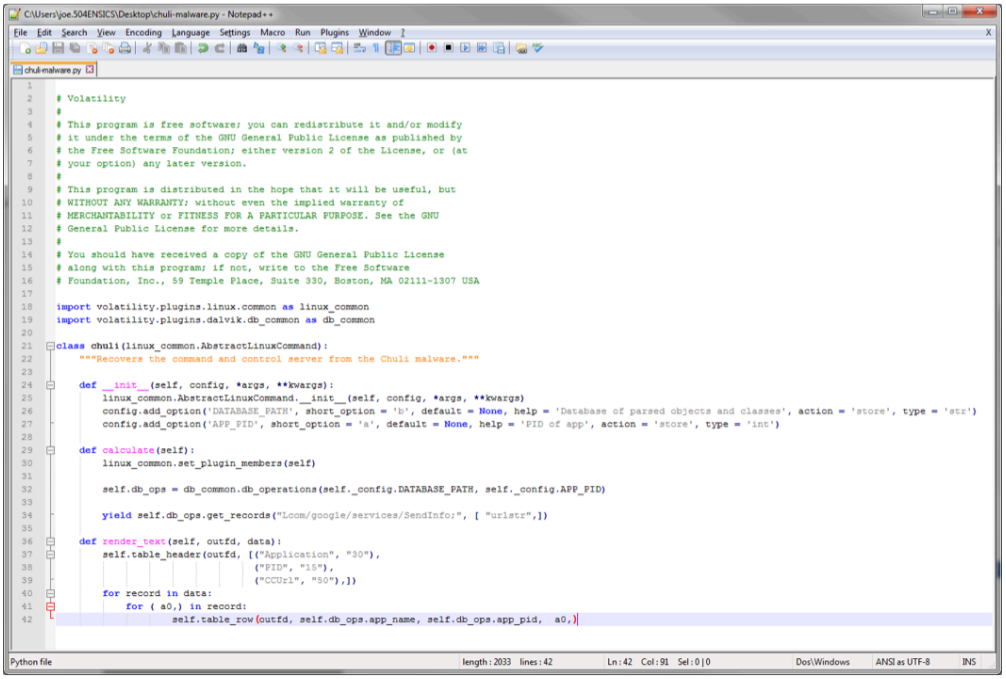


Figure 9. Chuli malware sample plugin creation output

DI has been designed to function in partnership specifically with the Volatility Framework, and included a function that allows it to generate Python code that can be used as a Volatility plugin and shared with other Volatility users. 504ensic Labs included several sample images (shown in Appendix B) demonstrating this capability using the details found while examining the database discussed above to produce a plugin that located the Chuli malware (figure 9). The plugin creation function allowed the user to graphically select those modules, classes, and variables and enter the search terms that the plugin used to examine future databases for similar Chuli malware content.

drozer pro

To establish this test environment, a clean installation of Lubuntu v.13.04 was installed on the HP Pavilion TouchSmart Notebook PC with all available system updates included. The Santoku Community Edition v.0.4 distribution ran on Lubuntu v.12.04.2 by default, so certain dependencies missing from Lubuntu v.13.04 were installed and the Santoku build script was run, which installed all of the Santoku distribution applications. Git was also installed and appropriate wireless kernel modules were built from source code and installed to restore the built-in wireless hardware capability. drozer pro installation required drozer (also referred to as the community edition) be installed first since it provided the core functionality and included the agent application. drozer required the Android SDK with AVD, and the Java Runtime Environment. Prior experience with running Android's stock device emulator proved extremely slow. For faster performance Genymotion, an alternative virtual device solution, and its dependencies were installed, and 2 virtual phones running Gingerbread v.2.3.7 and Jelly Bean v.4.2.2 were created.

drozer can be used as a standalone command-line environment application without the pro features, and its capability was assessed first as it is quite robust by itself. The application can be used to remotely exploit an Android device, by building malicious files or web pages that exploit known vulnerabilities, identified using drozer functions, to install drozer as a remote administration tool. Depending on the permissions granted to the vulnerable app, drozer can install a full agent, inject a limited agent into the process or spawn a bind shell (MWR InfoSecurity, 2013, para. 2). ADB was used to install the drozer agent APK on each virtual device, to forward the agent's output to the environment notebook, and to install the YouTube and Sieve APK applications as samples to conduct the experiment. Once installed, the drozer

agents were started and their embedded servers were activated (figure 10) allowing communication between the virtual devices and the drozer command-line console.

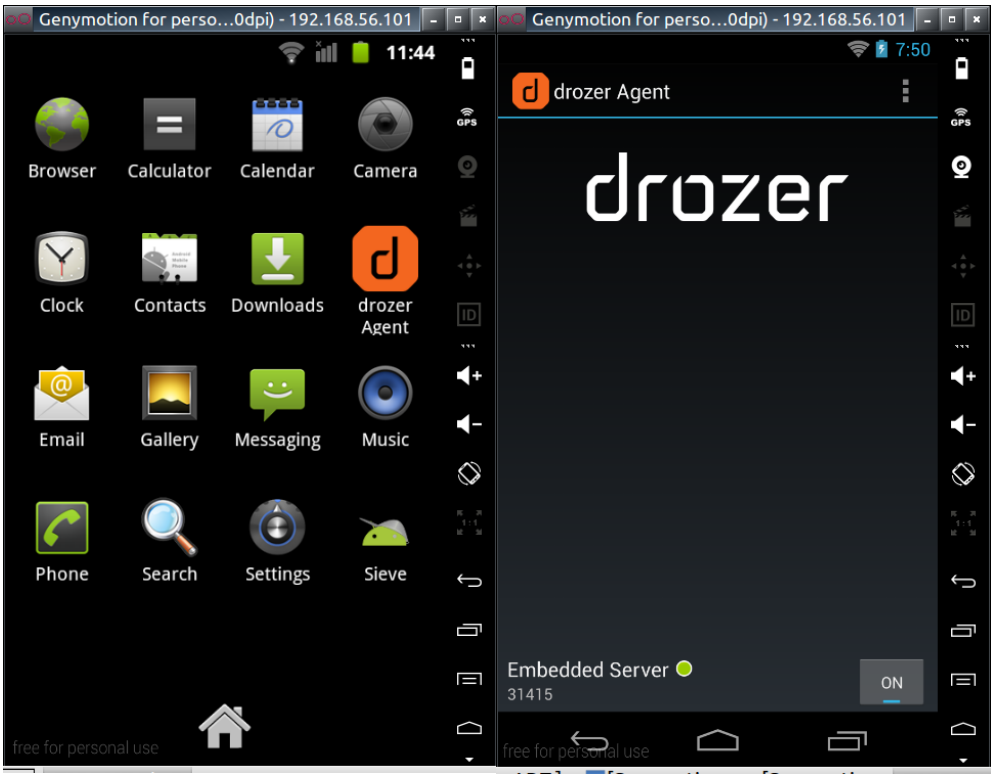


Figure 10. drozer Agent on devices and activated embedded server

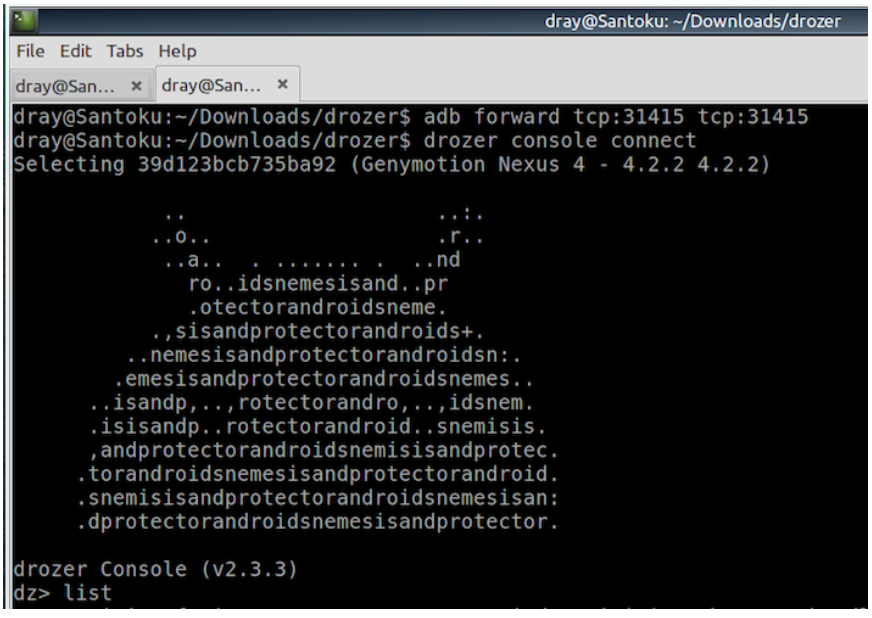


Figure 11. drozer command-line console connected to virtual Nexus 4 device

Once activated, the drozer program and its command-line console function were started, and it displayed its successful connection to the drozer Agent on the virtual device (figure 11).

Further directions on what steps to take to gather information about the target application and how to proceed in assessing its security features and weaknesses were followed from the drozer user guide. drozer provides a list of commands (table 3) and a wide range of modules for interacting with an Android device using the commands to assess its security posture.

Table 3. drozer commands

Command	Description
run MODULE	Execute a drozer module.
List	Show a list of all drozer modules that can be executed in the current session. This hides modules that you do not have suitable permission to run.
Shell	Start an interactive Linux shell on the device, in the context of the Agent process.
Cd	Mounts a particular namespace as the root of session, to avoid having to repeatedly type the full name of a module.
clean	Remove temporary files stored by drozer on the Android device.
contributors	Displays a list of people who have contributed to the drozer framework and modules in use on your system.
echo	Print text to the console.
exit	Terminate the drozer session.
help ABOUT	Display help about a particular command or module.
load	Load a file containing drozer commands, and execute them in sequence.
module	Find and install additional drozer modules from the Internet.
permissions	Display a list of the permissions granted to the drozer Agent.
set	Store a value in a variable that will be passed as an environment variable to any Linux shells spawned by drozer.
unset	Remove a named variable that drozer passes to any Linux shells that it spawns.

The modules each implement a function, e.g., listing all packages installed on the device, and are organized into namespaces that group functions and sub-functions (table 4).

Table 4. drozer built-in module commands

Namespace	Command	Description
app.activity		Find and interact with activities exported by apps
	app.activity.forintent	Find activities that can handle the given intent
	app.activity.info	Gets information about exported activities
	app.activity.start	Start an Activity
app.broadcast		Find and interact with broadcast receivers exported by apps
	app.broadcast.info	Get information about broadcast receivers
	app.broadcast.send	Send broadcast using an intent
app.package		Find packages installed on a device, and collect information about them
	app.package.attacksurface	Get attack surface of package
	app.package.backup	Lists packages that use the backup API (returns true on FLAG_ALLOW_BACKUP)
	app.package.debuggable	Find debuggable packages
	app.package.info	Get information about installed packages
	app.package.launchintent	Get launch intent of package
	app.package.list	List Packages
	app.package.manifest	Get AndroidManifest.xml of package
	app.package.native	Find Native libraries embedded in the application
	app.package.shareduid	Look for packages with shared UIDs
app.provider		Find and interact with content providers exported by apps
	app.provider.columns	List columns in content provider
	app.provider.delete	Delete from a content provider
	app.provider.download	Download a file from a content provider that supports files
	app.provider.finduri	Find referenced content URIs in a package
	app.provider.info	Get information about exported content providers
	app.provider.insert	Insert into a Content Provider
	app.provider.query	Query a content provider
	app.provider.read	Read from a content provider that supports files
	app.provider.update	Update a record in a content provider
app.service		Find and interact with services exported by apps
	app.service.info	Get information about exported services
	app.service.send	Send a Message to a service, and display the reply
	app.service.start	Start Service
	app.service.stop	Stop Service

Namespace	Command	Description
auxiliary		Useful tools that have been ported to drozer
	auxiliary.webcontentresolver	Start a web service interface to content providers
exploit.pilfer		Public exploits that extract sensitive information through unprotected content providers or SQL injection
	exploit.pilfer.general.apnprovider	Reads APN content provider
	exploit.pilfer.general.settingsprovider	Reads Settings content provider
exploit.root		Public root exploits
information		Extract additional information about a device
	information.datetime	Print Date/Time
	information.deviceinfo	Get verbose device information
	information.permissions	Get a list of all permissions used by packages on the device
scanner		Find common vulnerabilities with automatic scanners
	scanner.misc.native	Find native components included in packages
	scanner.misc.readablefiles	Find world-readable files in the given folder
	scanner.misc.secretcodes	Search for secret codes that can be used from the dialer
	scanner.misc.sflagbinaries	Find suid/sgid binaries in the given folder (default is /system)
	scanner.misc.writablefiles	Find world-writable files in the given folder
	scanner.provider.finduris	Search for content providers that can be queried from our context
	scanner.provider.injection	Test content providers for SQL injection vulnerabilities
	scanner.provider.sqltables	Find tables accessible through SQL injection vulnerabilities
	scanner.provider.traversal	Test content providers for basic directory traversal vulnerabilities
shell		Interact with the underlying Linux OS
	shell.exec	Execute a single Linux command
	shell.send	Send an ASH shell to a remote listener
	shell.start	Enter into an interactive Linux shell
tools.file		Copy files to and from the device
	tools.file.download	Download a File
	tools.file.md5sum	Get md5 Checksum of file
	tools.file.size	Get size of file
	tools.file.upload	Upload a File
tools.setup		Install handy utilities on the device, including busybox

Namespace	Command	Description
	tools.setup.busybox	Install Busybox
	tools.setup.minimalsu	Prepare 'minimal-su' binary installation on the device

The first step in assessing an application is to find it on the Android device. Applications installed on an Android device are uniquely identified by their package name. The **app.package.list** command displayed the identifier for all installed applications, including our samples which are listed by the package names **com.mwr.example.sieve** (figure 12) and **com.google.android.youtube**.

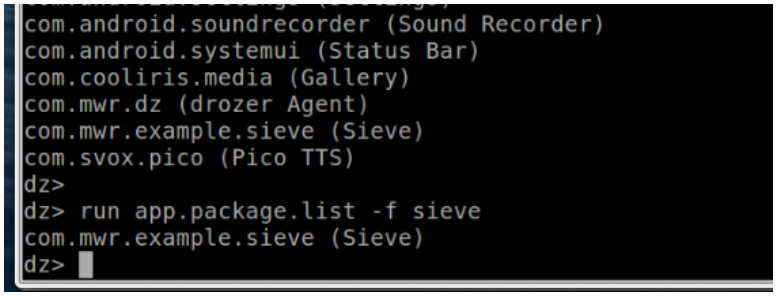


Figure 12. Sample drozer app.package.list command output

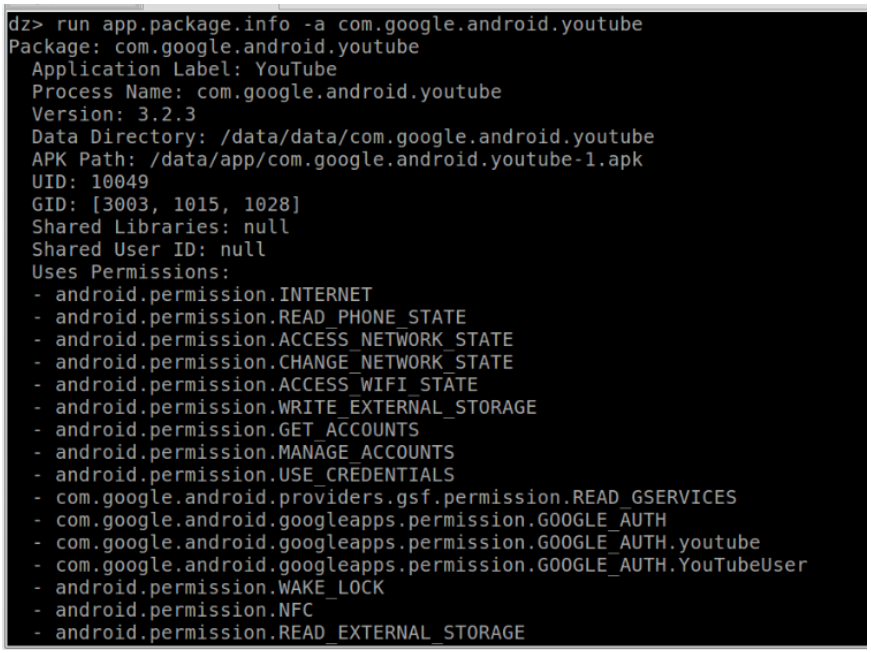


Figure 13. Sample drozer app.package.list command output

Further specific information about each application was obtained using the **app.package.info** command, which returned information such as its title, version, execution storage location, permissions, and services.

drozer was designed to identify and target vulnerabilities within an application, and can be likewise used to identify suspicious and potentially malicious services, and actions. Specific information about these factors can be gathered with the **app.package.attacksurface** command (figure 14). These indicators provide an outline of places where further investigation can be focused.

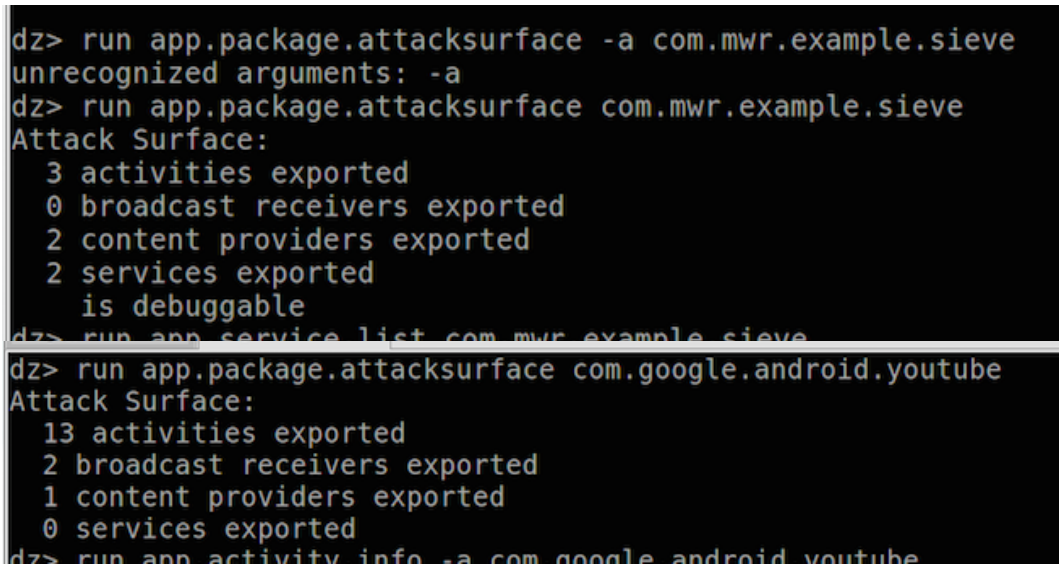


Figure 14. Sample drozer app.package.attacksurface command output

The command lists 5 areas that can be further looked into: exports, activities, broadcast receivers, services, and content providers. Exports are connectors to external applications, locations, and functions. Activities identify screens and items displayed (figure 15).

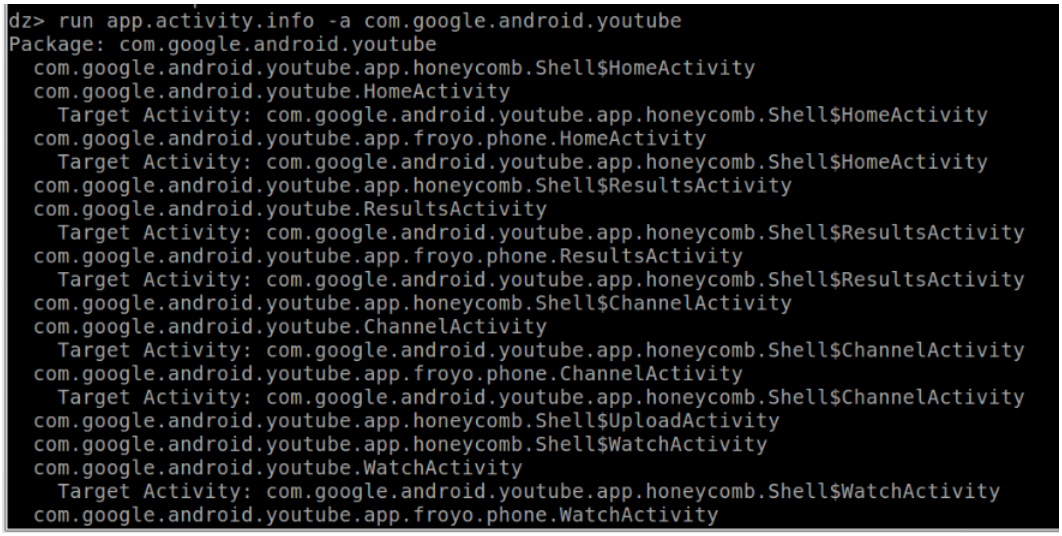


Figure 15. Sample drozer app.activity.info command output

drozer includes the capability to start an activity individually to assess or take advantage of its function to uncover more information or vulnerabilities to exploit. For example, Sieve has an activity named PWList that is expected to be associated with passwords. Drozer can instruct the system to launch the activity (figure 16), which produces a background process that generates a startActivity call. Interestingly, it causes the device to present a screen displaying user credentials (figure 17).

```
dz> run app.activity.start --component com.mwr.example.sieve
com.mwr.example.sieve.PWList
```

Figure 16. Sample drozer app.activity.start command

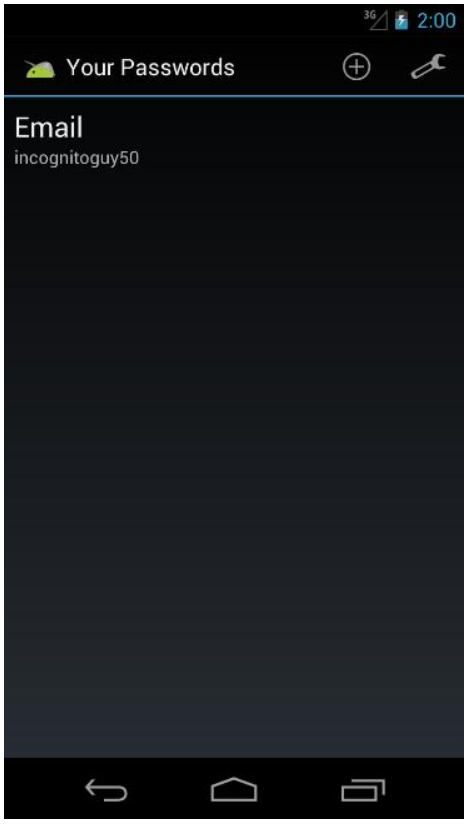


Figure 17. Sample drozer app.activity.start command output

Broadcast receivers are external devices or locations identified to receive content from the application. Services are those internal device functions that run in the background. Content providers can be any of several different types of content, such as databases, web content, other applications, etc. (figure 18). There are several different sub-functions commands, as identified

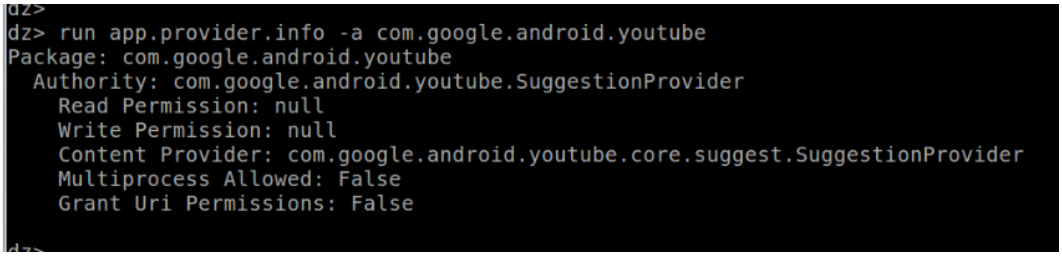


Figure 18. Sample drozer app.provider.info command output

in Table 4, that are designed to attempt to exploit content providers, particularly those associated with databases and file system content. For example, Sieve contains a database content provider

DBContentProvider, and drozer includes a command that is designed to query databases. When queried, this database showed vulnerability in its design by responding with details about its content without proper authentication (figure 19).

```
dz> run app.provider.query
        content://com.mwr.example.sieve.DBContentProvider/Passwords/ --vertical
     _id: 1
 service: Email
username: incognitoguy50
password: PSFjqXIMVa5NJFudgDuuLVgJYFD+8w== (Base64-encoded)
   email: incognitoguy50@gmail.com
```

Figure 19. Sample drozer app.provider.info command output

drozer has modules available in addition to those built-in that have been created by drozer community users and can be imported from drozer's module repository on GitHub. Current additional modules (figure 20) may include a description from their author about the module's function and options, if any. drozer also provides functionality similar to the Metasploit Framework, the popular penetration testing and hacking tool, to attempt to exploit various well-known vulnerabilities in its security assessment capability. Built-in exploit templates can be displayed using the **drozer exploit list** command, and payload templates can be displayed using the **drozer payload list** command. Those available as of this experiment are listed in Table 5.

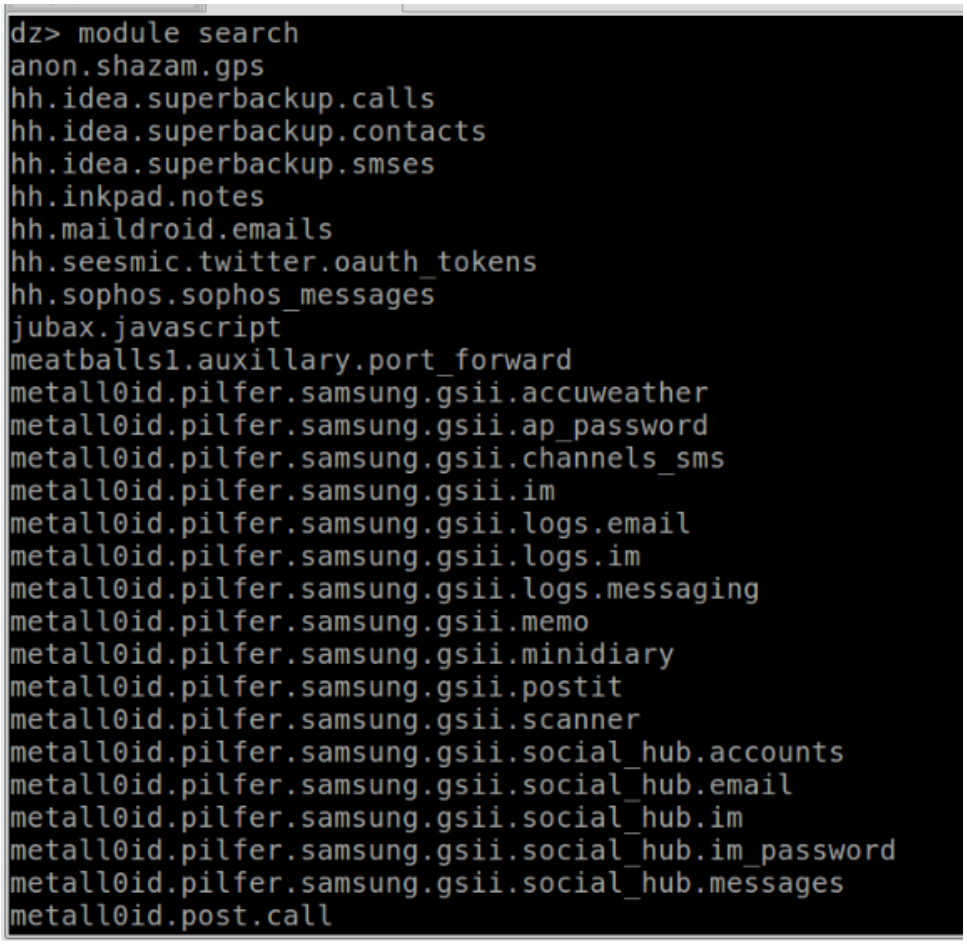


Figure 20. Sample of drozer additional modules

Table 5. drozer built-in exploits and payloads

Exploit	**Description**
exploit.remote.browser.nanparse	Webkit Invalid NaN Parsing (CVE-2010-1807)
exploit.remote.browser.normalize	Webkit Node Normalize (CVE-2010-1759)
exploit.remote.browser.useafterfree	Webkit Use After Free Exploit (Black Hat 2010)
exploit.remote.dos.remotewipe_ browserdelivery	Invoke a USSD code that performs a remote wipe on Samsung Galaxy SIII (Ekoparty 2012)
exploit.remote.fileformat.polarisviewerbof_ browserdelivery	Deliver Polaris Viewer 4 exploit files over browser (Mobile Pwn2Own 2012)
exploit.remote.fileformat.polarisviewerbof_ generate	Generate Polaris Viewer 4 exploit DOCX (Mobile Pwn2Own 2012)
exploit.remote.mitm.addjavascriptinterface	WebView addJavascriptInterface Remote Code Execution
exploit.remote.socialengineering. unknownsources	Deliver the Rogue drozer Agent over browser and hold thumbs the user will install it

exploit.usb.socialengineering.usbdebugging	Install a Rogue drozer Agent on a connected device that has USB debugging enabled
Payload	**Description**
shell.reverse_tcp.armeabi	Establish a reverse TCP Shell (ARMEABI)
weasel.reverse_tcp.armeabi	weasel through a reverse TCP Shell (ARMEABI)
weasel.shell.armeabi	Deploy weasel, through a set of Shell commands (ARMEABI)

Whereas drozer is freely distributed, drozer pro must be purchased which provides an activation code and vendor support. drozer pro adds a GUI (figure 21), allows improved visibility to applications and their structures (figure 22), allows the user to take screenshots, and makes it easier to takes notes and save your findings. After having become accustomed to controlling drozer via the command-line, using the GUI was a bit of an adjustment but did improve ability to better understand and keep track of various aspects of the overall architecture of the APK application being examined.

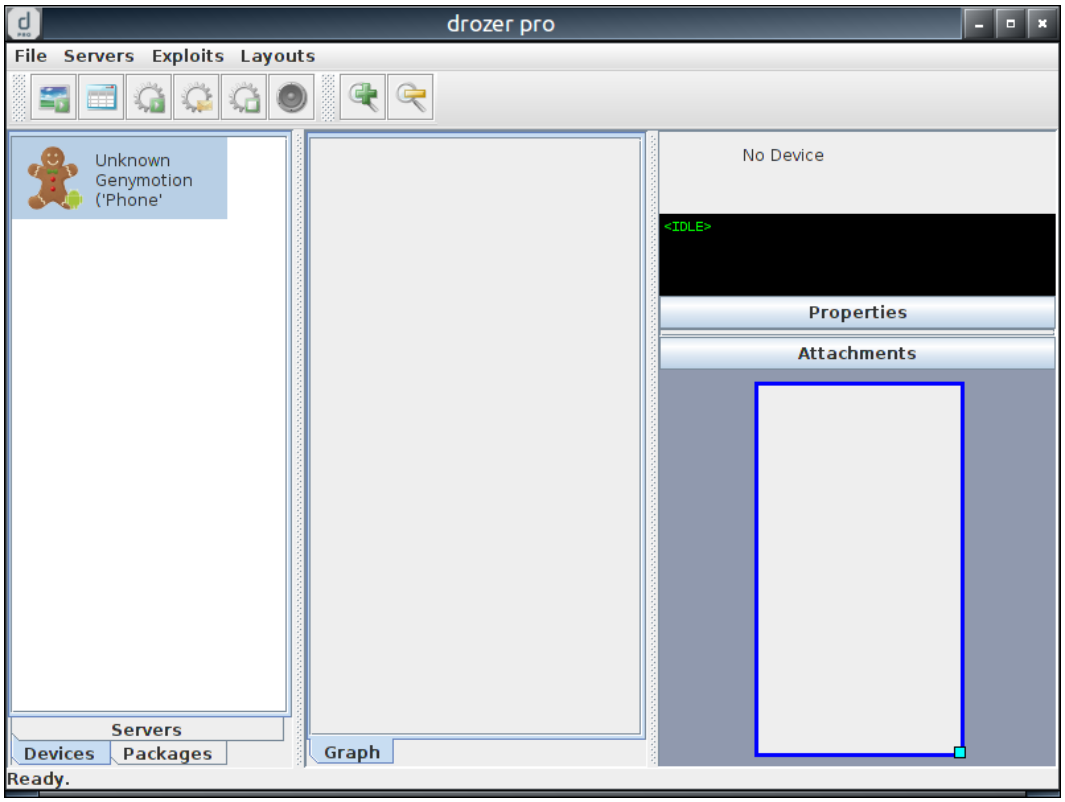


Figure 21. drozer pro interface

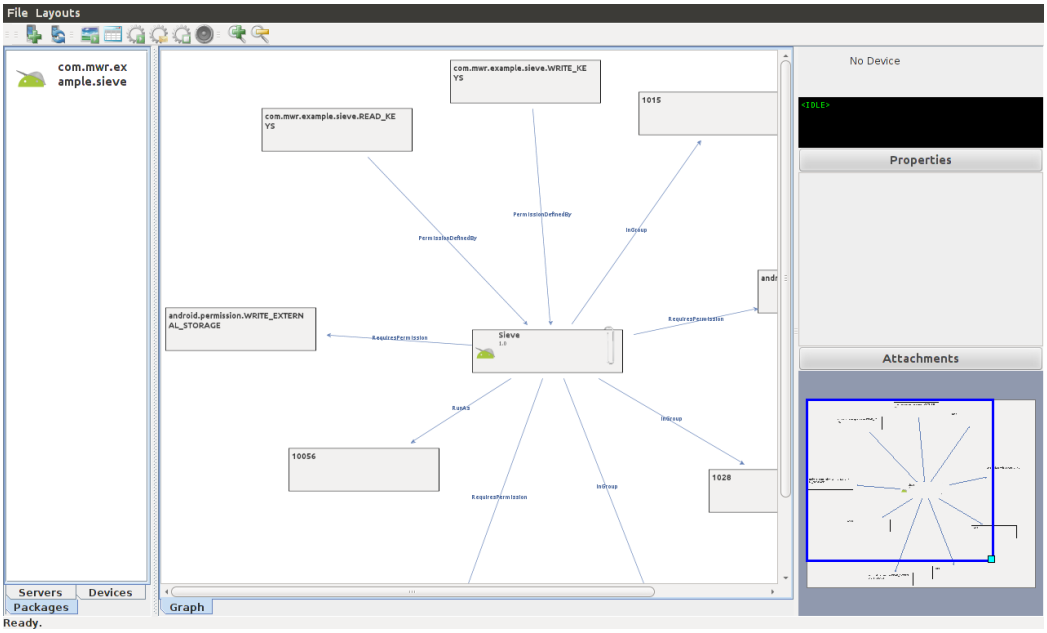


Figure 22. Sample drozer pro application structure display

Recommendations and Conclusions

Challenges

Many issues were encountered over the course of the research that spoke to the variety of challenges experienced by those actively practicing and developing in this field. Establishing a common work environment can be challenging and highly subjective, as individuals likely have a preferred operating system platform with which they are most familiar and tend to develop applications for. Originally, the research project was being conducted using the required applications and their respective dependencies installed on an Apple MacBook Pro running the latest Mac OS X operating system. While Windows and Linux are very different and highly incompatible, Mac OS X and Linux have a very high degree of similarity and one would assume that applications written for Linux should be able to function properly on OS X as well, or at least with minimal issue that could easily be resolved if they occurred at all. Unfortunately that was not the case for this project, as several problems arose requiring a solid understanding of the differences between the operating environments. When possible, bits of source code needed to be changed to make the application work properly with the OS X kernel, and even then it was largely trial and error to uncover what those changes needed to be. Despite the expertise available at 504ensics Labs and that of this examiner, ultimately the challenges in connecting to the physical test devices with ADB forced the project to abandon this environment and restart the project using a proper Linux platform. Given more time the examiner believes it may have been possible to overcome and resolve the connectivity issues, and plans to look into the problem further in the future to remove this limitation.

Another limitation encountered was based on the challenges of acquiring accurate, current kernel source code in order to complete LiME's cross-compile requirement. As

discussed, Android kernel security requires each kernel module installed to exactly match each specific Android device's kernel signature. This feature is ideal to prevent introduction of generic, rogue kernel modules from being able to run as a function of the system kernel, but it also means that each instance of LiME must be compiled using the exact version and patch level of the kernel installed on the Android device. For both physical test subject devices, the correct kernel source code was located and acquired for the proper version number, but kernel version strings did not exactly match for the HTC One V causing the kernel to reject module installation. Errors encountered with the Samsung Galaxy S module installation attempt initially suggested I didn't include certain flags while compiling. These issues occurred due to examiner inexperience, as through further research and communication with 504ensics Labs I learned that for the HTC I should have changed the version string within the kernel source's Makefile. This minor but highly important step would have produced the exact result the kernel needed for its proper module insertion. Unfortunately, a clean recompile of the Samsung's kernel source with the expected flags produced the same module installation error. The researchers and management at 504ensics Labs have likewise encountered problems like these in regards to the specific needs in building kernel modules. Going forward, the examiner must gain a better understanding of how to properly include version signatures and other needed steps to remove kernel module rejection from the process.

Advantages

Many of the issues faced during this project would have either taken much longer to resolve or could have halted the work indefinitely were it not for the extensive and open sharing of information by those individuals determined to grow and participate in the open source community. Many of the obstacles observed forced the examiner to search the Internet for

examples and situations where others had run into the same or similar problems, and in almost every case there was a web page found that contained details of someone else's struggle with the issue and responses that either suggested how to fix it or pointed to a hyperlink where someone else had detailed how they fixed the problem. Being pushed to this level of involvement in completing the various tasks was a great educational experience for the examiner, and encouraged the sharing of those lessons learned and solutions designed so others can likewise benefit.

One such invaluable solution was the discovery of the Santoku Linux distribution. Before finding this distribution when the research was being completed using OS X, all of the tools, applications, and the dependencies needed to conduct this had to be researched, located on the Internet, downloaded, in some cases compiled for the OS X kernel, and installed before the meat of the research could begin. Once the Santoku distribution was identified as a pre-installed, pre-configured alternative requiring minimal changes and ideally suited for the project, it introduced an option that allowed the project to place its focus on the work instead of spending a lot of time developing and configuring the test environment. The Santoku distribution includes many tools outside the direct purview of this research that could have easily been highly relevant in the overall subject matter. Using tools such as Dalvik Inspector and drozer in the real forensic world could have been in conjunction with conducting a forensic examination of an Android mobile device suspected of hosting malware, and tools related to mobile malware analysis are also included in the Santoku distribution. The researchers and programmers at viaForensics have put a lot of thought and care into this distribution, and the fact that it is a freely distributable resource is a great testament to their dedication to the mobile forensic industry and its practitioners.

Conclusion

Based on my observations, I found the LiME and Dalvik Inspector tools seem to be well designed, but further research needs to be done to validate them. If it were not for the many challenges that occurred along the way to produce a database file suitable for DI to parse and display, DI could have been much more fully examined for strengths, weaknesses, and overall effectiveness in the ultimate goal of forensically uncovering user and application process information that would have been unknown otherwise. I must concur that the current process to retrieve the data using these methods is very highly involved, and there is no guarantee that the return on the time invested in acquiring the data is commensurate to the effort. On the other hand, the features and capabilities of drozer and drozer pro were tested quite thoroughly and practically no problems were experienced with using these tools. If anything, the only issues encountered were associated with whether an APK could be installed on the virtual phone due to its compatibility with the Android version.

Recommendation

Dalvik Inspector's success relies on several factors in my opinion. First, 504ensics Labs' assertion of further development to simplify the physical memory acquisition process is the most important aspect of making this tool easier for the general mobile forensic practitioner, along with user training as to how to properly generate its needed particulars. Second, positive responses to 504ensics Labs' request to the community for assistance in gathering and compiling LiME kernel source samples is vital to the project success and bodes well for its acceptance as a valid tool within the community. Lastly, the forensic community's call for the success of such a tool encourages me and others like me to continue future testing, collaboration, and providing

assistance when possible to 504ensics Labs and their developers in bringing this product to full maturity where it can fill its role in the mobile forensic scheme.

In my opinion, drozer appears poised to succeed as an alternative method to revealing APK memory content. While it can be argued that this tool is technically not a memory analysis tool since it focuses on APK execution, it can likewise be argued that its ability to reveal data produced only during program execution and not statically stored within the APK code is akin to being able to read program execution memory. The small number of exploit and payload modules available was the most notable shortcoming of this tool, so continued effort by the vendor and developer community to create more will help drive this tool's success.

References

Abela, K. J., Angeles, D. K., Delas Alas, J. R., Tolentino, R. J., & Gomez, M. A. (2013). An automated malware detection system for Android using behavior-based analysis AMDA. *International Journal of Cyber-Security and Digital Forensics (IJCSDF*), 2(2), 1-11.

Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for Android. In (Ed.), *SPSM '11*. *Association for Computing Machinery* (). [Adobe Digital Editions]. Retrieved from www.acm.org

Coogan, P. (2014, March 5). Android RATs branch out with Dendroid [Blog post]. Retrieved from http://www.symantec.com/connect/blogs/android-rats-branch-out-dendroid

Di Cerbo, F., Girardello, A., Michahelles, F., & Voronkova, S. (2011). Detection of malicious applications on Android OS. In H. Sako, K. Franke, & S. Saitoh (Eds.), *Lecture notes in computer science*. *International Workshop on Computational Forensics* (pp. 138-149). [Adobe Digital Edition]. Retrieved from http://www.springer.com/computer/image+ processing/book/978-3-642-19375-0

Ehringer, D. (2010). *The dalvik virtual machine architecture*. Retrieved from http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf

F-Secure (2014). *Threat Report H2 2013*. Retrieved from http://www.f-secure.com/static/doc/labs_global/Research/Threat_Report_H2_2013.pdf

Huang, J. (2012). *Understanding the dalvik virtual machine* [PowerPoint slides]. Retrieved from http://www.slideshare.net/jserv/understanding-the-dalvik-virtual-machine

Isohara, T., Takemori, K., & Kubota, A. (2011). _2011 Seventh International Conference on Computational Intelligence and Security. In (Ed.), . *Institute of Electrical and*

Electronics Engineers (pp. 1011-1015). [Adobe Digital Edition]. http://dx.doi.org/10.1109/CIS.2011.226_

Lelli, A. (2013, July 16). Remote access tool takes aim with Android APK binder [Blog post]. Retrieved from http://www.symantec.com/connect/blogs/remote-access-tool-takes-aim-android-apk-binder

Moore, G. E. (2005). *Excerpts from a conversation with Gordon Moore: Moore's law/Interviewer: Intel Corporation*. , http://large.stanford.edu/courses/2012/ph250/lee1/docs/Excepts_A_Conversation_with_Gordon_Moore.pdf, .

Murphy, C. (2009). The fraternal clone method for CDMA cell phones. *SMALL SCALE DIGITAL DEVICE FORENSICS JOURNAL*, *3*(1), 1-8.

Sahs, J., & Khan, L. (2012). A machine learning approach to Android malware detection. In (Ed.), . *Institute of Electrical and Electronics Engineers* (pp. 141-147). [Adobe Digital Edition]. http://dx.doi.org/10.1109/EISIC.2012.34

Sylve, J., Case, A., Marziale, L., & Richard, G. G. (2012). Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 1-10. http://dx.doi.org/10.1016/j.diin.2011.10.003

Sylve, J. T. (2011). *Android memory capture and applications for security and privacy* (Masters thesis, University of New Orleans). Retrieved from http://scholarworks.uno.edu/cgi/viewcontent.cgi?article=2348&context=td

Thing, V. L., Ng, K., & Chang, E. (2010). Live memory forensics of mobile phones. *Digital Investigation*, *7*, S74-S82. http://dx.doi.org/10.1016/j.diin.2010.05.010

Vidas, T., Zhang, C., & Christin, N. (2011). Toward a general collection methodology for Android devices. *Digital Investigation*, *8*, S14-S24. http://dx.doi.org/10.1016/j.diin.2011.05.003

About Us. (2013). http://www.malware-analyzer.com/about-us/

Android pushes past 80% market share while Windows phone shipments leap 156.0% year over year in the third quarter, according to IDC. (2013). Retrieved from http://www.idc.com/getdoc.jsp?containerId=prUS24442013

Cellebrite's panel of leading industry experts identifies mobile forensics trends for 2013. (2013). Retrieved from http://www.cellebrite.com/corporate/news-events/forensic-press-releases/5-cellebrite-selected-to-deliver-mobile-phone-diagnostics-in-metropcs-stores-nationwide

Dexter @ BlueBox Labs website. (n.d.). https://dexter.bluebox.com/dashboard/analysis/

droidbox - Google Project Hosting website. (n.d.). https://code.google.com/p/droidbox/

FOR526: Memory Forensics In-Depth. (2014). https://www.sans.org/course/windows-memory-forensics-in-depth

GitHub - APKinspector website. (n.d.). https://github.com/honeynet/apkinspector/wiki/Usage-Notes

Index of /~niekt0/foriana. (2011). http://hysteria.sk/~niekt0/foriana/

Kaspersky lab identifies targeted attack utilizing malware for Android devices. (2013). Retrieved from http://www.kaspersky.com/about/news/virus/2013/Kaspersky_Lab_identifies_targeted_attack_utilizing_malware_for_Android_devices

Linux Memory Extractor - Google Project Hosting website. (n.d.). https://code.google.com/p/lime-forensics/

MWR InfoSecurity. (2013). Introduction. In *drozer pro Users' Guide* (p. 4). [Adobe Digital Editions]. Retrieved from https://www.mwrinfosecurity.com/system/assets/560/original/mwri_drozer-pro-users-guide_2013-09-10.pdf

MWR InfoSecurity. (2013). What is drozer? In *drozer Users' Guide* (p. 5). [Adobe Digital Editions]. Retrieved from https://www.mwrinfosecurity.com/system/assets/559/original/mwri_drozer-users-guide_2013-09-11.pdf

Appendices

Appendix A – LiME Cross-compile Instructions
Appendix B – Dalvik Inspector plugin creation sample images

Appendix A – LiME Cross-compile Instructions

3.0 Compiling LiME

3.1 Compiling LiME for Linux

LiME is a Loadable Kernel Module (LKM). LiME ships with a default Makefile that should be suitable for compilation on most modern Linux systems.

For detailed instructions on using LKM see the file Documentation/kbuild/modules.txt included with the Linux source.

3.2 Cross-Compiling LiME for Android

In order to cross-compile LiME for use on an Android device, additional steps are required.

PREREQUISITES

Disclaimer: This list may be incomplete. Please let us know if we've missed anything.

- Install the general android prerequisites found here.
- Download and un(zip|tar) the android NDK found here.
- Download and un(zip|tar) the android SDK found here.
- Download and untar the kernel source for your device. This can usually be found on the website of your device manufacturer or by a quick Google search.
- Root your device. In order to run custom kernel modules, you must have a rooted device.
- Plug the device into computer via a USB cable.

SETTING UP THE ENVIORNMENT

In order to simplify the process, we will first set some environment variables. In a terminal, type the following commands.

```
$ export SDK_PATH=/path/to/android-sdk-linux/
```

```
$ export NDK_PATH=/path/to/android-ndk/
$ export KSRC_PATH=/path/to/kernel-source/
$ export CC_PATH=$NDK_PATH/toolchains/arm-linux-androideabi-
4.4.3/prebuilt/linux-x86/bin/
$ export LIME_SRC=/path/to/lime/src
```

PREPARING THE KERNEL SOURCE

We must retrieve and copy the kernel config from our device.

```
$ cd $SDK_PATH/platform-tools
$ ./adb pull /proc/config.gz
$ gunzip ./config.gz
$ cp config $KSRC_PATH/.config
```

Next we have to prepare our kernel source for our module.

```
$ cd $KSRC_PATH
$ make ARCH=arm CROSS_COMPILE=$CC_PATH/arm-eabi- modules_prepare
```

PREPARING THE MODULE FOR COMPILATION

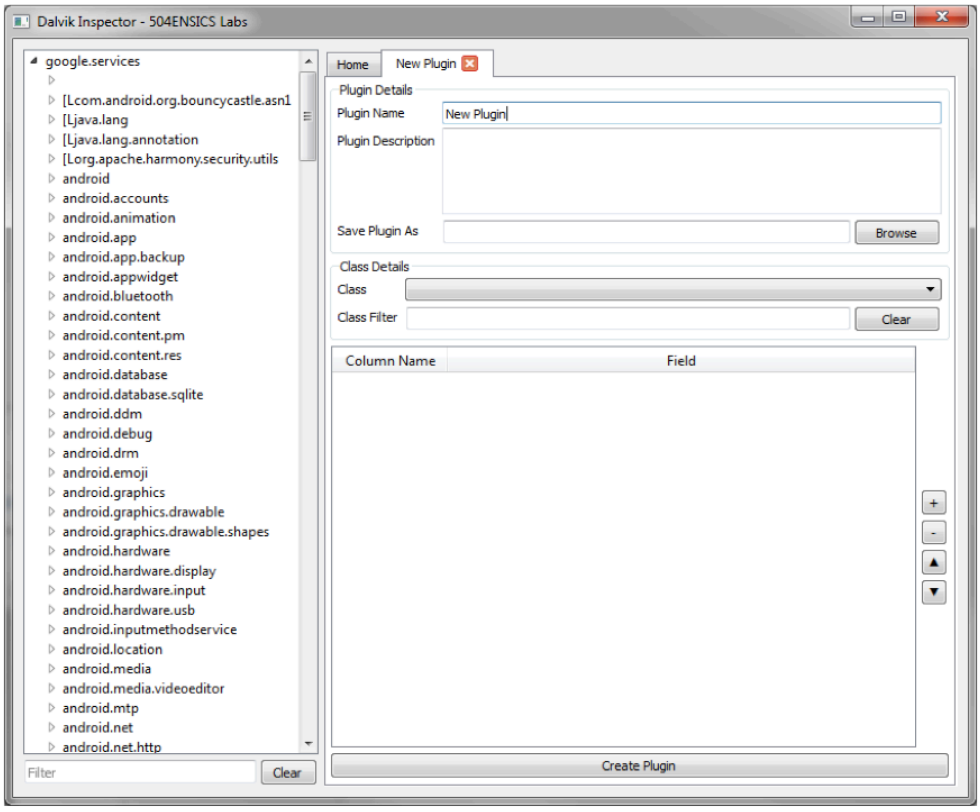
We need to create a Makefile to cross-compile our kernel module. A sample Makefile for cross-compiling is shipped with the LiME source. The contents of your Makefile should be similar to the following:

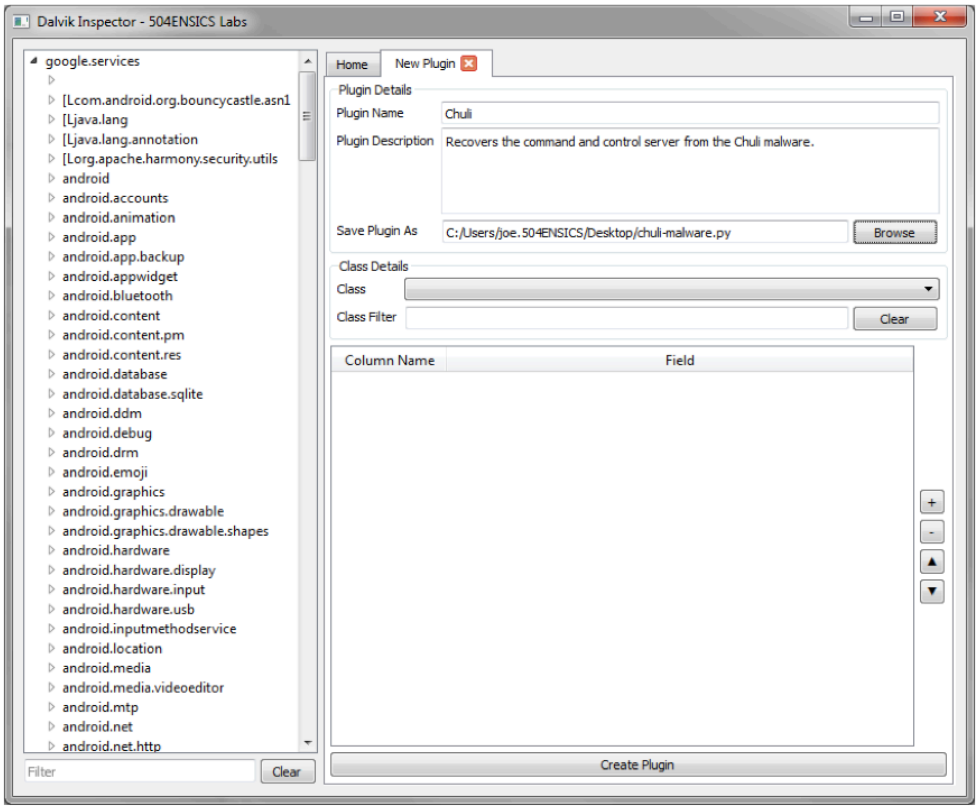
```
obj-m := lime.o
lime-objs := main.o tcp.o disk.o
KDIR := /path/to/kernel-source
PWD := $(shell pwd)
CCPATH := /path/to/android-ndk/toolchains/arm-linux-androideabi-
4.4.3/prebuilt/linux-x86/bin/
default:
 $(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-eabi- -C $(KDIR)
M=$(PWD) modules
```

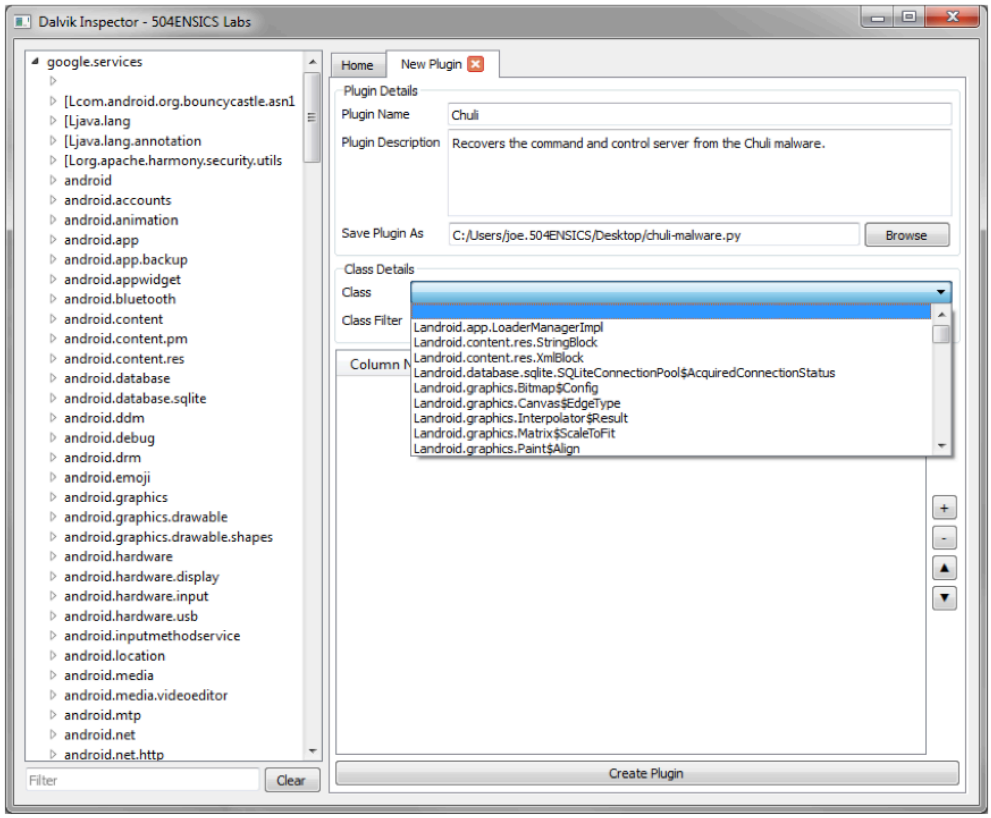
COMPILING THE MODULE

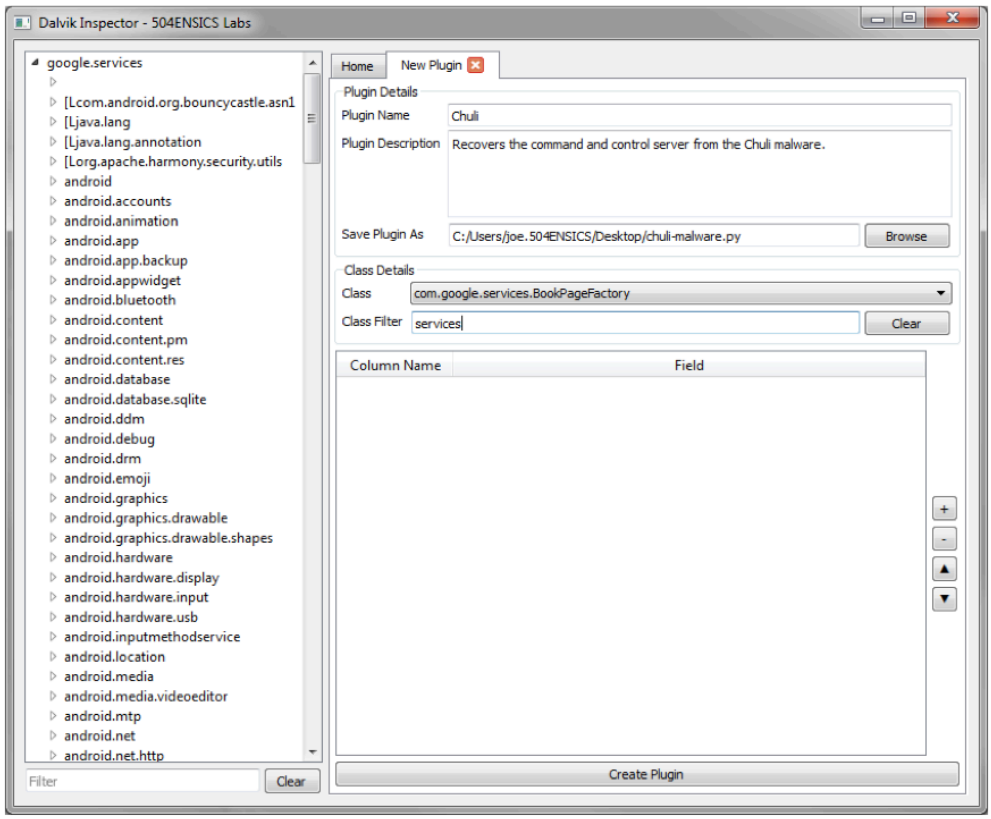
```
$ cd $LIME_SRC
$ make
```

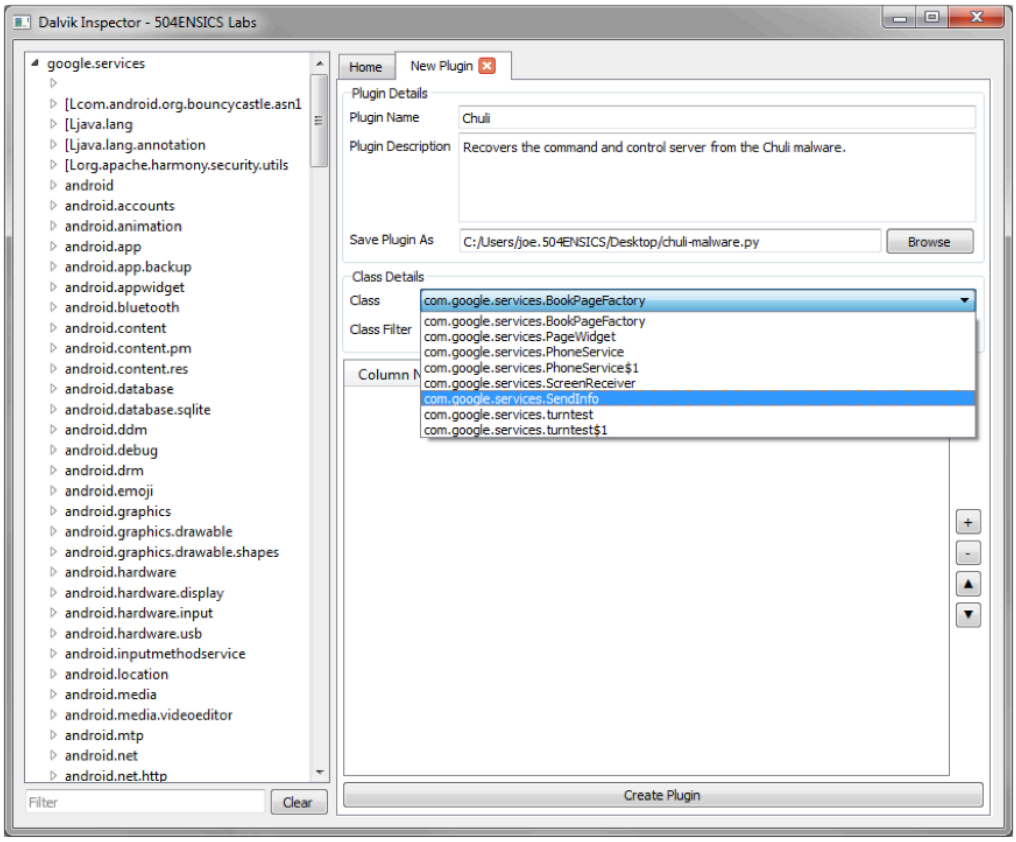
Appendix B – Dalvik Inspector plugin creation sample images

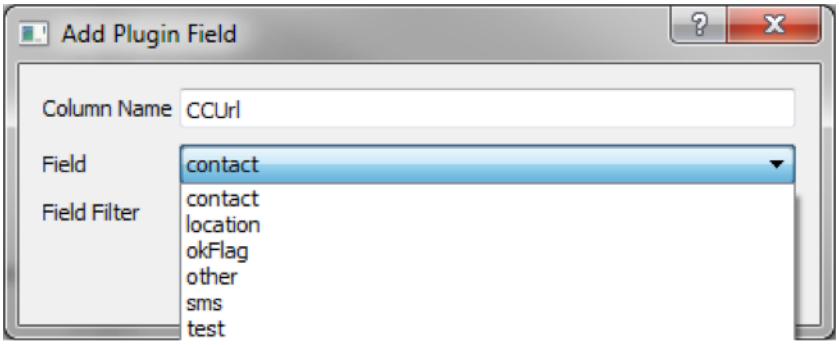


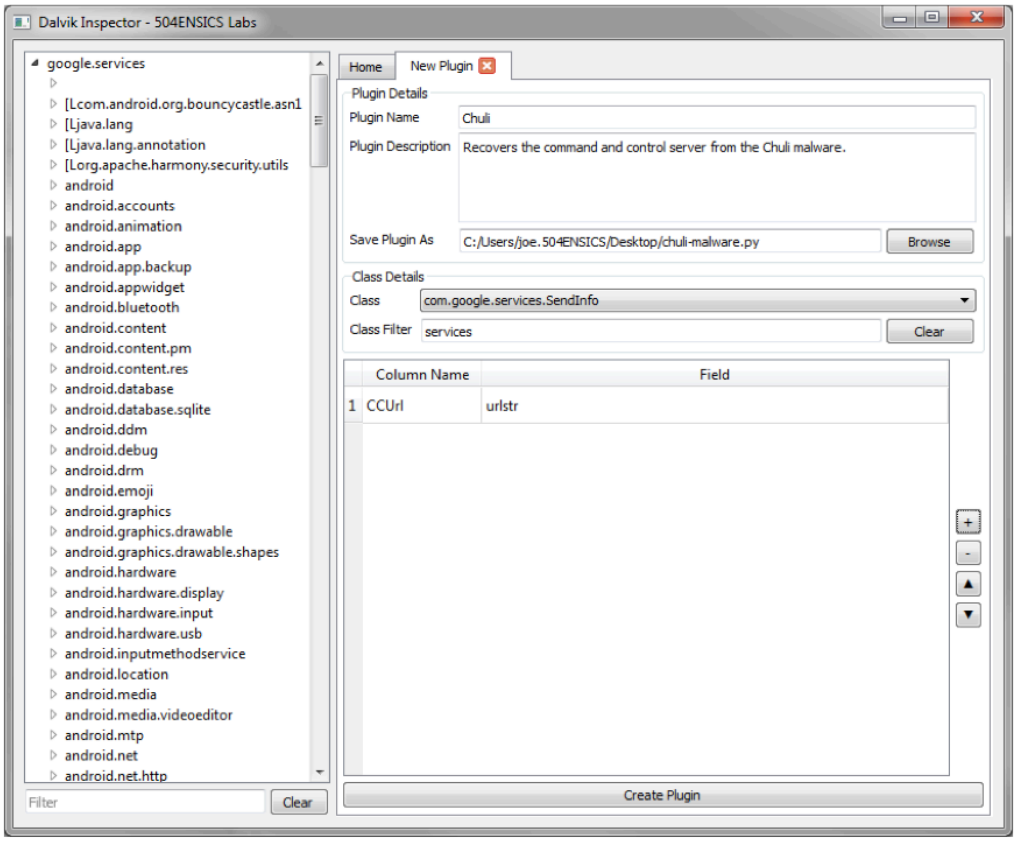



Dalvik Inspector - 504ENSICS Labs
google.services
[Lcom.android.org.bouncycastle.asn1
[Ljava.lang
[Ljava.lang.annotation
[Lorg.apache.harmony.security.utils
android
android.accounts
android.animation
android.app
android.app.backup
android.appwidget
android.bluetooth
android.content
android.content.pm
android.content.res
android.database
android.database.sqlite
android.ddm
android.debug
android.drm
android.emoji
android.graphics
android.graphics.drawable
android.graphics.drawable.shapes
android.hardware
android.hardware.display
android.hardware.input
android.hardware.usb
android.inputmethodservice
android.location
android.media
android.media.videoeditor
android.mtp
android.net
android.net.http
Filter
Clear
Home
New Plugin
Plugin Details
Plugin Name
Chuli
Plugin Description
Recovers the command and control server from the Chuli malware.
Save Plugin As
C:/Users/joe.504ENSICS/Desktop/chuli-malware.py
Browse
Class Details
Class
Class Filter
Landroid.app.LoaderManagerImpl
Landroid.content.res.StringBlock
Landroid.content.res.XmlBlock
Landroid.database.sqlite.SQLiteConnectionPool$AcquiredConnectionStatus
Landroid.graphics.Bitmap$Config
Landroid.graphics.Canvas$EdgeType
Landroid.graphics.Interpolator$Result
Landroid.graphics.Matrix$ScaleToFit
Landroid.graphics.Paint$Align
Create Plugin


Dalvik Inspector - 504ENSICS Labs
google.services
[Lcom.android.org.bouncycastle.asn1
[Ljava.lang
[Ljava.lang.annotation
[Lorg.apache.harmony.security.utils
android
android.accounts
android.animation
android.app
android.app.backup
android.appwidget
android.bluetooth
android.content
android.content.pm
android.content.res
android.database
android.database.sqlite
android.ddm
android.debug
android.drm
android.emoji
android.graphics
android.graphics.drawable
android.graphics.drawable.shapes
android.hardware
android.hardware.display
android.hardware.input
android.hardware.usb
android.inputmethodservice
android.location
android.media
android.media.videoeditor
android.mtp
android.net
android.net.http
Filter
Clear
Home
New Plugin
Plugin Details
Plugin Name
Chuli
Plugin Description
Recovers the command and control server from the Chuli malware.
Save Plugin As
C:/Users/joe.504ENSICS/Desktop/chuli-malware.py
Browse
Class Details
Class
com.google.services.BookPageFactory
Class Filter
services
Clear
Column Name
Field
Create Plugin


Dalvik Inspector - 504ENSICS Labs
google.services
[Lcom.android.org.bouncycastle.asn1
[Ljava.lang
[Ljava.lang.annotation
[Lorg.apache.harmony.security.utils
android
android.accounts
android.animation
android.app
android.app.backup
android.appwidget
android.bluetooth
android.content
android.content.pm
android.content.res
android.database
android.database.sqlite
android.ddm
android.debug
android.drm
android.emoji
android.graphics
android.graphics.drawable
android.graphics.drawable.shapes
android.hardware
android.hardware.display
android.hardware.input
android.hardware.usb
android.inputmethodservice
android.location
android.media
android.media.videoeditor
android.mtp
android.net
android.net.http
Filter
Clear
Home
New Plugin
Plugin Details
Plugin Name
Chuli
Plugin Description
Recovers the command and control server from the Chuli malware.
Save Plugin As
C:/Users/joe.504ENSICS/Desktop/chuli-malware.py
Browse
Class Details
Class
com.google.services.BookPageFactory
Class Filter
com.google.services.BookPageFactory
com.google.services.PageWidget
com.google.services.PhoneService
com.google.services.PhoneService$1
com.google.services.ScreenReceiver
com.google.services.SendInfo
com.google.services.turntest
com.google.services.turntest$1
Create Plugin


Add Plugin Field
Column Name
CCUrl
Field
contact
Field Filter
contact
location
okFlag
other
sms
test


Dalvik Inspector - 504ENSICS Labs
google.services
[Lcom.android.org.bouncycastle.asn1
[Ljava.lang
[Ljava.lang.annotation
[Lorg.apache.harmony.security.utils
android
android.accounts
android.animation
android.app
android.app.backup
android.appwidget
android.bluetooth
android.content
android.content.pm
android.content.res
android.database
android.database.sqlite
android.ddm
android.debug
android.drm
android.emoji
android.graphics
android.graphics.drawable
android.graphics.drawable.shapes
android.hardware
android.hardware.display
android.hardware.input
android.hardware.usb
android.inputmethodservice
android.location
android.media
android.media.videoeditor
android.mtp
android.net
android.net.http
Filter
Clear
Home
New Plugin
Plugin Details
Plugin Name
Chuli
Plugin Description
Recovers the command and control server from the Chuli malware.
Save Plugin As
C:/Users/joe.504ENSICS/Desktop/chuli-malware.py
Browse
Class Details
Class
com.google.services.SendInfo
Class Filter
services
Clear
Column Name
Field
1
CCUrl
urlstr
+
-
Create Plugin