OULUN YLIOPISTO
UNIVERSITY of OULU

DEGREE PROGRAMME IN WIRELESS COMMUNICATIONS ENGINEERING

INSTRUMENTATION OF A LINUX-BASED MOBILE DEVICE

Tekijä	Harri Luhtala
Valvoja	Juha Röning
Toinen tarkastaja	Thomas Schaberreiter
Työn tekninen ohjaaja	Christian Wieser

Kesäkuu 2015

Luhtala H. (2015) Instrumentation of a Linux-based mobile device. University of Oulu, Department of Electrical Engineering. Master's Thesis, 50 p.

ABSTRACT

Sensitive information are extensively stored and handled in users' mobile devices that sets challenges in terms of information security. One of the main targets of malicious mobile applications is to steal sensitive information. Mobile devices need tools and mechanisms to provide visibility how applications access sensitive system resources and handle information. Security assessment for a randomly selected application in a resource-constrained mobile environment requires an overall understanding of the target system and might involve a significant amount of work for selecting a suitable monitoring method.

This thesis presents two extensions on top of general purpose instrumentation tools. The instrumentation tools and developed extensions are executed on a Linux-based mobile device in order to monitor the behavior of applications. An application monitor extension is used for providing an overview of system resource usage by monitored application. Network monitor extension is used for analyzing content of the network traffic in real-time for selected application layer protocols. Additionally application layer data is monitored from intercepted secure connections based on user defined keywords. Developed instrumentation extensions were successfully used in a Linux-based mobile device to monitor applications' resource access and sensitive information from outbound network traffic. The approach selected for real-time network traffic analysis provided promising results while not causing significant performance problems for the mobile device.

Key words: application instrumentation, system monitoring, security assessment.

Luhtala H. (2015) Linux-pohjaisen mobiililaitteen instrumentointi. Oulun yliopisto, sähkötekniikan osasto. Diplomityö, 50 s.

TIIVISTELMÄ

Arkaluontoista tietoa käsitellään ja tallennetaan laajamittaisesti käyttäjien mobiililaitteissa, mikä asettaa haasteita tietoturvallisuudelle. Yksi tärkeimmistä vahingollisten ohjelmistojen tavoitteista on varastaa luottamuksellista informaatiota. Mobiililaitteet tarvitsevat työkaluja ja mekanismeja tarjoamaan näkyvyyttä miten ohjelmistot käsittelevät informaatiota ja arkaluontoisia järjestelmän resursseja. Sattumanvaraisesti valitun sovelluksen tietoturva-arviointi resurssirajoittuneessa mobiililaitteessa vaatii järjestelmän kokonaisvaltaista ymmärtämistä ja voi sisältää huomattavan määrän työtä soveltuvan monitorointimenetelmän valitsemiseksi.

Tässä diplomityössä esitetään kaksi laajennusta yleiskäyttöisiin instrumentointityökaluihin. Instrumentointityökalut ja laajennukset suoritetaan Linux-pohjaisessa mobiililaitteessa sovellusten käyttäytymisen tarkkailemiseksi. Application monitor -nimistä laajennusta käytetään tuottamaan yleisnäkymä tarkkaillun sovelluksen järjestelmäresurssien käytöstä. Network monitor -nimistä laajennusta käytetään reaaliaikaisesti analysoimaan tietoverkkoliikenteen sisältöä sovelluskerroksen protokollille. Lisäksi suojattujen yhteyksien sovelluskerroksen dataliikennettä tarkkaillaan määritettyjen avainsanojen perusteella. Kehitettyjä instrumentointilaajennuksia käytettiin onnistuneesti Linux-pohjaisessa mobiililaitteessa tarkkailemaan sovellusten järjestelmäresurssien käyttöä ja luottamuksellista tietoa ulosmenevästä dataliikenteestä. Valittu lähestymistapa reaaliaikaisen dataliikenteen tarkkailemiseksi tuotti lupaavia tuloksia vaikuttamatta merkittävästi mobiililaitteen suorituskykyyn.

Avainsanat: sovelluksen instrumentointi, järjestelmän monitorointi, tietoturva-arviointi

TABLE OF CONTENTS

FOREWORD

I would like to thank Christian Wieser for helping with research methods at the start point of this thesis and keeping things happening in schedule due to follow-up meetings, and Pekka Pietikäinen for providing valuable feedback for implementation related issues. I would also like to thank Prof. Juha Röning for offering interesting thesis writer position at the OUSPG research group at the University of Oulu and for supervising this work.

ABBREVIATIONS

ACL	Access Control List
AVC	Access Vector Cache
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CA	Certificate Authority
CN	Common Name
CPU	Central Processing Unit
DAC	Discretionary Access Control
DNAT	Destination Network Address Translation
GID	Group Identifier
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IMEI	International Mobile Equipment Identity
IP	Internet Protocol
IPC	Inter-process Communication
LSM	Linux Security Module
MAC	Mandatory access control
MITM	Man-in-the-middle
NAT	Network Address Translation
OS	Operating System
PID	Process Identifier
SAN	Subject Alternative Name
SNI	Server Name Indication
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TID	Thread Identifier
TLS	Transport Layer Security
UDP	User Datagram Protocol
UID	User Identifier
URI	Uniform Resource Identifier
XMPP	Extensible Messaging and Presence Protocol

1. INTRODUCTION

Mobile devices are available on the market from entry-level devices without Internet connectivity to high-end devices with relatively huge amount of features and computing power. For example in mobile handset category, devices classified as smartphones, have overtaken lead position from entry-level cell phones in many countries [1]. According to the first quarter of 2014 mobile threat report by F-Secure labs, mobile malware continues to focus on the Android devices. Especially third-party application stores provide high number of malicious applications [2]. The majority of malicious mobile applications are intended to steal user's personal information and have financial targets [3]. Consequently, the security and privacy community and the industry have an increased interest to focus on mobile devices' security issues.

Early mobile devices used to be closed systems, where functionality was not possible to be extended by installing third-party applications and the only Internet connectivity option was a trusted cellular network [3]. The security aspect of mobile devices were totally different what it is nowadays. Rich feature set and multiple connectivity options of the recent mobile devices have added challenges in terms of information security. In addition, mobile devices store great amount of personal and company confidential information including emails, passwords and location.

Mobile device ecosystem consists of several stakeholders, who have a different interest for device security. For example device user wants personal information to be stored safely and device manufacturer prevents tampering of the device so that any hardware parameters cannot be changed. Operators protect their business model by offering subsidized devices and application developers are interested in protecting source code of the application and mobile platform providers protect against malicious application installations. [3]

Security vulnerability infection mechanisms are moving very quickly from desktop computers to mobile devices. Mobile devices providing security-critical functions and lacking proper security implementation altogether makes them vulnerable for security threats. Mobile devices need to provide software and hardware based device security mechanisms. Hardware based mechanisms are created in order to prevent simple device tampering attempts. Software based mechanism protect against external threats. The most crucial software based protection is process isolation, usually called sandboxing, which isolates each application into a private execution and storage environment. [3, 4]

Linux-based mobile devices implement applications access control restrictions using different approaches. One platform might count solely to the traditional style permissions for all third-party applications, while other platforms have dedicated user identifier assigned per application. Per application user identifier approach enable specific access control definition for each application. Usually native applications provided by platform manufacturer run with privileged user permissions. [5]

Mobile devices are extensively integrated with the Internet using cellular and wireless technologies that make them an attractive and susceptible target for criminal intent to exploit them. Every once in a while appears that an application with a malicious intent has managed to pass verification process of an application store. Obviously, the verification process cannot reject every single malicious application. On desktop computers the most common method of antivirus scanner uses a classical signature based malware detection mechanism. The signature detection is based on already identified characteristics of malware samples. Malware detection on mobile devices is a challenging task due to limited processing power. The signature based approach is not the most suitable option for mobile devices, because matching algorithm running on background cause heavy burden on CPU and faster battery exhaustion [6]. In considering these issues, mobile devices should have security mechanisms against malicious applications and be equipped with an ability to monitor applications within the device in order to discover malicious behavior, and identify sensitive information leakage. To address this shortage, this thesis will discuss software instrumentation methods and tools for a Linux-based mobile device. Monitoring extensions are developed on top of the existing instrumentation tools and used for providing input data for a security assessment whether an application is acting against its expected behavior. In addition, sensitive information are monitored from the Internet traffic based on defined keywords.

2. MOBILE SYSTEM SECURITY

The security is an important part of the mobile devices. Functionality of security mechanisms need to be addressed in every layer of software stack and as well on hardware level [3]. In the Linux-based operating system a good level of security is already built-in and can be optionally extended by security modules. This chapter gives an overview of information security and discuss software and hardware based security solutions used in Linux-based mobile devices.

2.1. Information security

The history of information security begins with a need to secure computer's physical location, hardware and software. During the early years of information security the primary security threats were physical theft of equipment, espionage against the products of the system and sabotage. To maintain national security during the World War II eventually lead to sophisticated security solutions. The movement towards security that went beyond safety of physical locations began with a study sponsored by advanced research project agency (ARPA). The study was focused on a process to secure classified information systems and it attempted to define necessary mechanisms for protecting them. The Internet was made available for general public in 1990s, which brought connectivity between computers. Early Internet era deployment was treating security as a low priority and relied on a security provided by data centers. However, ability to physically secure networked computers was lost and stored information became more exposed to security threats. [7 p. 3 - 8]

The information security can be defined as protection of information and its critical elements. The industry standard for information security model is called confidentiality, integrity and availability triangle. It defines information confidentiality as protection from disclosure or exposure to unauthorized persons or systems. The confidentiality of information is high when it is a personal information. When the information is complete and uncorrupted, it has integrity. Corruption of information can take place when it being transmitted, stored or intentionally modified by a computer virus. Information corruption is not necessarily caused by external forces. For instance, noisy transmission can cause data to lose its integrity. The availability enables authorized persons and computers to access information and receive it in the required format. [7 p. 8 - 15]

Information system consists of components that enable information handling, which are entire set of software, hardware, data, people and networks that each have own security requirements. Software comprises applications, operating system and utilities. Exploiting of errors in software implementation forms substantial portion of attacks against information system. Hardware provides technology that executes the software and provides interfaces for information handling. Security policies define hardware as a physical asset. Data is often the most valuable part of information system and it is the main target of attacks. Data storage is likely to use database management systems, which should utilize all available security capabilities to improve overall information system security. The people have always been a threat for information systems that refers to vulnerabilities caused by system users. The security leverage need of information system is mainly caused by networking. The network security mechanisms are essential part of the information system security. [7 p. 16 - 19]

2.2. Threats and attack vectors

Mobile devices encounter various threats depending on usage environment. For instance, threats might appear in a form of another malicious application or an adversary on the same local network. Vulnerable application implementations can be suffering in many fields, these include insecure data storage, insufficient transport layer protection, outdated cryptographic algorithms and many more. A security weakness exists when application developer assumes that other applications in the system are not able to access sensitive data storage. However, a malicious application might have gained enough privileges to read file system's sensitive storage. Same sort of weakness for sensitive data occurs when an application developer unintentionally uses a system mechanism, which stores data temporarily for easily accessible locations. For instance, keyboard press caching or copy-paste buffering might use storage locations available for unprivileged users. Mobile applications tend to lack proper implementation of transport layer security or it is enabled only for authentication phase. Thus adversary in the same local network is able to monitor network traffic and capture sensitive data. Application data encryption implementation might use cryptographic algorithms, which are commonly known to have a significant weakness or does not fulfill modern security requirements. Alternatively custom made algorithms have been used for data encryption, which does not give enough protection. Application can accept input data from various sources without performing proper input validation and can allow an inter-process communication with other applications. For instance, business application handling sensitive data should restrict communication only for trusted applications, because sensitive data passed through inter-process communication may be read by third-party application in certain conditions. [9]

Attack vectors are methods used to get into device that enable intruders to exploit vulnerabilities in the device. Mobile device attack vectors can be roughly classified in vulnerabilities that require a physical access to the device and vulnerabilities that can be exploited remotely. Remotely exploited attack vectors can be further classified technical and non-technical vulnerabilities. Mobile web browser is a good example of technical and software based attack vector, which has led to various exploited vulnerabilities in the recent past. The non-technical attack vector tricks the user into overriding technical security mechanism. [5]

Drive-by download is an attack method taking advantage of bugs in a software, usually in web browsers. Visiting a compromised web page is enough to initiate background download of malicious code. Websites are used to exploit known vulnerabilities in the web browser or plug-ins and execute attacker's code. Initial download is often small and its job is to pull the rest of the malicious code to the target device. This kind of attack has significant impact for the system security, because the attacker can execute code without user's knowledge. A social engineering attack targets primarily to a human element of the system without gaining a physical access to user's device. The user of the device is tried to be convinced to either download a malicious application or access malicious content on the website. The user may easily install malicious application that requests extensive permissions for system resources, and let the application bypass sandbox restrictions. Advantage over desktop computer environment in the most mobile systems is that user is able to review requested application permissions and then proceed in installation process. [8]

2.3. Platform security

A mobile platform consists of operating system (OS) kernel and middleware components [3]. The middleware provides system libraries and services that are used by system components and applications. Inter-process communication (IPC) framework is responsible of providing communication facilities between applications and services. The IPC framework enables communication between software components, which are hence able to utilize functionalities exported by each other. Application requests to system resources are typically handled by a corresponding system service. The system service receives a request through IPC and then performs actual resource access, for instance an information request for a peripheral device. Security model in mobile platforms are based on software isolation, access control and cryptographically signed applications.

The software isolation is also known as sandboxing of application's execution environment. Sandboxed application is confined so that the application is not able to directly access another application's data storage. It is also possible to divide applications for separated domains based on application type. Domains may define different restriction levels for business applications that are used to handle confidential data and another level for third-party applications. Application process address spaces are also separated from one another, additionally process address space randomization may be used and memory areas can be set as non-executable. Intention to limit memory execution addresses is to prevent malicious applications from modifying its code and affecting to execution of other processes. A runtime buffer overflow exploit takes place when adversary utilizes software vulnerability and modifies a call stack return address to a memory segment containing injected code by attacker. Memory page configuration as non-executable prevents attacker injected code execution. [3]

The access control mechanisms implement permission based model where system resources usage can be limited. Platform provider sets permissions required to access exposed APIs of system services. Also third-party developed services define required permission to access API. Large number of dedicated permissions allow accurate control policy, but might be difficult to understand. Respectively coarse permission control levels might violate least privilege principle, in which application must be able to access only resources that are necessary for its operation. [10]

A software distribution model varies across mobile device platform manufacturers. Several options exist for the distribution model. Applications can be installed from platform provider hosted application store. In addition, multiple auxiliary stores and direct application loading to the device might be allowed, the latter one is also called application sideloading. The application signing is done by application store operator after the application is verified to fulfill a publication criteria. Additionally a developer signing is used to prove the origin of the application at the time of updating a new version of an existing application. An application distribution package contains a manifest file that defines which protected system APIs are required by application. Platform application installer component is responsible to check the signature and the manifest file and then verify that application signing authority is allowed to grant requested permissions. Additionally user might be prompted to approve access to user sensitive data, such as address book and calendar. Finally application installer assigns requested permission to the application in the installation process. [3]

2.4. Access control and enhancements

Computer system security is growing problem, which is seen by endless stream of security vulnerabilities. Security research has produced numerous access control mechanisms to improve system security. Linux security model assigns a unique identifier for every user and group, which primary use is to determine ownership of system resources and control process permissions accessing those resources. Traditional file permission model is sufficient for most applications, it defines permissions for each file and directory. However, often finer control over permissions for users and groups is needed. The Linux security module (LSM) framework address this problem by providing a general purpose framework to Linux kernel, which allows security modules to be implemented as loadable kernel modules. [11, 12]

File system and process privileges

An extension for the traditional file permission model is called an access control list (ACL), which allows permission to be specified per user or per group. Minimal ACL configuration is equal for the traditional model, where permissions are defined separately for user, group and other. The ACL is actually a series of entries, each defining permissions for individual user or a user group. An ACL entry consist of tag, qualifier and permission fields. The tag specifies whether entry applies for a single user or a group of users. The qualifier field is an optional one, and it is used to define certain user or a group identifier for the entry. [11 p. 319 – 337]

Discretionary access control (DAC) is a standard security mechanism in a Linux based operating system that runs each process under specified user and group. The DAC has a disadvantage when vulnerable process gets exploited, and the attacker gains access to all resources that run under the same user and group as the exploited process. This issue takes place because of coarse granularity of permissions in the DAC system. In addition, owner of a resource can decide how resource can be accessed by other processes.

Linux capabilities scheme is used to divide traditional all-or-nothing process capability model into individually enabled capabilities. Capabilities are used to allow certain program to perform privileged operations. The traditional model divides processes in two categories; processes which bypass all privilege checks and processes whose privileges are checked according to user and group IDs. Processes bypassing all checks are called super-user and its user ID is set to zero. The traditional model have a coarse granularity over the process privilege control and it does not have a mechanism to permit a single privileged operation for an unprivileged program. Allowing certain privileged operation by changing effective user ID temporarily to the super-user, also permits process to perform other privileged operations as well. This kind of privilege control mechanism opens number of possibilities for malicious users to perform unwanted operations. In the modern Linux system an application process can have one or more capabilities, which are grouped for permitted, effective and inheritable sets. The permitted set limits for capabilities which can be added for effective and inheritable sets. If process drops a capability from its permitted set, it can never acquire it back again by itself. The effective set is the one, which kernel uses for privilege checking for the process. [11 p. 797 – 806]

Security enhancements

Mandatory access control (MAC) provides fine-grained permission levels that can restrict damage. The MAC is a system where operating system is used to constrain processes performing an operation for system resources. A security policy defines restrictions how resources can be accessed, it is loaded at the startup of the system. Typically system administrator is the only user who can change the security policy. [12, 14]

Various Linux security modules (LSM) have been implemented, which provide security improvements such as fine-grained MAC and reduction of an attack surface. Linux security modules are security extensions, which are hooked on important security-critical points of the Linux kernel. Linux security module framework provides hooks into kernel components that can be utilized by LSMs to perform access control checks. Currently only one LSM can be enabled at the time, although stacking support of multiple LSMs have been under discussion in the community several times. The security module hooks are implemented in a way that existing frameworks such as standard DAC are not disturbed. LSM hooks are not invoked when functional error is detected or classical security check denies requested operation. [12, 14] Linux security module architecture for secure enhanced Linux is presented in Figure 1.

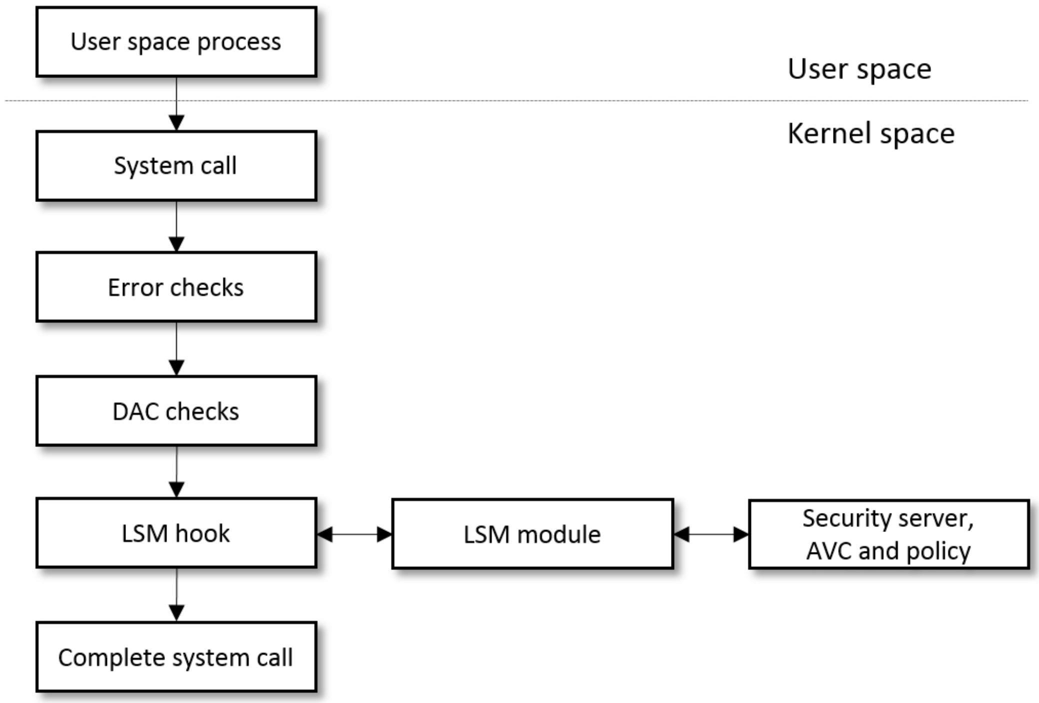
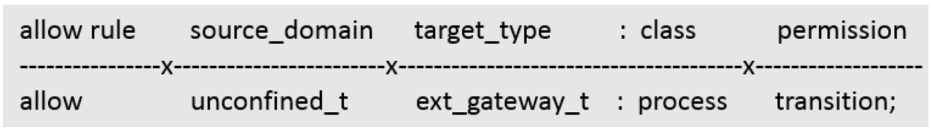


Figure 1. Security enhanced Linux architecture

Security-Enhanced Linux (SELinux) is one of the LSM extensions that implement MAC security improvements. It is included in number of Linux distributions by default. The SELinux development was originally started as Flux Advanced Security Kernel (FLASK) and then further developed by National Security Agency (NSA). The SELinux is not intended to stop buffer over-runs or malware applications getting into system, instead it can limit damage they cause. Android mobile operating system has also adopted the SELinux as a part of its security model. The SELinux was first enabled in a permissive mode for the Android, in which permission denials are logged, but not enforced. Full enforcement mode was enabled for Android 5.0 release [15].

SELinux policy component defines allowed access types and operations for system resources [14]. Policy decision making component is separated to own component called security server. Access vector cache (AVC) is a component which caches security server access decisions to minimize overhead. The policy enforcement functionality is implemented in kernel subsystems. Application processes can be confined to its own domain and allow only minimal set of privileges to perform its job. SELinux assigns a type security identifier in the security context of various system resources that have associated permissions to define what operations are allowed. This model is known as type enforcement.

SELinux requires the security context to be associated for resources that are used by security server to make access decision against policy [14]. In general, every subject and object in the system have an associated security context. For instance, a system process could be a subject and a file could be an object. The security context is defined as variable length string and it is also called security label. The security context consists of user, role and type identifiers. The user and the role identifier are used by the policy to define constrains based on identifier values. When the type identifier is used with a process, it identifies processes or domains that user can access. In case of type identifier is used for object, it defines what permission user has for it. In addition object gets automatically an object class identifier when it is instantiated. SELinux policy rule definition is shown in Figure 2.



```
allow rule      source_domain    target_type      : class        permission
---------------x-----------------------x--------------------------------------x-------------------
allow           unconfined_t     ext_gateway_t    : process      transition;
```

Figure 2. SELinux policy rule definition

Figure 2. illustrates how SELinux policy rule is defined to allow process running in unconfined_t domain to perform transition of target process to ext_gateway_t domain.

2.5. Hardware enforced security

TrustZone technology is a security extension used in ARM microprocessors. It provides a secure domain or secure world for security-critical software execution. For example mobile payment and virtual keypad for credentials input are potential software modules to be executed in secure world in order to separate it from normal execution environment. The secure world is able to access memory of the normal mode, but access is not possible the other way round. The secure world is implemented as logical ARM core, which is able to utilize memory management unit to further divide the secure world to sub-zones. Additionally, any of the system peripherals and interrupts can be allocated for secure world, thus general purpose OS running on the normal mode is not able to access those peripheral neither see interrupts. Secure world can be configured to run dedicated operating system or synchronous library in the simplest option. [4, 16]

Virtualization is a security mechanism that enables the abstraction of system resources. It is implemented by placing a relatively small control program called hypervisor or virtual machine monitor (VMM) between OS and the hardware. In full virtualization privileged and sensitive instructions are trapped, while user level instruction run at native speed. Typically modern computer architectures can execute in multiple operation modes with respective privilege levels. Traditionally in the ARM architecture privilege levels are called user and supervisor modes. Operating systems are designed to execute in the privileged mode. However, in the virtualized environment the VMM needs to run in the most privileged mode available in the system. The VMM runs in a privileged mode and hosts one or more guest OSs, which operate under illusion to have an exclusive access for system resources. Prior to hardware virtualization extensions, full virtualization was possible only using dynamic binary translation. Performance of the dynamic binary translation is not even close to execution speed of native system code. To address this issue, hardware assisted virtualization extensions have been added for mobile system on-chips. The ARM architecture hypervisor mode is added to support hardware assisted virtualization technology. To enhance system security, one guest OS can be dedicated for security critical functions, while another guest OS is dedicated for less critical applications. [4, 17]

3. SOFTWARE INSTRUMENTATION

Many approaches to gain understanding of application's internal behavior are available for the Linux environment [10]. Tools are useful for determining performance issues of the system and in addition provide a trace output to track application's behavior. Instrumentation is a technique to analyze and modify the behavior of the application by inserting additional code into it [18]. The instrumentation can be implemented on source code or binary level. A static instrumentation refers to a source code level instrumentation and generates persistent modifications for an executable. A dynamic instrumentation is a code injection for an executable at runtime, thus no permanent modification are made for a binary. This chapter discuss instrumentation methods and tools, which are relevant for understanding application monitor functionalities presented later in this thesis.

3.1. Instrumentation methods

The main difference between instrumentation methods are level of the information that can be produced, a performance impact generated for the instrumented application and a capability to directly instrument a binary object [10]. The application can be analyzed as a white box, when source code is available. Instrumentation code can be added for relevant points of the application and then recompiled to provide trace output. Different analysis methods has to be used when the application is available only as binary executable. This is called a black box executable analysis. In the black box analysis behavior of the application can be analyzed by monitoring an interaction with the operating system. Monitoring of the application's OS interface usage can take place for system calls, inter-process communication, signals and other interesting events. It exposes information how the application uses filesystem, network sockets and memory.

Thorough instrumentation of application behavior on the mobile device is not a trivial task, and might be even impossible because of limited operating system configuration. Typically an end-user configuration of the mobile device includes a subset of useful debugging and monitoring facilities due to performance and security reasons. However, most of the Linux-based mobile devices provide built-in system utilities in order to perform basic level instrumentation of applications. For detailed application and system instrumentation purpose, there are selection of advanced software instrumentation tools that can be installed to the system. Additionally mobile OS kernel can be even recompiled to enable features required by the instrumentation tools.

3.2. Tools

Various instrumentation tools are available depending on processor architecture and operating system. Most of the instrumentation tools are designed primarily for general-purpose architectures. Mobile devices are typically built on top of ARM architecture, which limits availability of instrumentation tools [19]. The Linux environment can be instrumented in several ways using utilities available in Linux mainline release. Instrumentation frameworks and tools are capable of instrumenting the Linux system on various levels such as user-space, file system, subsystems and system call interface.

Furthermore, presented instrumentation tools are suitable to be used in mobile devices built on top of the ARM architecture. Usually Linux instrumentation does not require patching of the kernel, but might involve modifications of kernel configuration depending on the target device. Instrumentation tools are presented in such order that underlying mechanisms providing certain kernel level tracing facilities are presented first, and then proceeding to instrumentation frameworks in subsequent chapters.

3.2.1. Kprobes

Kprobes is a dynamic instrumentation mechanism for Linux kernel, which allows information gathering without a need to compile or reboot the kernel. It was initially developed to be underlying mechanism for higher level tracing tools. The kprobes is organized in way that its functionalities can be easily extended by other tools. Kprobes package consists of user defined probe handlers, kprobes manager and architecture dependent exception handling mechanisms. [20, 21]

A kernel probe is a set of handlers placed for certain instruction address, which are executed when a breakpoint is hit [22]. The original instruction at the breakpoint address is executed when handler returns and context restore is performed. There are three probe types available: kprobes, kretprobes and jprobe. A kprobe can be inserted on any instruction address in the kernel. The kprobe is also called as pre-handler, because it is executed before the probed instruction. The kretprobe is executed when probed function returns, it is also known as post-handler. Figure 3 presents simplified kprobe handling steps.

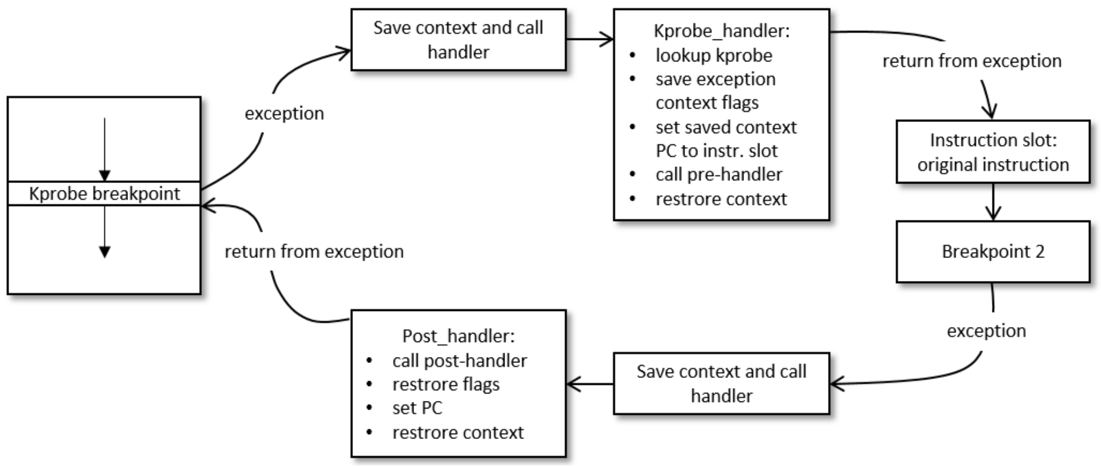


Figure 3. Simplified kprobe handling flow

The jprobe is inserted at the entry point of the kernel function, providing access for function parameter values. User defined probe instrumentation code is packed to a loadable kernel module that handles registration and unregistration of probe handlers at kernel module's entry and exit functions respectively. Actual implementation of the kprobes heavily depends on processor architecture. For instance, an exception-handling mechanism to support probe points varies across processor architectures.

3.2.2. Ftrace

Ftrace is a framework of several tracing utilities for debugging and analyzing kernel internals [23]. The ftrace is a suitable tool for tracing performance and latency problems within the kernel. A debug file system is used to hold ftrace control and output files. The ftrace is also used as a building block for other system monitoring tools. The framework consists of several tracers, and the most relevant ones are explained below.

Function tracer

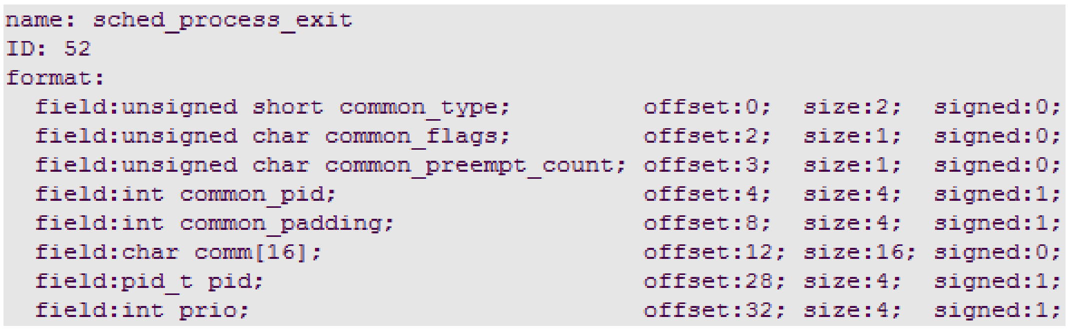
A function tracer is able to trace all kernel functions, its output contain function name and additional fields configured with trace options [23]. The trace options control data items and format of the ftrace output. Various data items can be included for output, such as a caller of the function and symbol related information. A function graph tracer does similar things as the function tracer, but it probes a function from its entry and exit points. On function entry point the graph tracer overwrites return address with a probe and original return address is stored in task structure. This enables function execution time measurement and provides reliable call stack for function call graphs.

Dynamic ftrace

A dynamic ftrace feature provides runtime control to enable tracing of selected kernel functions [23]. Runtime control is an essential feature to reduce overhead generated by tracing activity and it also reduces unnecessary output of the tracing session. Dynamic ftrace feature is a compile time option and it utilizes compiler's profiling option, which adds a call to a profiling function at the beginning of each kernel function. Responsibility of the profiling function is to check whether to call tracing function or just directly return. As the profiling function gets called a lot, it is carefully optimized to avoid performance issues. All profiling references are collected into single table in the linking stage of the kernel. On kernel boot up phase each of the table locations are replaced with a no-operation instruction and corresponding functions are made available for function filter list. When certain function is enabled for tracing, respective table location is modified back into trace call.

Event tracer

Kernel introduces compile-time defined static tracepoints, which are commonly referred as events in ftrace context. There are hundreds of events defined, which are organized based on kernel subsystems. Events can be enabled separately or in groups for an entire subsystem. All ftrace events contain common and event-specific data fields. As an example, process exit event is illustrated in Figure 4.



```
name: sched_process_exit
ID: 52
format:
  field:unsigned short common_type;         offset:0;  size:2;  signed:0;
  field:unsigned char common_flags;         offset:2;  size:1;  signed:0;
  field:unsigned char common_preempt_count; offset:3;  size:1;  signed:0;
  field:int common_pid;                     offset:4;  size:4;  signed:1;
  field:int common_padding;                 offset:8;  size:4;  signed:1;
  field:char comm[16];                      offset:12; size:16; signed:0;
  field:pid_t pid;                          offset:28; size:4;  signed:1;
  field:int prio;                           offset:32; size:4;  signed:1;
```

Figure 4. Process exit event

Each event has an associated filter expression, which controls whether a specific event is allowed to be added on trace output. The filter expression defines multiple numeric and string operators to test field values. The filter expression covers entire subsystem or only a single event. A trace event has an associated trigger capability, which is used to invoke commands. These commands can be invoked conditionally when the trace event is hit. Commands can be used to enable or disable other events, dump stack trace, take a snapshot of event at the time of trigger occurred or control entire tracing system state.

3.2.3. Systemtap

Systemtap is a dynamic instrumentation tool targeted for performance and functional problem solving of Linux kernel [24]. It provides an infrastructure to monitor running Linux kernel eliminating a time consuming process for recompile, install and reboot sequence. The systemtap is built on top of the Kprobes and kernel static tracepoints. The essential idea behind the Systemtap is to write handler scripts for events generated by the monitored system and tool itself. Scripts can be defined to react to several types of the kernel and the Systemtap internal events, such as a timer expiration, entering specific kernel function or a system call. A systemtap library provides wide variety of reusable scripts for system instrumentation purpose.

The systemtap operates by using the system C compiler to translate a handler script to a loadable kernel module. A command line utility is used to invoke a probing session. In turn kernel module gets loaded and it hooks the probes into the kernel. The handler function is executed when hooked event occurs. Hooks are unregistered and the kernel module is unloaded as a final step of probing session. [24]

Generated instrumentation code needs to be placed exactly right place in the Linux kernel, this requires system information packages to be available for the Systemtap. Linux kernel debug information is provided in development packages for desktop Linux distributions, which need to be exactly matching for installed kernel version. This is not the case for the most mobile devices running on top of the Linux kernel, hence requires Linux kernel compilation as a preparation of using the Systemtap.

3.2.4. LTTng

Linux Trace Toolkit next generation (LTTng) is an instrumentation framework for correlated tracing of the Linux kernel, user-space and libraries [25]. The LTTng consists of user-space trace libraries, kernel modules and tools for controlling tracing session. The kernel modules are needed when intention is to produce trace output from the kernel itself. User-space trace libraries are required when intention is to trace user-space applications. Linux kernel static tracepoints, kprobes and performance counter instrumentation facilities are supported through adaptation layer. Trace output format is a common trace format (CTF), which is a compact binary format containing packets of concatenated trace events. In order to analyze trace data, it is required to be converted to human readable text output.

Actual tracing session contains attributes and object for tracing. The tracing session defines domains to be traced and channels associated with them. Domain in LTTng context means kernel or user-space. A channel specifies parameters such as buffering mode, context information and list of events associated with the channel. Event can be separately enabled or disabled within the tracing session. The context information fields can be optionally added for generated events, which describe process information and performance counter values at the time of generating an event.

User-space application tracing for functions entry points take place with help of compiler's function instrumentation option. A compiler can be instructed to generate instrumentation calls for entry and exit of the functions, which are hooked by user-space tracing libraries of the LTTng. Additionally, static instrumentation can be performed using tracepoints, which can be placed at any point of the application. Tracepoints are defined either manually or generated using tracepoint tools. Custom argument expression of tracepoints makes it very flexible for tracing of user space applications. A tracepoint may have assigned an optional log level field that can be useful in tracing session control to limit amount of generated tracepoint events.

Several options exist for trace output viewing; tracing session can be configured to show events as they arrive, record events locally to files or even relay events to remote machine. Built-in feature of sending trace events over the network to remote machine is implemented for relay daemon component, which receives events on the remote system.

3.2.5. Ktap

Ktap is the most recent addition for the Linux dynamic instrumentation facilities, it has been designed toward needs of embedded users. The ktap tool is still on development phase and hence not available in the kernel mainline. The ktap differs from other mainstream instrumentation tools in way that it is a scriptable utility, which bases on a byte-code interpreter [26]. This design decision have an advantage that it omits a need for a compiler toolchain installation in the target system. The ktap operates using a special kernel module, which implements a virtual machine to interpret ktap scripts. Ktap scripting language is relatively simple and efficient, though it supports multiple features that are beneficial for dynamic tracing needs, such as control structures and built-in function library. More flexibility is introduced over the kernel built-in facilities due to tracing block definition can be used to collect additional data at the time of tracepoint hit. For instance, a backtrace of executing kernel task and value of global variables can be stored in associative array using built-in functions and data can be printed out when tracing session ends. Other instrumentation utilities presented in the previous chapters such as tracepoints, kprobes and function tracer are supported in the scripting language.

3.2.6. Audit

Linux Audit system is a monitoring framework to collect information about system in a form of Audit events. Rather than providing any additional security, Audit can be used to collect security relevant information about the system it is running. Properly configured Audit system makes possible to detect and analyze attacks againts the system. Audit rules are used to define which events are to be caught to a log file. This information can be used to determine violator of system security policy and details of operation. Audit consist of several components, each providing important functionality for overall framework. Audit components can be categorized for two main parts: user space utilities and kernel side system call processing module. [27, 28] Linux Audit framework components and their connections are presented in Figure 5. Solid line presents data flow and dashed control flow.

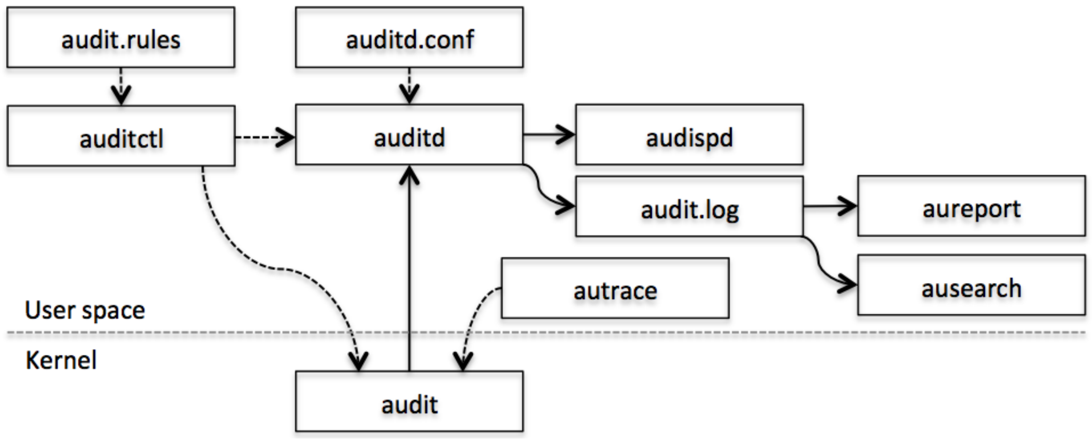


Figure 5. Components of the Linux Audit framework.

Audit kernel component

Audit kernel component responsibility is to filter system calls received from user space applications and deliver events to user space audit daemon based on activated rules. The Audit kernel component implements three rule lists: user list for requests originated in user space, task list for clone and fork system calls and exit list for system call exit. The rule lists are processed in presented order. A system call can trigger just one rule from the lists, i.e. the first matching rule generates the Audit event. Exclude list is processed as a final step to check whether filter is enabled for particular event type. Actual system call processing takes place between the user and task lists, which means that limited number of rule option fields have usable value at the time of the user list processing. [27]

Audit daemon and utilities

Audit daemon (auditd) is user space part of auditing, which allows inspection of system activities in great detail. Audit generates events based on rules, which are triggered by a system call or a file system access. Audit daemon is responsible for writing events received from kernel interface to audit log. Audit daemon configuration defines several options such as log file format, log file path, event rate limit and actions to be taken in case of disk full. Audit framework control happens through audit control utility (auditctl), it controls rule settings and changing of parameters for the kernel module. Audit search (ausearch) is a post-processing utility for filtering certain events from audit log file. Several filtering keys and event field values can be used as a filtering parameter. Audit dispatcher daemon (audispd) can be used to deliver events in real time to other application as well, hence enabling audit plugin implementation. Audit trace (autrace) is a process tracing utility designed for collecting audit events for a single application process. It does similar thing as strace, a well know system call tracing utility for Linux. Audit report utility (aureport) creates custom event reports out of audit event log. [27, 28]

Audit rules and events

Audit file system rules are used for watching access to files or entire directories. The file system rule defines a path and access types to trigger an event. Access type defines read, write and attribute change options. System call rules are watch points allowing more specific rule definition. The system call rule can define several field values to fine tune triggering of an event. Audit rule definition to catch a connect system call for D-Bus user bus socket is presented in Figure 6. The system call rule is added to the system call exit list, which allows Audit to properly inspect return value of the system call. In addition, it requires connect system calls to be successful, having architecture value set to ARM and system call associated path to be user D-Bus socket.

```
-a always,exit -F arch=armeb -S connect -F success=1 -F path=
/run/user/100000/dbus/user_bus_socket -F key=dbus_user
```

Figure 6. Audit rule definition for system call

A rule definition can contain optional user defined key parameter, which is automatically added for generated events. The key value is useful in log analysis to match event log entries for a specific rule. Figure 7 presents an Audit, which has been generated as a result for the Audit rule presented in Figure 6.

```
type=PATH msg=audit(01/24/15 18:34:57.643:127665) : item=0 name=(null) inode=13655 dev=00:
11 mode=socket,644 ouid=nemo ogid=nemo rdev=00:00
type=SOCKADDR msg=audit(01/24/15 18:34:57.643:127665) : saddr=local /run/user/100000/dbus/
user_bus_socket
type=SYSCALL msg=audit(01/24/15 18:34:57.643:127665) : arch=armeb syscall=connect per=PER_
LINUX_32BIT success=yes exit=0 a0=0x1b a1=0xbe9a5894 a2=0x27 a3=0xbe9a58b9 items=1 ppid=13
18 pid=24049 auid=unset uid=nemo gid=nemo euid=nemo suid=nemo fsuid=nemo egid=privileged s
gid=privileged fsgid=privileged tty=(none) ses=unset comm=booster-silica- exe=/usr/libexec
/mapplauncherd/booster-silica-qt5 key=dbus_user
```

Figure 7. Audit event

The Audit event in Figure 7 consists of three different types of records. A path type record describes accessed file system path. A socket address record contains exact path for local sockets. For Internet domain sockets record data would be an IP-address and port information. A system call record contains lot of information about system call in question. For instance, the system call record exposes parameters of the system call and its return value, process and parent process information, user and group IDs and name of the command used to invoke the process.

3.2.7. Summary

Multiple instrumentation tools are available on the Linux environment that can be used on mobile devices. However, restrictive software configuration on the mobile devices might avoid using the most suitable instrumentation tool. Typically two main properties of the instrumentation tools direct selection process that are level of details required and a need for customized event output. For instance, a scriptable utility such as the Systemtap is capable to hook on a system call and output content of memory reference arguments. Respectively an instrumentation utility providing only predefined static tracepoints outputs a memory reference, which obviously hides valuable information. Key features of presented instrumentation utilities are shown in Table 1.

Table 1. Feature comparison of instrumentation tools

Feature	Ftrace	Kprobes	Systemtap	Audit	LTTng	Ktap
User-space tracing			x	x	x	
Static tracepoints	x		x	x	x	x
Mainlined	x	x	x	x		
Output filtering	x		x	x	x	
Require symbols			x			
Require system compiler			x			
Byte code interpreter						x
Scriptable			x			x

The Table 1 contains instrumentation tools and frameworks that were presented in this chapter. Kprobes and ftrace tools are commonly used as building blocks for other instrumentation frameworks. The most of the presented tools have been available in the kernel mainline for long time and the latest ones are still provided as patches. The Ktap introduces scripting support while omitting need for a system compiler and symbols. This is kind a novelty feature for kernel instrumentation.

The audit instrumentation framework was selected to be an underlying tool for a monitoring utility implemented as part of this thesis. The audit is available in the kernel mainline and its user-space components were also available. Moreover, it does not require system compiler to be installed and provides good set of output filtering utilities. Audit omits a scripting support, but that level of system call monitoring or customized output was not needed.

4. INSTRUMENTATION MODULES

Monitoring extensions presented in this chapter are called application monitor and network traffic monitor. The application monitor is implemented as control and data processing extension on top of both the Audit instrumentation framework and proc filesystem. The function of the application monitor is to provide a report for utilized system resources. The network traffic monitor is a plugin extension to a Mitmproxy interception tool [31]. The plugin extension implements keyword-based content analysis for HTTP and HTTPS application layer data.

4.1. Application monitor

The application monitor controls the Linux Audit framework, which produces events according to activated rules. The Audit rules define system events to be collected. The Audit events are the main data source for the application monitor to generate a system resource usage report. The proc file system provides information about process names and identifiers. A startup notification of monitored application is resolved using system call interface. Application monitor's architecture is presented in Figure 8.

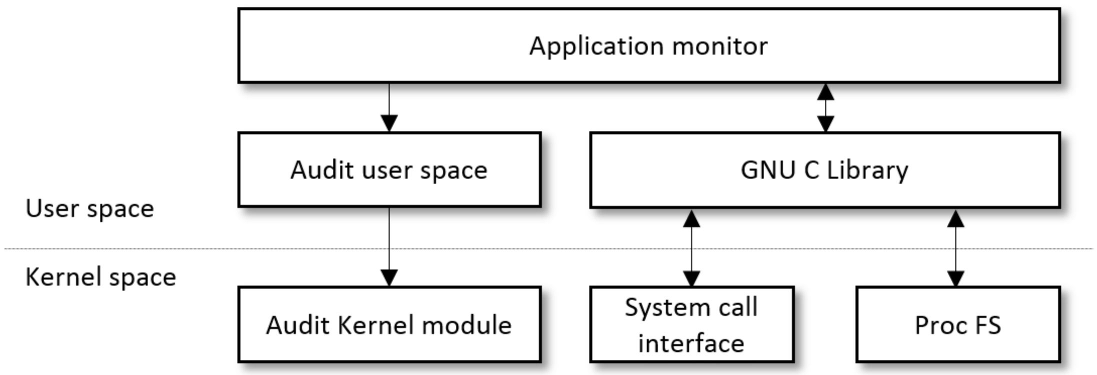


Figure 8. Application monitor architecture

The Audit framework collects data at runtime and stores events to a log file for post-processing purpose. The Audit control interface is used to setup system wide rules for file system paths and system calls. Process handling system call rules are used for collecting process identifiers (PID) and thread identifiers (TID). Both the PID and the TID values are used for filtering relevant information from the Audit log files. Some of the functionalities in the application monitor are tightly connected to the Sailfish operating system way to handle its activities. For instance, native application start-up condition detection by monitoring certain system calls.

Application monitor execution

The application monitor is a console application; it is started from the command line within the monitored device. The application monitor execution can be roughly divided in three phases. In a preparation phase the proc file system provides details of native processes that are to be assigned for started applications. Then the application monitor is configured for observing process state changes. In monitoring phase, one of the native applications has been started and the Audit is collecting events. In addition, Internet domain socket information are polled from the proc file system. In post-processing phase, the monitored application process has exited and filtering is applied for the Audit event log.

Native application startup detection

An application startup detection feature is implemented in order to synchronize enabling of Audit framework with the start of the monitored application. Unnecessary events received prior to monitoring session are discarded, making analysis of Audit events easier. Another advantage is a PID detection for monitored application. Otherwise PID of each application process should be resolved manually before starting the monitoring session. Application monitor execution flow is presented in Figure 9.

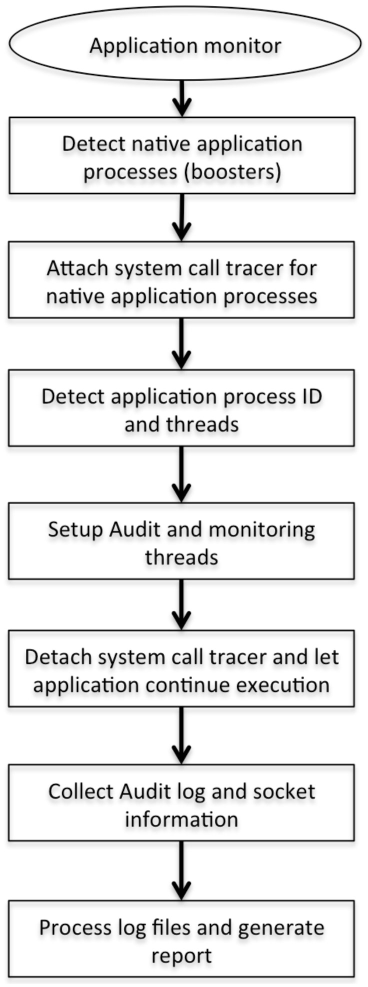
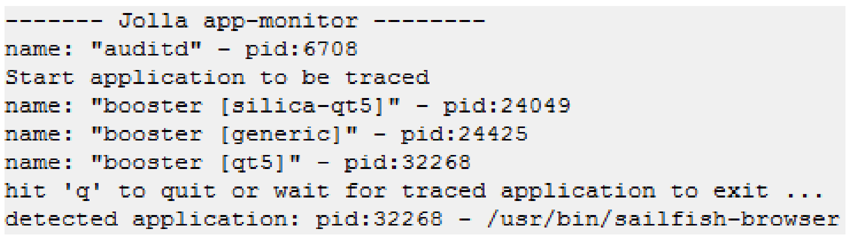

Application monitor
Detect native application processes (boosters)
Attach system call tracer for native application processes
Detect application process ID and threads
Setup Audit and monitoring threads
Detach system call tracer and let application continue execution
Collect Audit log and socket information
Process log files and generate report

Figure 9. Application monitor execution flow

The application monitor detects a startup condition of the native applications by attaching a tracer for each booster process. The native application startup detection method bases on a principle how applications are started in Sailfish OS. Sailfish application launcher creates a process called booster for each native application type in the system startup phase. A new booster process is created when the native application takes existing booster process in use. Booster processes are used by Sailfish application launcher to reduce application startup delay. The booster process exists for all native application types: qt5, silica-qt5 and generic. Figure 10 shows application monitor console output of native application detection.



```
------- Jolla app-monitor --------
name: "auditd" - pid:6708
Start application to be traced
name: "booster [silica-qt5]" - pid:24049
name: "booster [generic]" - pid:24425
name: "booster [qt5]" - pid:32268
hit 'q' to quit or wait for traced application to exit ...
detected application: pid:32268 - /usr/bin/sailfish-browser
```

Figure 10. Native application startup detection

The application monitor resolves booster process identifiers and waits for the user to launch an application. Finally application is detected and its real process name is resolved as can be seen in Figure 10.

Application monitor uses a process trace system call to attach for each booster process separately. Attach causes a stop signal for target process, which is in turn arranged to stop at every system call enter and exit entry points. The application monitor catches process state changes and looks from CPU registers whether the system call is a request to change name of the calling thread. The application monitor uses the process name set request as indication for application startup detection.

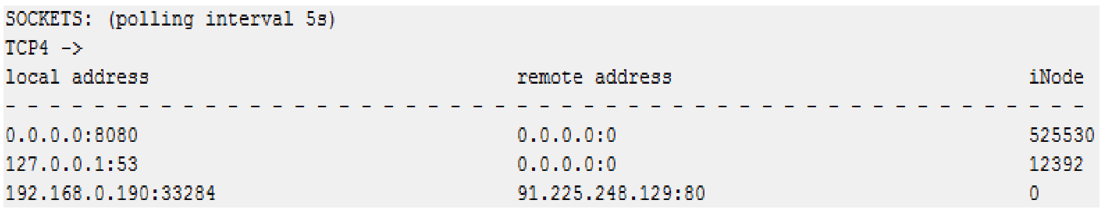
Application thread detection

Typically an application creates multiple threads during execution. Thread identifiers are collected and used in log file filtering phase to identify Audit events having a relation to the monitored application. In general, there are two types of application threads to be detected; booster process threads that already exist at the time of the application startup, and threads that are created during application execution.

Each booster process has threads that exist prior to actual application startup, those are resolved using the proc file system that exposes system process information. In the Linux-based environment, the proc file system contains a directory for each process. Thread identifiers are available in the process-specific directory. The booster PID is a process group leader, which is assigned for the started application. This information is used to resolve TID belonging to the monitored application process. At application runtime created threads are detected from the Audit log file by searching system call events that have been triggered by a clone system call.

Socket monitoring

Socket connections are monitored to get information of both local and Internet domain sockets activity during execution of monitored application. Socket monitoring is implemented with two separated approaches; by polling the proc file system and parsing socket information from the Audit log file. The socket information retrieved from the proc file system provides Internet domain sockets and contains also connections opened prior to the monitoring session. Collected sockets are system wide and hence might be created also by background applications. Monitoring of the Internet domain sockets are performed in a dedicated thread, which polls transmission control protocol (TCP) and user datagram protocol (UDP) specific files for new sockets. Each socket has an associated file system object, which is uniquely identified by index node (inode) entry that refer to a file system object node. The polling thread compares the proc file system sockets against the existing entries in the application monitor's socket list. In case of a new socket is found, it is appended to the socket list. Socket address and port information is stored for local and remote addresses. As an example, TCP sockets section of result report is presented in Figure 11.



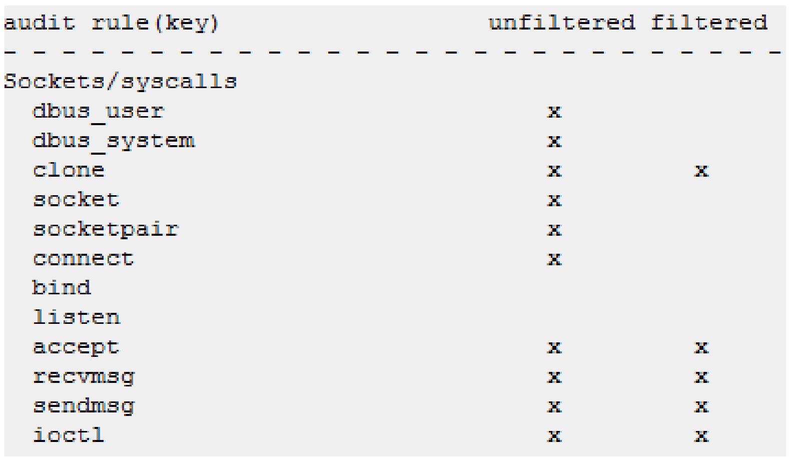
```
SOCKETS: (polling interval 5s)
TCP4 ->
local address                                   remote address                                  iNode
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
0.0.0.0:8080                                    0.0.0.0:0                                       525530
127.0.0.1:53                                    0.0.0.0:0                                       12392
192.168.0.190:33284                             91.225.248.129:80                               0
```

Figure 11. TCP-socket entries in the result report

In the Audit log filtering option, socket address records are filtered out from the Audit log and stored to a socket connection log file. A socket log entry contains address family, process information and host address for Internet domain sockets.

Results report

The application monitor generates the result report for thread identifiers, sockets and audit rule hits. Audit rule match section of the report contains filtered and unfiltered rule match results. Figure 12 shows rule match report for sockets and system calls.



```
audit rule(key)                  unfiltered filtered
- - - - - - - - - - - - - - - - - - - - - - - - - - -
Sockets/syscalls
  dbus_user                          x
  dbus_system                        x
  clone                              x         x
  socket                             x
  socketpair                         x
  connect                            x
  bind
  listen
  accept                             x         x
  recvmsg                            x         x
  sendmsg                            x         x
  ioctl                              x         x
```

Figure 12. Report for sockets and system calls

Unfiltered result column shows all rule hits for the entire Audit log file, i.e. it does not distinguish whether an audit event was generated by monitored application or a background process. Filtered result column shows only rule match results, which PID or parent PID field value match for the collected application thread identifiers. This is one way to identify the log entries generated by the monitored application. Individual Audit rule hit counts are not visible in the results. However, the count values can be resolved using Audit event log filtering tools. The Audit rules are read from the rule definition file and each identifier is added to the result report as well, this enables Audit rule changes without recompiling the application monitor itself. Complete report of application monitor can be found from appendix 1.

4.2. Network traffic monitor

The network traffic monitor plugin extensions can be used for analyzing HTTP and HTTPS network traffic intercepted by mitmproxy. The mitmproxy is a man-in-the-middle (MITM) proxy, capable of extracting encrypted application data from transport layer security (TLS) and secure socket layer (SSL) connections [30]. The application layer data is provided as a plaintext for the monitor plugin. The network traffic monitor runs entirely on target device and performs data analysis in real-time. User defined keywords are searched from multiple HTTP header fields and message body. Results are stored in several files that contain secure connections, certificate details and application layer data analysis results.

4.2.1. Mitmproxy

The mitmproxy supports multiple proxy operation modes. Suitable operation mode to be used depends on configurability of a client and use case. A transparent proxy mode is ideal option, when the client cannot be configured explicitly to use a HTTP proxy. In the transparent proxy mode client configuration can be omitted, because traffic is directed into a proxy at the network layer. Client is not aware of existence of the transparent proxy. In network monitor setup proxy server and routing mechanism are running on the same host, hence redirection is accomplished using iptables redirection mechanism. The iptables is a packet filtering and network address translation (NAT) capable tool in the Linux kernel. The transparent proxy needs to consult iptables in order to resolve original destination address.

For secure connection interception mitmproxy works as certification authority (CA), it contains a full CA implementation to generate interception certificates on the fly. Secure connection handshake process is illustrated in Figure 13.

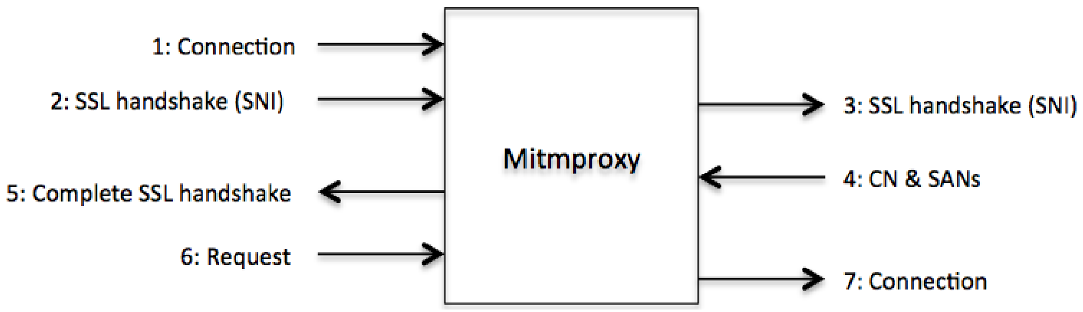


Figure 13. Mitmproxy SSL handshake message flow

1. Client establishes connection, which gets redirected to mitmproxy according to iptable rules. Mitmproxy is configured to listed local host port on the same host as client. Mitmproxy retrieves original destination utilizing routing mechanism.

2. Client initiates SSL connection as it thoughts to be communicating with remote server. Client sends server name request (SNI) to indicate host name it attempt to connect.

3. Mitmproxy establish secure connection to remote server using SNI requested by client.

4. Server responds with SSL certificate containing common name (CN) and subject alternative name (SAN). These values are used to generate interception certificate.

5. Mitmproxy generates interception certificate and continues handshake process.

6. Client starts to communicate over secure connection.

7. Mitmproxy passes client request to remote server using secure connection.

Client device needs to have proper certificate files installed for mitmproxy CA in order to avoid browser warning for SSL connections.

4.2.2. Monitor plugin

The mitmproxy provides an event driven Python scripting application programming interface (API) for plugin implementation. The scripting API can be used for modification of HTTP messages at runtime, and for implementing additional network traffic monitoring facilities. The scripting API provides hooks for several events such as request, response and connection information. Monitor plugin architecture is presented in Figure 14.

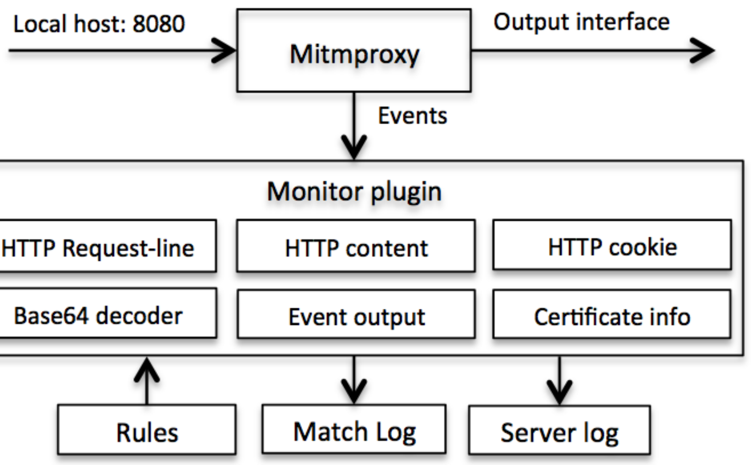


Figure 14. Mitmproxy monitor plugin architecture.

The network monitor plugin is automatically loaded at the startup phase of the mitmproxy. The monitor plugin initialization contains a processing of a rule file and opening files for both events log and server connections. Rules are defined as key-value pairs, in a way that the key is a symbolic name for actual plaintext string to be searched from intercepted traffic. The event log contains entries that describe rule matches found during monitoring session. The match event contains entire request line section of the HTTP request that match was detected for. In addition, event details contain a timestamp, client and server address, scheme and match type. The match type can be either plaintext or base64. An event containing a request line match is presented in Figure 15.

```
"reqLine": "POST /op/?id=10000&kielikoodi=fi",
"client": "192.168.0.190:39468",
"server": "157.124.22.10:443"
"cookie": "",
"reqContent": "REQUEST_LOGIN_ATTEMPTED=true&USERNAME=0765&PWD=&x=0&y=0",
"timestamp": "2015-04-19 14:55:53.161968",
"scheme": "https",
"match": [
    [
        {
            "value": "username",
            "type": "plaintext",
            "location": "content",
            "key": "username_1"
        }
    ]
]
```

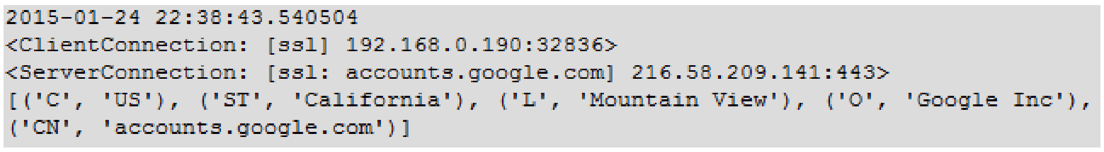
Figure 15. Monitor plugin event for a request line match.

A standard HTTP request consists of a request-line, a collection of headers and a body section. The request-line begins with a method, followed by uniform resource identifier (URI) and protocol version. The method defines an operation to be performed on the resource identified by URI. The header fields are used for passing additional information about the request to the server. The message body section is an optional part, when it is available it is used to carry entity-body associated with the request. The monitor plugin hooks to mitmproxy's client request and server response events. Actual data inspection for supported HTTP request's fields are performed on a client request handler. Details of upstream certificate are resolved on a server response handler.

The request-line inspection is performed to make sure that a query string does not contain user sensitive data. Content of entire request-line is converted to lowercase and then compared against plaintext keywords defined in the rule file. The request-line is also split in key-value pairs to perform both a base64 decoding and a plaintext comparison. The base64 is encoding scheme used to represent binary data in an ASCII format.

The HTTP request may contain a cookie header field. The cookie is a small piece of data stored in a user's web browser. Cookies are sent from the websites and stored in a user's browser. The browser sends the cookie back to the server when the webpage is revisited. The cookies provide a mechanism for a website to remember user's previous activity. The type of cookie can be either session or persistent. The session cookie is temporary and is removed when the web browser is closed. The persistent cookies remain in the web browser over sessions and are removed until they expire. The cookie consists of key-value pairs and can potentially store sensitive information. The network monitor plugin performs parsing for the cookies on client request handler. Content of the cookie is inspected the same way as the other request fields mentioned above.

The body section is used commonly with a put request to deliver data from the client to the server. The put request has an associated header specifying a type of the content that is ignored on the content parsing. The body section of the put request is inspected on the same way as the request-line. The purpose of the server response handler is to extract information of secure connection upstream certificate and store it to server connections log file. Format of extracted certificate information is shown in Figure 16.



```
2015-01-24 22:38:43.540504
<ClientConnection: [ssl] 192.168.0.190:32836>
<ServerConnection: [ssl: accounts.google.com] 216.58.209.141:443>
[('C', 'US'), ('ST', 'California'), ('L', 'Mountain View'), ('O', 'Google Inc'),
('CN', 'accounts.google.com')]
```

Figure 16. Extracted certificate information

Server connection log contains client and server connections addresses, and details of upstream certificate such as organization (‘O’) and common name (‘CN’).

5. TEST CASES

The goal of test cases is to produce information to perform behavior and security assessment for application under testing. Test cases try to provide an answer for questions: “what system interfaces and resources are utilized by the application and does the application expose user sensitive data using Internet connectivity?”

5.1. Target environment

Test cases are executed in Jolla smartphone, which runs on top of open source Sailfish OS. The Jolla was selected as a target environment due to its openness and that it represents minority on mobile market. Additionally such an open system allows changing of kernel level configurations and extensive usage of instrumentation tools. Sailfish is a Linux-based mobile OS combining Jolla’s user interface and middleware, Mer core and kernel hardware adaptation. In addition for native Qt5 applications Sailfish OS supports Android application through third-party libraries [29]. Figure 17 presents Sailfish operating system components.

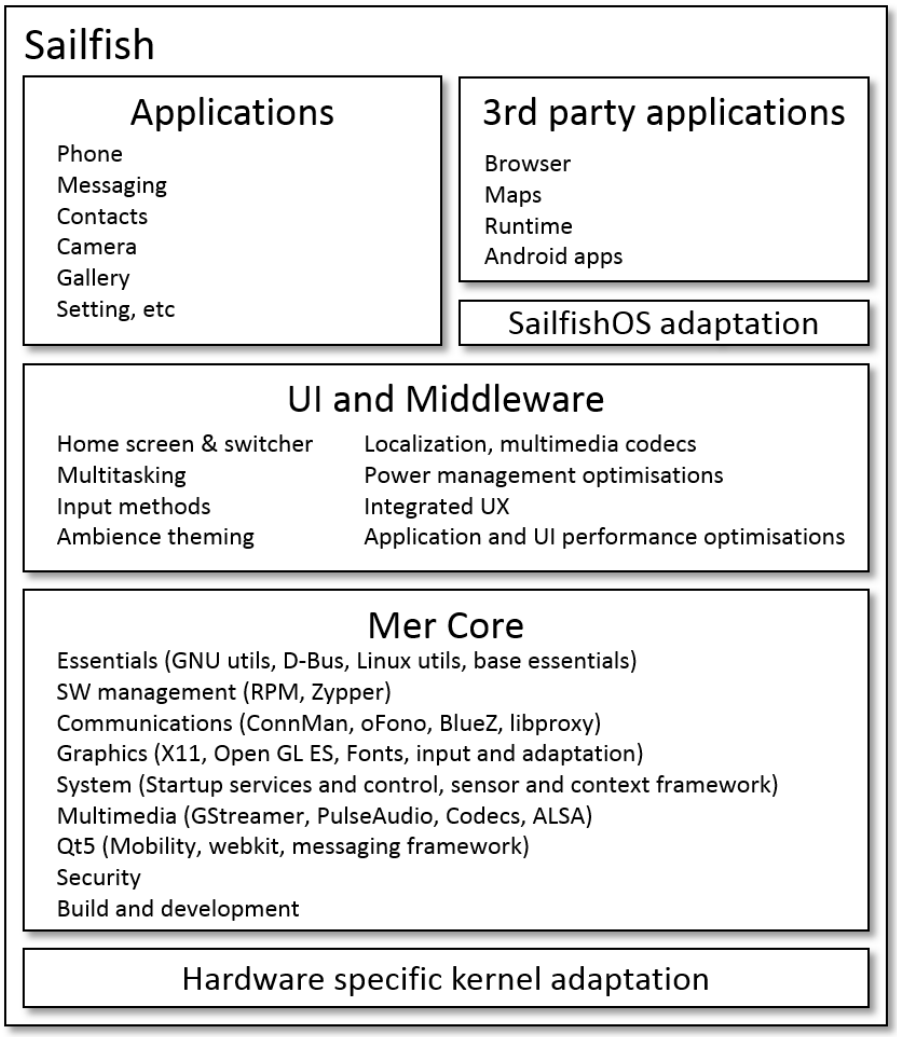

Sailfish
Applications
Phone
Messaging
Contacts
Camera
Gallery
Setting, etc
3rd party applications
Browser
Maps
Runtime
Android apps
SailfishOS adaptation
UI and Middleware
Home screen & switcher
Multitasking
Input methods
Ambience theming
Localization, multimedia codecs
Power management optimisations
Integrated UX
Application and UI performance optimisations
Mer Core
Essentials (GNU utils, D-Bus, Linux utils, base essentials)
SW management (RPM, Zypper)
Communications (ConnMan, oFono, BlueZ, libproxy)
Graphics (X11, Open GL ES, Fonts, input and adaptation)
System (Startup services and control, sensor and context framework)
Multimedia (GStreamer, PulseAudio, Codecs, ALSA)
Qt5 (Mobility, webkit, messaging framework)
Security
Build and development
Hardware specific kernel adaptation

Figure 17. Sailfish OS architecture.

Native applications in Jolla are built using Qt5 Sailfish silica components [29]. Native applications runs with privileged user permissions to access user sensitive data storage of Sailfish OS, such as contacts, calendar, messages, gallery etc. Native applications run as user identifier (UID) “nemo” and group identifier (GID) is set to “privileged”.

Android applications are supported by the Sailfish OS via third-party provided Alien-Dalvik virtual machine. Android applications do have different security policy compared to the native applications, because they do not have permission to access for privileged data storage and application permissions are requested in the installation phase. Installation procedure follows general Android application installation, where user needs to confirm application permission requests to proceed in installation. In the installation Android applications get assigned exactly same UID and GID values, which are set for certain application every time it starts. In addition all Android apps are assigned to process control group created by Alien-dalvik, which process group identifier is same as Alien-dalvik process identifier (PID).

Mer core is openly developed set of services and utilities between user interface and kernel. Its intention is to provide mobile-optimized core distribution for device manufacturers using Qt and HTML5 technologies. Mer core is maintained by Mer project and it is used in several different projects including Jolla. [30]

Sailfish kernel is Android fork of standard Linux kernel; it supports Android specific additions such as binder IPC mechanism, wakelocks, ouf of memory killer and Android shared memory. Jolla has implemented hardware adaptation for kernel to support HW configuration.

5.2. Application monitor testing

Application monitor is able to generate a report of Audit events for file system access and system calls used by monitored application. Event log output of the monitoring session is used for identifying system resource usage and outbound socket connections. In general, file system path rules are useful to track access for certain paths storing user sensitive data. The system call events provide detailed information about system activities and can be used to determine monitored application relationship to other system resources and services.

5.2.1. Environment preparation

Running the application monitor and Audit framework in the Sailfish OS based mobile device requires kernel level system configuration and user space component installation for target device. Jolla device used in testing was equipped with Tahkalampi software release, which is equal for version 1.0.8.21 of Sailfish OS. This chapter introduces how Jolla device was configured in order to perform tests using application monitor.

Linux kernel configuration requires several options to be enabled for a target device specific configuration file to take advantage of Audit framework. End-user version of Jolla kernel configuration does not enable all options required for Audit framework. In addition for existing Audit framework related kernel configurations, relay and Audit system call support were enabled. Audit system call configuration enables Linux kernel low-overhead system call auditing infrastructure, which enables Audit framework to trace system calls. Relay support enables efficient mechanism to transfer large amount of data from kernel to the user space. Audit framework uses relay interface to deliver events for audit user space dispatcher component.

Kernel configuration changes require compilation of the kernel and updating of corresponding flash image to the Jolla device. Kernel compilation produces a zImage, which is to be packed with a ramdisk to form a boot image. The ramdisk image contains initial root file system, which is loaded as part of kernel boot. The ramdisk changes were not needed, thus it was extracted out of the device's boot image and then repacked to with compiled zImage. A new boot image also involved adjustment of image file offsets, because the compiled zImage was a bit larger than the original one due to added configurations. In order to update the new boot image for Jolla device, it was required to perform bootloader unlock operation to allow operating system to boot from a custom boot image. Audit user space components were compiled from source and installed to the Jolla device.

5.2.2. Audit rule configuration

Audit rule configuration plays a significant role in application monitoring, because the accuracy of rules define usefulness of the event log. User sensitive data access attempts are one of the most interesting things to look for in application monitoring. Jolla device stores application and user sensitive data to the file system paths accessible by nemo and privileged user groups. A privileged storage contains user sensitive data such as contacts, images, calendar, notifications, social media and positioning information. The privileged path is accessible only for certain native applications belonging to privileged user group. Applications running with privileged permission can access all file system paths restricted for privileged group. These file system paths are added to the Audit rules to generate separated event for each watched file system path. File system watch rules for application data storage are presented in Table 2.

Table 2. Watch rules for application data storage

Path - /home/nemo/.local	Access	Key
/share/jolla-email	rwa	email
/share/commhistory	rwa	commhistory
/share/system/privileged/Contacts	rwa	privileged_contacts
/share/system/privileged/Images	rwa	privileged_gallery
/share/system/privileged/Calendar	rwa	privileged_calendar
/share/system/privileged/Notifications	rwa	privileged_notification
/share/system/privileged/Posts	rwa	privileged_posts
/share/system/privileged/Sync	rwa	privileged_sync
/share/system/privileged/qtposition-geoclue	rwa	privileged_position
/share/system/privileged	rwa	privileged_other
/share	rwa	nemo

A permission parameter for file system rules specifies, which type of access triggers an event. All monitoring session rules are defined to generate event for file read, write and attribute change. A key parameter is useful for report and post processing utilities to classify events. The last rule in the Table 2 collects all application data storage events, which are not matching for the preceding rules.

To track system resource usage, some of the system devices can be directly watched with file system rules. These devices are opened when the application starts to use them and closed at the time of the application termination. For example multimedia system devices, such as camera and audio can be watched with file system rules. Rules for camera and audio device access are presented in Table 3.

Table 3. Watch rules for multimedia devices

Path	Access	Key
/dev/v4l-subdev8	rw	back_camera
/dev/v4l-subdev9	rw	front_camera
/dev/snd	rw	capture_playback

Multimedia devices that handle user sensitive data are audio recording and camera. For instance, malware application might try to record phone call or use camera and later on upload recorded files for remote server. The most of hardware devices are already opened by system services at device startup phase and functionalities are provided for upper layers through specific services, hence watching that kind of device nodes directly do not generate Audit events.

To detect trivial system compromise attempts, file watch points are added for shadow, passwd and group files that hold encrypted users' passwords, system account information and user groups respectively. Audit watch rules for account monitoring are presented in Table 4.

Table 4. Watch rules for account monitoring

Path	Access	Key
/etc/shadow	wa	Shadow
/etc/passwd	wa	Passwd
/etc/group	wa	Group

In case of a malicious application manage to gain root access to a device or some other way manage to exploit system, it most probably attempts to deliver collected data to a remote server to take advantage of it. Data needs to be delivered by using one of the Internet connectivity options of the device. In order to detect a usage of connectivity interfaces, system calls are monitored for sockets and selected inter-process communication mechanisms. Audit system call monitoring rules are presented in Table 5.

Table 5. System call rules

System call	Options	Key
connect	path=user_bus_socket	dbus_user
connect	path=system_bus_socket	dbus_system
connect	success=1	connect
clone	success=1	clone
socket	success=1	socket_other
socket	success=1, a0=af_local	socket_local
socket	success=1, a0=af_inet	socket_inet
socket	success=1, a0=af_inet6	socket_inet6
socket	success=1, a0=af_netlink	socket_netlink
socketpair	success=1	socketpair
pipe	success=1	pipe
shmat, shmdt, shmget	success=1	shared_mem
recvmsg, recv, recvfrom	success=1	recvmsg
sendmsg, send, sendto	success=1	sendmsg
ioctl	success=1	ioctl

Data exchange activity between processes within the same host is monitored using IPC related system call watch points. Linux-based operating system provides various facilities for IPC, such as D-Bus, local sockets, message queues, shared memory and pipes. The D-Bus provides two separated buses for communication, which are watched using connect system call rule. D-Bus watch rules contain an option to specify a file system path for user and system bus sockets. The usage of message queues are monitored using recvmsg and sendmsg system calls and their variations. A watch point for socket system call logs an event for each successful socket creation. Event data contains socket details in its data fields, which are used for distinguishing local and Internet domain sockets. Intention of other socket related system call watch points are to provide additional information for socket connection in question. Monitoring of hardware device control is done using an ioctl system call, which exposes controlled device and request details.

5.2.3. Test sessions

Tests were performed for pre-installed applications and applications available from Jolla application store. In addition, applications were tested from openrepos, which is a distribution channel for applications that do not pass a validation process of official application store, or development of an application is still in progress. Test focus was set for native applications since some of the application monitor features rely on Sailfish OS. Various applications were selected from different categories. Goal was to select applications that would use as many system functionalities as possible.

Tests were run for one application at the time and avoiding use of other applications, which could cause unnecessary events for monitoring session output. Tested applications were operated normal way by trying to cover all functionalities available. With a full set of defined Audit system call rules, test session duration was kept relatively short in order to avoid exhausting output log. Tested applications and Audit rule match results are presented in table Table 6.

Table 6. Application monitor test results

	Devices			File system			Sockets and IPC							Test summary		
Application	back_camera	front_camera	sound_cap_playback	commhistory	email	privileged	socket_inet, inet6	socket_local	socket_netlink	dbus_user, system	recvmsg, sendmsg	shared_mem	pipe	Filtered threads	Audit events	Duration
jolla-calendar						x		x	x	x	x			11	4272	1:05
jolla-camera	x	x	x			x			x	x	x		x	110	11900	1:14
jolla-contacts				x		x		x	x	x	x	x	x	12	7459	1:41
jolla-email					x	x	x	x	x	x	x	x	x	27	9302	1:40
sailfish-maps						x	x	x	x	x	x		x	22	6033	1:20
sailfish-browser							x	x	x	x	x		x	41	5601	0:36
Harbor-meecast							x	x	x	x	x			14	6275	1:10
Harbor-friends							x	x	x	x	x			25	15365	2:03
Harbor-recorder			x					x		x	x			17	35010	0:55

Audit rules in the result table columns are named according to real rule identifiers, with an exception that multiple results are combined for a single result column for privileged files, inet sockets, dbus and message queue. The results do not contain all Audit rules being active during testing, because those were used to provide additional data for analysis of the other events. Applications are named exactly how those are identified in the device. Application names preceded with "Harbor" denotes that certain application is from sailfish store or openrepos. Browser and maps application are third-party provided applications. Results are taken from unfiltered Audit log file to avoid dropping out important events. The details section denotes number of application threads detected and Audit events collected during the test session. Duration is total time, which tracing was active.

5.2.4. Results analysis

Based on the results summary of the application monitor, there was no suspicious applications found. However, it required a deeper investigation for Audit event output to get better visibility of IPC mechanisms and Internet domain sockets usage. Sailfish-maps was selected to be an application for further analysis.

According to the result report, the sailfish-maps application did a file system access for the privileged storage and utilized several IPC mechanisms. Audit event filtering for file system access events exposed multiple events for the privileged storage. Accessed path contained a database for location services and nemo-user accessible geoclue service storage. This was analyzed to be a normal operation of the sailfish-maps application.

System call events were used to identify application process causing triggering of an event. The most of the system call argument fields were not usable on analysis due to memory reference type of argument. One exception being a connect system call event that exposes address to be connected in human readable format. Analysis for IPC and internet domain sockets usage needed collecting of individual rule match counts at the first place. Rule match counts are presented in Table 7.

Table 7. Sailfish-maps Audit events

Rule name	Number of hits
recvmsg	3886
sendmsg	1859
socket_inet	75
socket_local	30
socket_other	9
socket_netlink	8
dbus_user	8
dbus_system	7
pipe	5

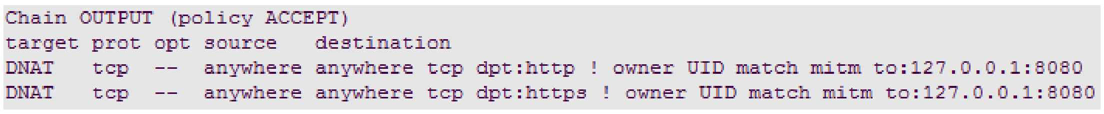
Relatively high number of message queue send and receive events were generated by various processes in the system, those include the sailfish-maps related processes. D-Bus session and system bus events were triggered by the sailfish-maps application and the geoclue providing D-Bus based location information services. Pipe events were also generated by the geoclue services.

Local socket events were produced by system daemon logging facilities, Android system logging and property service usage, wayland window manager and virtual keyboard connection. Internet domain sockets were connections for servers providing map data, those were opened multiple times for the same IP-address and different servers as well, which explains relatively high number of connections. Netlink socket events were raw type sockets, which were created by sailfish-maps application or one of its parent processes. In addition, Audit framework used a netlink socket to control its kernel component, those events were simply discarded. Sockets falling to the “socket_other” rule were created by the geoclue service.

5.3. Network monitor testing

In order to analyze HTTP and HTTPS network traffic entirely in the Jolla device, the mitmproxy with a network monitor plugin was installed to the device. All network traffic to be analyzed were routed through localhost port, which the mitmproxy was listening. This was accomplished by using iptables. In addition, a mitmproxy interception certificate was installed to the device to avoid the Internet browser warnings about untrusted secure connections.

Iptables output chain rules were defined to route standard HTTP and HTTPS ports to a loopback interface port 8080 that is the listening port of the mitmproxy. The output chain rules use destination network address translation (DNAT) target to rewrite address of matching IP packets. Mitmproxy was configured to run as dedicated user called mitm. This was done to avoid iptables to route packets originated from the mitmproxy again to the loopback interface. Without an owner option packets would be routed in an infinite loop to the loopback interface. Iptables rule setup is presented in Figure 18.



```
Chain OUTPUT (policy ACCEPT)
target prot opt source   destination
DNAT   tcp  --  anywhere anywhere tcp dpt:http ! owner UID match mitm to:127.0.0.1:8080
DNAT   tcp  --  anywhere anywhere tcp dpt:https ! owner UID match mitm to:127.0.0.1:8080
```

Figure 18. Iptable rules for mitmproxy

Several plaintext keywords were added to network monitor's rule input file to track phone identification numbers, personal information, location and account details. These keyword rules were automatically compared to application layer network traffic, which was routed through the mitmproxy. List of the keyword rules are presented in Table 8.

Table 8. Network monitor plaintext rules.

Key	Value
first_name	harri
last_name	luhtala
app1_pw	0765
uname_1	username
uname_2	harriluh
email	gmail
imei	359745050103180
phone_num	4877130
bt_address	5056a8002728
gps_lat	65.0
gps_long	25.

Several test cases were run to prove functionality of the network monitor plugin. Goal of simple test cases were to generate data in various forms, which should trigger defined rules. Test data was generated mainly with a native browser of the Jolla smartphone. As the network monitor is capable to inspect several HTTP request fields in a plaintext and a base64 format, all of those options were covered in the simple test cases. In addition, tests were run for real applications, which were observed to be using HTTP or HTTPS application layer protocol for communication.

The mitmproxy was started in a mitmdump mode with a transparent proxy option enabled. These startup parameters did not enable interactive shell for flow examination and neither retain HTTP flows in a memory for runtime manipulation. Instead, the mitmdump mode just prints client connections and monitoring plugin output to console.

Request line tests

Test data was generated with the Sailfish native browser by entering HTTP get requests. The request line contained matching string identifiers for the monitoring plugin keywords. The request line parameters were entered in plaintext and base64 encoded format. Additionally single request line content was also mixed with plaintext and base64 encoded values. The network monitor plugin writes runtime analysis results for the match log file. Test cases and results are presented in Table 9.

Table 9. Test result for HTTP request line

Request line	Key	type	scheme	section
GET http://www.kaleva.fi/harriluh/	uname_2	plaintext	http	request line
GET http://www.kaleva.fi/?data= MzU5NzQ1MDUwMTAzMTgw	imei	base64	http	request line
GET http://www.google.fi/ ?q=4877130&x=MDc2NQ==	phone_num, app1_pw	plaintext, base64	http	request line

The request line column presents HTTP requests used in the test cases. Domain names used in the requests are just for giving a clue how parameters might be delivered within the request line. Subsequent columns are test result provided by monitoring plugin. Tests cases did prove that the network monitor plugin is capable of identifying plaintext and base64 formatted strings within the HTTP request line.

Tests for encrypted content

Monitor plugin was tested with PUT requests that involve sending of an encrypted content section. This was accomplished by using the native browser and entering input data for secure websites. Identifier strings were entered in website's forms and then request was submitted. All network traffic for standard HTTPS port was routed through the mitmproxy as explained in environment preparation section. Test results for encrypted content are presented in Table 10.

Table 10. Test results for encrypted content

Request type, input form data	key	type	scheme	section
PUT, username: harriluh password: 0504877130	uname_2, phone_num	plaintext, plaintext	https	content
PUT, username: 359745050103180 password: MDUwNDg3NzEzMA==	imei, phone_num	plaintext, base64	https	content

Input data was entered on username and password fields as shown in above table. Network monitor plugin was able to successfully detect plaintext and base64 decoded content from the submitted requests.

Tests for cookies

Web browsers commonly use cookies for storing state information, which is delivered back to server for subsequent connections for the same web site. Cookie content inspection was tested using cookie testing web page that allowed user to define content of the cookie. Network monitor match log entry for cookie testing is presented in Figure 19. Match log entry for cookies.

```
"reqLine": "GET /tools/cookietester/",
"client": "192.168.0.190:33379",
"cookie": "{' Name': ('harri', {}), 'MDcwNTgw': ('Value', {})}",
"reqContent": "",
"timestamp": "2015-03-15 23:15:16.040982",
"scheme": "http",
"match":
        {
            "value": "harri",
            "type": "plaintext",
            "location": "cookie",
            "key": "first_name"
        },
        {
            "value": "070580",
            "type": "base64",
            "location": "cookie",
            "key": "birthdate",
            "encoded": "MDcwNTgw"
        }
```

Figure 19. Match log entry for cookies

The match log entry contains two rule matches in a single get request. Plaintext and base64 encoded fields were detected for cookie section of the request.

Real applications

Real application testing was performed in a way that the phone would be used in a daily usage. This involves using various applications and installing new ones from application stores, using browser, messaging services and social media. The network monitor was enabled on background to expose suspicious activity for outbound HTTP and HTTPS ports. Identifier keywords were exactly same as in the simple test cases, which will obviously generate false positives for match log in normal use. This means that the match log file needs to be investigated manually after test session to identify false positives. Network monitor tests result for real application are presented in Table 11.

Table 11. Rule match results for applications.

Application – version	Key	type	Scheme
Vopium instant messenger – 3.4	phone_num, imei	plaintext, base64	https
Sailfish maps – 1.0.3	gps_lat, gps_long	plaintext	http
Aptoide appstore – 5.2.0.2	gps_lat, gps_long, imei	plaintext	http

Several instant messenger application were installed to the device as personal information related security issues have been reported for this application category in the past. Vopium instant messenger application exposes user’s phone number and International Mobile Identity (IMEI) code for remote server. This information was delivered using secure connection and the IMEI code was base64 encoded within the request’s content section. Phone number is used as a user name in the Vopium application. One reason to deliver IMEI could be intention to prevent multiple logins with the same user name. The network monitor was not able to detect other suspicious communication events based on defined keywords for instant messenger applications. Some of the tested applications did stop working due to failing MITM proxy certificate interception. The interception of the MITM proxy might be failing, because a certificate pinning prevents interception process. The certificate pinning means that an application contains hard-coded information of the certificates to be used by the server, thus the application trusts to certificates signed by a small set of certificate authorities. In addition, many of the instant messengers were using different application layer protocol such as extensible messaging and presence protocol (XMPP).

Device location information was exposed in a form of global positioning system (GPS) coordinates by two applications. Sailfish maps was delivering coordinates using HTTP get request. This was harmless activity because location information was used for requesting map data for certain location. Apptoide application store delivered a location information and IMEI code for a remote server. This activity happened when application installation was started in the Apptoide store. Additionally unsecure HTTP connection was used to transfer sensitive information to the remote server.

6. DISCUSSION

The main challenges in thorough mobile application instrumentation are to identify limitations caused by target system and to select a suitable instrumentation tool. It might be cumbersome to understand what are the kernel level configurations required by instrumentation tools. In the most cases, mobile device's limited and performance optimized configuration cause obstacles for instrumentation frameworks. This obviously leads relatively big efforts due to need for cross-compilation environment setup and kernel rebuild with required options. Additionally device manufacturer might have implemented security features on the device in order to prevent unauthorized updates of the kernel. The mentioned limitations give an overview what kind of problems might be faced in mobile device environment, before actual instrumentation is even started.

Instrumentation frameworks produce system wide events, which of course can be limited by defining a monitoring rule associated with an application process identifier. However, the application process identifier does not exist before the application is started, and typically multiple threads are created during execution. To limit amount of instrumentation events, understanding of the monitored system is required in order to properly configure selected instrumentation framework. In general, too widely defined instrumentation rules might overload resource-constrained system when CPU intensive application is monitored. For example, this issue happens when system call interface is monitored and there are no restrictive parameters defined for the event generation.

The Audit framework was selected as an underlying instrumentation tool for application monitor extension. The Audit turned out to be a capable tool to monitor privileged storage and system calls with flexible configuration options, thus it was a good choice for detecting malicious applications that would act against their expected behavior. Additionally, Audit was configured to observe an access to the privileged storage, system devices and network resources. However the Audit does not provide a way to monitor explicitly a single application. It was soon realized that events produced with the Audit would need some sort of processing to get understanding of application's resource usage. The application process and thread identifiers are the information in Audit events to distinguish them to monitored application. However, these identifiers could not be resolved after closing the monitored application. These issues led to a decision to develop a native application process detection mechanism that is run prior to the Audit framework. The application startup and dispose detection mechanism automatically controls the Audit framework to limit event generation for the time period application being active. In addition, filtering of Audit events were implemented based on collected process and thread identifiers to distinguish monitored application events.

The goal was to utilize existing instrumentation tools and extend functionalities to provide an automatic way to observe suspicious applications. The developed extension and Audit framework were successfully used for application resource usage monitoring. For instance, the application monitor was able to identify privileged storage usage and an access to sensitive device resources such as audio recording or camera. Monitored application's usage of Internet domain sockets were also identified. In general, the application monitor was able to produce a report that can be used for application resource usage assessment. A problematic situation happens when an application does an access to a resource through inter-process communication. In this case, application access is tracked for IPC mechanism usage, not for the actual resource. The resource access is marked to a background process although it is initiated by monitored application.

As the application monitor provides an overview of used system resources without providing any analysis results for data content, network monitor was developed to search keywords from the outbound HTTP and HTTPS traffic. The keyword definitions passed to the monitor as input parameters resulted several findings for encrypted and encoded content as well. Problems were generated by several applications, because protection mechanism against SSL connection interception was obviously implemented using certificate pinning. The network monitor functionalities could be easily further developed by adding more clever content analysis functionality within limits of target system performance. The content analysis functionalities developed in scope of this thesis were plaintext and base64. The network monitor or equivalent setup could be considered as potential background process on the mobile device. In a real usage, network traffic routing through the transparent proxy should be dedicated for the subset of applications at the time, and number of keywords should be limited. Additionally keywords contain exactly the same sensitive information that would interest an attacker. Thus, secure execution and storage environment should be taken in use for that sort of monitoring application.

Related research papers have been published that discuss monitoring of applications and tracking of sensitive data on mobile devices. The prior research uses mainly dynamic taint analysis approach to malware detection and analysis. A good example of this is TaintDroid for Android [32]. Although the TaintDroid monitoring approach differs from the solution presented in this thesis, it has somewhat the same goals. The TaintDroid tracks sensitive information flows in the system. The tracking takes place by automatically labeling sources of sensitive data and observing how labeled data propagates in the system. The taint analysis provides more accurate results for the sensitive information tracking compared to the approach presented in this thesis. The main advantage of TaintDroid is that information flows between applications are identified and modified sensitive information can be detected.

7. CONCLUSION

This thesis presented a resource usage monitor and a keyword-based network traffic content monitoring for mobile device environment. General purpose system instrumentation tools were extended by developing additional features that enable behavior assessment of monitored application. Test cases were performed to prove functionality of developed extensions and to monitor native applications of Jolla smartphone. The test cases focused to discover malicious behavior and sensitive information leakage.

The resource usage monitor for native applications did not produce finding related to unexpected system resource usage, however the selected method was proven to be suitable for monitoring applications in a resource-constrained mobile device. The network traffic content monitoring produced multiple matches that can be interpreted as personal information. Although network traffic content analysis were performed only for limited number of application layer protocols and subset of their content and encoding schemes, it produced promising results. To further develop network content monitoring, it could be improved by supporting several encoding schemes and multiple application layer protocols. Another important consideration would be a mechanism to focus the network content monitoring on a single application, which enables activation of extended content analysis in a resource-constrained system.

8. REFERENCES

[1] Mobile planet smartphone research. (accessed 24.1.2015), Google inc. URL: http://think.withgoogle.com/mobileplanet/en/.

[2] Mobile threat report Q1 2014. (accessed 24.1.2015), F-secure. URL: https://www.fsecure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014_print.pdf.

[3] Asokan N., Davi L., Dmitrienko, A. (2013) Synthesis lectures on information security, privacy, and trust: mobile platform security. Morgan & Claypool publishers, CA, USA, 110 p.

[4] Kleidermacher D., Kleidermacher M. (2012) Embedded systems security: practical methods for safe and secure software and systems development. Elsevier, Amsterdam, 1st ed., 416 p.

[5] Becher M. (2011) Mobile security catching up? Revealing the nuts and bolts of the security of mobile devices. In: IEEE symposium on security and privacy, May 22 – 25, Berkeley, CA, United States.

[6] Android system permissions. (accessed 12.4.2015). URL: http://developer.android.com/guide/topics/security/permissions.html.

[7] Whitman M. & Mattord H. (2011) Principles of information security. Course Technology Press, Boston, MA, United States, 4th ed., 617 p.

[8] Vidas T., Vopitka D. & Christin N. (2011) All your droid are belong to us: A survey of current android attacks. In: 20th USENIX Security Symposium, August 8 – 12, San Francisco, CA, United States.

[9] OWASP mobile security project. (accessed 12.4.2015). URL: https://www.owasp.org/index.php/OWASP_Mobile_Security_Project.

[10] Desnoyers M. & Dagenais M. (2006) Tracing for hardware, driver and binary reverse engineering in Linux. In: Recon conference, June 16 – 18, Montreal, Canada.

[11] Kerrisk M. (2010) The Linux programming interface, a Linux and UNIX system programming handbook. No Starch Press, San Francisco, CA, United States, 1st ed., 1506 p.

[12] Wright C., Cowan C., Smalley S., Morris J. & Kroah-Hartman G. (2002) Linux Security Module Framework. In: Linux Symposium, June 26 – 29, Ottawa, Canada, Vol. 1, s. 604 – 617.

[13] Another LSM stacking approach. (accessed 12.4.2015). URL: http://lwn.net/Articles/518345/.

[14] Haines R. (2014), The SELinux notebook. URL: http://freecomputerbooks.com/books/The_SELinux_Notebook-4th_Edition.pdf.

[15] Security-enhanced Linux in Android. (accessed 12.4.2015). URL: https://source.android.com/devices/tech/security/selinux/index.html.

[16] ARM ltd. (2009) Building a secure system using TrustZone technology. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[17] Mijat R. & Nightingale A. (2011) ARM white paper: virtualization is coming to a platform near you. URL: http://www.arm.com/files/pdf/system-mmu-whitepaper-v8.0.pdf.

[18] Laurenzano M., Tikir M., Carrington L. & Snavely A. (2010) PEBIL: Efficient static binary instrumentation for Linux. In: IEEE international symposium, March 28 – 30, White Plain, NY, United States.

[19] Hazelwood K. & Klauser A. (2006) A dynamic binary instrumentation engine for the ARM architecture. CASES '06: proceedings of the 2006 international conference on compilers, architecture and synthesis for embedded systems. ACM, New York, NY, United States, pp. 261-270.

[20] Kernel probes (Kprobes), (accessed 12.4.2015). URL: https://www.kernel.org/doc/Documentation/kprobes.txt.

[21] Goswami, S., (accessed 3.11.2014), An introduction to KProbes. URL: http://lwn.net/Articles/132196/.

[22] Barnes Q. (2007) Kernel Probes for ARM. In: the embedded Linux conference, April 17 – 19, San Jose, California, United States.

[23] Ftrace - function tracer. (accessed 3.11.2014). URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[24] Eigler F. (2014) Systemtap tutorial. URL: https://sourceware.org/systemtap/tutorial.pdf.

[25] The LTTng Documentation. (accessed 12.4.2015). URL: http://lttng.org/docs/.

[26] The ktap Tutorial. (accessed 12.4.2015). URL: http://www.ktap.org/doc/tutorial.html.

[27] System auditing. (accessed 3.11.2014). URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/chap-system_auditing.html.

[28] Linux audit quick start. (accessed 12.4.2015). URL: https://www.suse.com/documentation/sles11/singlehtml/audit_quickstart/audit_quickstart.html.

[29] All about us and the OS. (accessed 12.4.2015). URL: https://sailfishos.org/about/.

[30] Mer. (accessed 20.11.2014). URL: http://merproject.org.

[31] Mitmproxy. (accessed 12.4.2015). URL: https://mitmproxy.org/doc/mitmproxy.html.

[32] Enck W. (2010) TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: USENIX symposium on operating systems design and implementation, October 4 – 6, Vancouver, Canada.

Appendix 1. Example output of the application monitor.

```
------- Jolla app-monitor --------
name: "auditd" - pid:10468
name: "booster [silica-qt5]" - pid:10657
name: "booster [generic]" - pid:1447
name: "booster [qt5]" - pid:1397

detected application: pid:10657 - /usr/bin/jolla-camera

----- Report -----
date:2015-04-06, start:23:27:18, stop:23:27:51

Thread identifiers:
10657 10692 10693 10698 10699 10700 10701 10935 10938 10939 10940 10942 10943 10944 10950
10951 10952 10953 10954 10955 10956 10957 10958 10959 10960 10961 10963 10965 10966 10967

SOCKETS: (polling interval 5s)
TCP4 ->
local address                                 remote address                      iNode
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
127.0.0.1:53                                  0.0.0.0:0                           12542

UDP4 ->
local address                                 remote address                      iNode
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
127.0.0.1:53                                  0.0.0.0:0                           12534
0.0.0.0:67                                    0.0.0.0:0                           415934

Audit rule(key)                  unfiltered filtered
- - - - - - - - - - - - - - - - - - - - - - - - - - -
Devices
  back_camera                        x
  front_camera                       x
  sound_cap_playback                 x         x
Files (/home/nemo/.local/share/...)
  email
  commhistory
  privileged_contacts
  privileged_gallery
  privileged_calendar
  privileged_notification
  privileged_posts
  privileged_sync
  privileged_other
  nemo                               x         x
Files
  group
  passwd
  shadow
IPC/syscalls
  dbus_user                          x
  dbus_system                        x         x
  socket_local                       x         x
  socket_inet
  socket_inet6
  socket_netlink                     x
  socket_unspec
  socket_other
  socketpair                         x         x
  connect                            x         x
  socket_details                     x
  shared_mem
  recvmsg                            x         x
  sendmsg                            x         x
  ioctl                              x         x
  pipe                               x         x
  clone                              x         x
```