Innopolis University

System and Network Engineering

Secure Telemetry in Sailfish Mobile OS RUS

Author:
Nikita Mokhnatkin
Niyaz Kashapov
Sergey Grebennikov

Supervisor:
Dr. Rasheed Hussain
Teacher Assistant:
Anatoly Tykushin

January 25, 2018

System and Network
Engineering

Abstract—**The collection of debugging data (telemetry) are used to locate logic error locations or performance problems and to show them to the developer. This data can display both the internal state of the application and the user's behavior. Telemetry data in mobile devices can be associated with personal information or business data of users. The standard application development tools for Sailfish Mobile OS RUS do not provide instruments for secure collection of telemetry. Therefore, it is required to develop an approach that would guarantee the security of personal data when working with them and when they are transferred to developers. This approach should be safe and not disrupt the current state of security.**

I. Introduction

Telemetry is designed to make life easier for the developer when debugging applications. Based on the debugging data, the developer will be able to compose a complete picture of what is happening in the application. In order not to miss important information during the execution of the application, it is crucial to avoid the collection of debugging information too little. On the other hand, it is also crucial to avoid the collection too much, that can affect the performance and cost of telemetry. In addition, in a client-centric trusted mobile environment, it is very important that the received data should not be associated with personal information and business data of users. Therefore, it is required to develop an approach that would guarantee a secure environment when collecting debug information.

Sailfish Mobile OS RUS is a POSIX-compatible operating system based on the kernel and Linux libraries which is being developed since 2012 by the Finish company Jolla. In 2016, the Russian company Open Mobile Platform began to cooperate with Jolla to create an internal operating system Sailfish Mobile OS RUS, which would be an advanced secure operating system suitable for corporate use. Sailfish Mobile OS RUS carefully applies to personal information and business data by providing security mechanisms that are built into the kernel of the operating system.

At the moment, standard application development tools for Sailfish Mobile OS RUS do not have the means for collecting telemetry. Therefore, it is required to investigate technical ways of introducing such tools into the source code of applications and libraries. Their implementation should help application developers when debugging and processing error messages.

A. Description of the problem

The main problem considered in the research work is the lack of a methodology for collecting debug application data for Sailfish Mobile OS RUS, which guarantees the safety of personal and business data of device users. It is required to investigate the technical ways of implementing such a methodology. Its implementation should help application developers when debugging the application, and also ensure the security of the transmitted data.

B. Research Question

The research project has the following questions:

- What kinds of telemetry can be collected in mobile applications Sailfish Mobile OS RUS, and what known methods exist to collect this information?
- Which information can be collected for the developers without touching the personal and business data and which risks are considered in the delivering?
- Which architecture of implementing telemetry acquisition tools in applications and libraries Sailfish Mobile OS RUS will ensure the security of personal data?

II. Theory

Telemetry collects debugging information. It is widely used in the different tasks of the software development, such that, finding of anomalies, error debugging, performance diagnostic, system behavior analysis, statistical analysis of the user behavior and etc. Telemetry must help for a developer to get the full picture of the application usage. Also, telemetry helps to analyze application behavior on the different devices (usually, applications are tested on the one-two different devices, not more). From another hand, telemetry is used to analyze which features and abilities of the application are most usable and invokable. Detailed information can get information about the most slow-speed sections of code.

Software companies are trying to improve the performance and reliability of their products by introducing telemetry into system libraries. For example, the implementation of the telemetry was given by Microsoft in the Windows SDK, that allows developing applications for Windows 10, Windows Phone 10 and ASP.Net. Telemetry delivers information about application behavior and system errors. Following common information are collected by applications that used the Microsoft SDK: device type, name, and model, network type, country, application name, events' properties. The OnePlus company implemented telemetry into the operating system to collect whole information about the device. All collected information was used for optimization of device working. All devices collected following parameters: phone serial number, IMEI, identifiers, battery status, information about opening/closing of the applications, time of blocking and unblocking, the screen on times and etc. But, this information can be used for user identification, and this is a kind of the personal data.

Some companies offer their solutions to collect the debug application data. The Bugfender company developed a remote logging framework for cross-platform applications (Android and iOS). This framework allows to collect logging information (errors, some technical information, developers' tags and etc) and deliver this information to the cloud storages. Libraries for Android and iOS uses system logger systems to collect and aggregate information - `logcat` and `iOSLogInfo` respectively. Logs can get from the system - information about events, system interrupts, exceptions, and can be from the developer - custom events, variables' values and etc. This kind of telemetry is wider, but not secure for the users.

In addition to proprietary solutions, there are also open cross-platform libraries for collecting debugging information. An example of such a library is a cross-platform telemetry library p7, that collects information about the application and send to the special server named a "Baical". The p7 library can be used from the programs are written in C, C++, C#, Python. The library collects information about threads, memory and CPU usage, internal events and exceptions. But, all telemetry can be analyzed only in the proprietary server, that works only on Windows hosts.

All of these examples are good enough and allow to collect and deliver debugging information from the user to the developer. But none of the above methods can be implemented or launched on Sailfish Mobile OS RUS. And it is necessary to write a unique solution to collect and send telemetry information.

III. The actual conducted research

A. *Collected Information*

One of the most important and actual issues of the telemetry is a keeping confidentiality and safety of the sent information.

First, it is required to determine what "personal information" and "business data" are. In accordance with the federal law of the Russian Federation [1], *personal information* is any information related directly or indirectly to a specified or determined individual (the subject of personal data), including:

- Name
- Postal address (including billing and shipping addresses, and state/province and zip/postal code)
- Email address
- Telephone number
- Usernames and passwords
- Credit or debit card number you provide to our third-party service provider(s)
- *etc*

Also, in accordance with the Federal Law [1], the processing of personal data is carried out with the consent of the subject of personal data to the processing of his personal data.

Business data is a collection of data necessary for making commercial decisions about the market situation (data on prices, suppliers, competitors, the dynamics of customer demand, *etc*) [3]. Business data can include the following information:

- external environment of the enterprise (legislative restrictions, demographic trends, *etc*)
- internal state of the firm, its strengths and weaknesses
- market situation (market capacity, dynamics and supply, distribution of shares between firms, *etc*)
- consumers (the composition of consumers, their income level, housing, buying motives, the determinants of behavior, tastes, habits, reaction to advertising, *etc*)
- competitors (the activities of the main competitors, their strengths and weaknesses, their sales techniques, *etc*)
- suppliers (the composition of the supplier enterprises, their financial and economic status, reputation, terms of delivery)
- goods (the assortment of goods entering the market, the assortment of competitors' products, the appearance of novelties, the evaluation of the competitiveness of goods)
- prices (the level of current wholesale and retail prices, their dynamics, prices of suppliers, competitors)
- distribution systems (common distribution channels, their effectiveness)
- advertising (the effectiveness of various means of advertising, information about consumer awareness, *etc*)

These data can be classified as commercial secret and protected by the federal law of the Russian Federation [2].

Most companies do not provide an opportunity to use a trusted mobile platform and do not care about the personal data and business data of users. For example, the OnePlus company had big problems with telemetry because collected information allowed to identify a device user. This is the violation of confidentiality. Also, telemetry on the Android devices are given by third-party libraries are dangerous too, because they are sending all information, include the entered logins and password of the user. From the library perspective, the system is secure, because all information is sent via HTTPS with encryption. But, from the developer and intruder perspective, telemetry systems allow stealing important information, like passwords, card numbers, locations and etc., by using exploits in the system.

Sailfish Mobile OS RUS is positioned as a secure and trusted operating system. Telemetry will help developers to make the application faster and optimize work. But telemetry functionality should not violate the security and confidentiality of personal user data and business data.

Unfortunately [5], due to the heterogeneous nature of the packages in Sailfish Mobile OS RUS, there is no unified way to enable verbose debug logging, and so the contributor will in some cases have to read the source code to figure out what they need to do to make debug logging work. It may even require setting some #define in the source code and rebuilding the package.

Attempts were made to create a framework Log4Qt under Sailfish Mobile OS RUS for collecting debugging data on the basis of the Java-framework Log4j. But the project was abandoned in 2014. [6]

Before the development, telemetry information should be defined. In results of literature research [7] [8], two main types of the telemetry can be defined:

- **events** - the information from the application source code, such as format string, function name, file name, file line number, module ID, variable arguments, sequence number, and etc. That is information of the application inner state.
- **metrics** - the information about system statistics (CPU, memory, HDD, etc.), buffers filling, threads cyclogramms or synchronization, mutexes, networks delays, packets sizes, etc. That is, information related to the performance of this application.

To separate user credential from telemetry is offered to exclude the opportunity of using the following information in the library methods:

- sensitive personal data and some forms of personally identifiable information
- authentication passwords
- database connection strings
- encryption keys and other master secrets

- bank account or payment cardholder data
- data of a higher security classification than the logging system is allowed to store
- commercially-sensitive information
- information it is illegal to collect in the relevant jurisdictions
- information a user has opted out of collection, or not consented
- and another user sensitive information that is not listed here

Collecting and sending the telemetry data carries a certain risk to the security of the system. There are the following risks when using telemetry:

- data collection by developers for personal use
- creating an additional data leakage channel
- the introduction of malicious software through vulnerabilities in the telemetry system
- interception of personal data when transmitting them over open communication channels
- the use of data by attackers for social engineering attacks
- theft of personal data by third parties.

All these risks were taken into account in the design of the solution since data security is a priority for Sailfish Mobile OS RUS application developers.

B. Architecture

Telemetry consists of three main components: Application, Log-server, and website (Developers' tools). The Application in this case means all telemetry information from the device. If there is a possibility of cellular data transmission, the collected information from one device is sent to the log server through reliable communication channels. If this is not possible, the debugging data is not collected. Developers can view all collected information about applications from the website. Information is available only via HTTPS. Information from all devices are collected into the log-server, that aggregates, sorts, divides and prepares information to the developers.

Whole architecture scheme is shown in the Figure 1. The scheme shows an interaction between different parts of the telemetry system.

C. Library

The telemetry part located on the devices is a library used in applications and a system daemon for collecting and sending information to the server. Developers should use functions from the library to highlight the events that interest them. The functions of the library, called by the application, send key information to the system daemon via the D-Bus system.

D. Daemon

The daemon is a process that runs in the background and waits for the occurrence of the application telemetry events. Once the event has occurred, the daemon works as described below. It collects all the information from the applications into the local storage - Redis data structure store. After the time has elapsed and there is an Internet connection, the daemon sends the data packet to the logging server. Also, daemon collects common information about the device, reads last error logs, prepares telemetry information to transfer and sends information to the log-server.

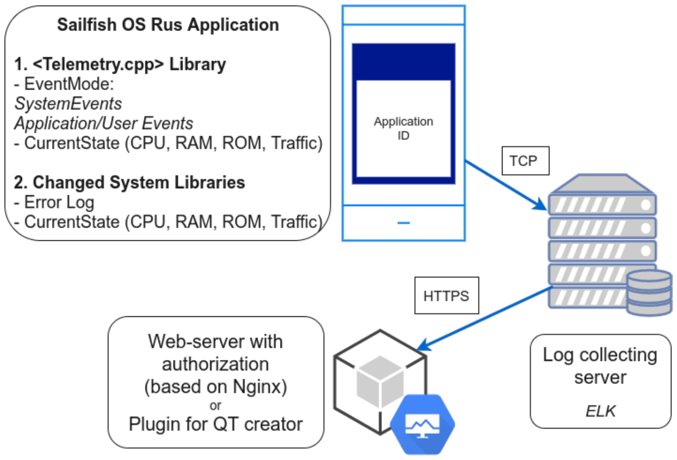
Sailfish OS Rus Application
1. <Telemetry.cpp> Library
- EventMode:
SystemEvents
Application/User Events
- CurrentState (CPU, RAM, ROM, Traffic)
2. Changed System Libraries
- Error Log
- CurrentState (CPU, RAM, ROM, Traffic)
Application ID
TCP
HTTPS
Web-server with authorization (based on Nginx)
or
Plugin for QT creator
Log collecting server
ELK

Figure 1. Architecture of the solution.

Scheme of the Client implementation are shown in the Figure 2

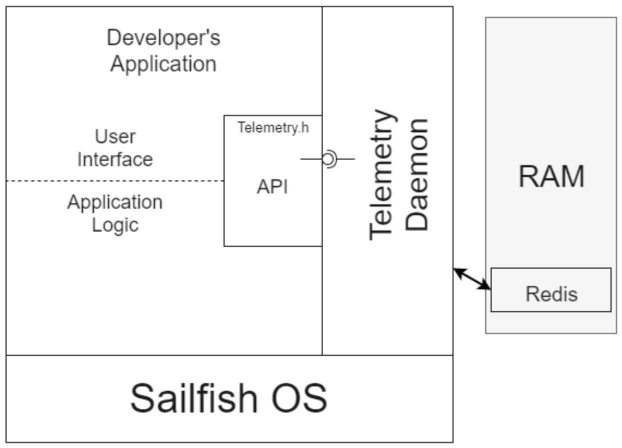

Figure 2. Architecture of the client side.

E. Log server

There is some types logging system with different database models and different approaches for storing and searching a data in a collected logs. Elastic stack and Prometheus was selected as a part of research because only that can receive, store, analyze and represent collected data by themselves. These two products were analyzed to select the best solution.

The Elastic Stack consists of three modules:

- Logstash - application that receiving, filtrate, modify and redirect an input message
- Elastic search - a search and analytics engine based on Apache Lucena

- Kibana - web interface for representation of data from elastic search.

That three components provides all functionality for collecting, storing and processing data

The Prometheus consists of two modules :

- Prometheus - the main module that includes a kernel of application, internal database, and web-interface.

- Alertmanager - notice system, handles alerts sent by client applications such as the Prometheus server. It takes care of deduplicating, grouping, and routing them to the correct receiver integration such as email, PagerDuty, or OpsGenie. It also takes care of silencing and inhibition of alerts.

But that two modules do not allow push-based collecting of data. For this reason a *prometheus pushgateway* is required.

- prometheus push gateway - is responsible for receiving data from remote nodes from outside the LAN.

The first and main difference between Elastic Stack and Prometheus is a database type. Elasticsearch using a search engine that is NoSQL database management systems dedicated to the search for data content. Also, this engine offering the functions like support for complex search expressions, full-text search, stemming, ranking and grouping results, geospatial search, distributed search for high scalability. Prometheus using a Time Series DBMS that is optimized for handling time series data: each entry is associated with a timestamp. That approach leads to Prometheus not suitable for long-term storage(up to 60 days maximum). The second contrast is supporting operating systems. Elasticsearch can be run in any OS with Java VM, Prometheus on Linux or Windows only. Elastic search is a data schema-free, that means the administrator can create their own types in the system. Prometheus support only numeric data only. Also, Elasticsearch creates secondary indexes for each field automatically, supports server-side scripts, triggers. Data can be split into shards and storing on multiple nodes. Prometheus does not control the consistency. Elastic search has a configurable eventual consistency(all, quorum, one).

IV. Proof of concept

Application library is written on the C++ and gives to the developer ability to use clear C++ functions and Qt quick objects. API includes 20 predefined events for application monitoring. All generated information is sent to the D-Bus of the daemon. API does not allow to send user's data because the policy prohibits the sending of any user data, this is the threat of data confidentiality.

The D-Bus connection are shown in the Figure 3.

The library allows to track up to 20 different events in the system. Each event should be spelled out by the developer when writing the application code. The main events are opening, closing the application, loading pages, filling out forms, opening application pages, calling the context menu, accessing other applications and API, errors and exceptions, making transactions, changing settings. Other events are added for the developer, for tracking of any non-standard events or situations.

The daemon is implemented with the Qt framework. Qt provides a set of intuitive and stable bindings to D-Bus API

```
#include <QDBusReply>
#include <QDBusMessage>
#include <QDBusInterface>
struct Telemetry::Impl
{
    const QString serviceName{”ru.sonarh.dbus.telemetry”};
    const QString path{”/ru/sonarh/dbus/telemetry”};
};
void Telemetry::sendMessage(const QString &messageToSend)
{
    auto interfaceName = ”ru.buffer”;
    auto methodName = ”sendTelemetry”;
    QDBusInterface interface(d->serviceName, d->path,
        interfaceName);

    if (interface.isValid())
    {
        interface.call(QDBus::Block, methodName,
            messageToSend);
    }
}
```

Figure 3. D-Bus Connection Code

in the form of C++ classes. In a common way, the D-Bus technology works on POSIX-compatible operating systems and is used for Inter-Process Communication. This technology allows getting information from one process to another. The information transmitted can take the form of application data, method calls, or signals.

Daemon also collects information about device. Example of collected info are shown in the Figure A. Piece of collection code are shown in Figure A. Code for collection information about process are shown in Figure A.

Elastic stack was selected for storing and representation of logs, according to research conducted in section III-E. ELK receiving data from nodes via TCP connection. Authentication of data provides the certificate. Authorization for users provides with Nginx web-server.

V. Discussion

This model guarantees the security of personal and business data when collecting and processing telemetry data, but it can not prevent an unscrupulous developer from changing the telemetry library and introducing destructive functions into it.

In addition, perhaps there will be a case where developers need to work with personal data. In this situation, all responsibility for this data lies with the developer. To safely store such user data, it is required to use hashes (*eg*, SHA-1) instead of plain text.

Outside the study, it remains that when working with corporate data, information is collected from many users, whose statistical analysis can give some general private information about the company.

VI. Conclusions and suggestion for further research

When collecting debugging data, mobile users must be sure that their personal and business data are protected in a secure manner. However, there exists no work on studying such practices in mobile industry or helping developers make informed decisions. To fill this significant gap, this paper presents our studies on investigating the technical ways of implementing a methodology for collecting debug data from the application for Sailfish Mobile OS RUS so that it ensures the security of personal and business data of mobile users. We focus on three main issues: what is telemetry, what information should be collected and what architecture to implement. We provide fairly complete solutions to the above issues. As a result of the project, the basic concepts of telemetry collection were derived. A library and a daemon to collect data from applications have been written, a test server has been raised to collect logs. The system had tested on the Sailfish Mobile OS RUS virtual machine and on the mobile device with the installed test application.

In addition, there are some potential directions for improving the existing methodology, which is valuable for further research. These include: a secure storage of personal and business data, as well as protection from static analysis of corporate data.

This study facilitates the understanding of modern telemetry methods and is the first step in improving the practice of telemetry in the industry.

Full source code can be open on the GitHub: Daemon Code and Library Code

Appendix A
Code and Examples

```
"OS Release" : "NAME=SailfishOS
ID=sailfishos
VERSION=\"2.1.3.7 (Kymijoki) (armv7)
VERSION_ID=2.1.3.7
PRETTY_NAME=\"SailfishOS 2.1.3.7 (Kymijoki)
(armv7)
SAILFISH_BUILD=7
SAILFISH_FLAVOUR=release
HOME_URL=\"https://sailfishos.org/

"Statistics" : "cpu 373 7 177
21380 1510 57 22 0 0 0
cpu0 373 7 177 21380 1510 57
22 0 0 0
intr 30394 118 11 0 0 0 0 0
0 6 0 0 0 153 0 0 0 198 0 0 719 399 5061

"Memory Information" : "MemTotal: 1028972kB
MemFree: 612832 kB
Buffers: 13544 kB
Cached: 176576 kB
SwapCached: 0 kB
Active: 235104 kB
Inactive: 149676 kB
Active(anon): 198868 kB
Inactive(anon): 4672 kB\
```

Figure 4. Example of collected information

```
  const QString buddyInfoFile                =
      "/proc/buddyinfo";
    const QString buildPropertiesFile          =
        "/system/build.prop";
    const QString characterAndBlockDevicesFile =
        "/proc/devices";
    const QString cpuInformationFile           =
        "/proc/cpuinfo";
    const QString defaultPropertiesFile        =
        "/default.prop";
  ..............................
    const QString partitionsFile               =
        "/proc/partitions";
    const QString sailfishReleaseFile          =
        "/etc/sailfish-release";
    const QString systemReleaseFile            =
        "/etc/system-release";
    const QString voldFstabFile                =
        "/system/etc/vold.fstab";

struct DeviceInformation::Impl
{
    QString readDataFromFile(const QString& fileName)
    {
        QFile file(fileName);

        if (!file.open(QIODevice::ReadOnly |
            QIODevice::Text))
        {
            qDebug() << "Can't open" << fileName <<
                "file!";
            return {};
        }

        QTextStream in(&file);
        return in.readAll();
    }
};
...
QMap<QString, QString> DeviceInformation::allInformation()
    const
{
    return {
        {
          "Buddy Info",                this->buddyInfo()
            ...
        }
}
QString DeviceInformation::buddyInfo() const
{
    return d->readDataFromFile(::buddyInfoFile);
}
```

Figure 5. Piece of code that collects information about device

```
#include "process_information.h"
#include <QProcess>
#include <QTextStream>
namespace
{
    const QString psProcessName = "ps";
    const QStringList psProcessParameters = {"axo",
        "%cpu,%mem,comm"};
    const int waitForStartedInterval = 2000;
    const int waitForFinishedInterval = 3000;
}
struct ProcessInformation::Impl
{
    QList<ProcessAttributes> parsePsData(QTextStream&
        stream)
    {
        QList<ProcessAttributes> attributes;
        QString line;
        while (stream.readLineInto(&line))
        {
            auto splittedLine = line.split(" ",
                QString::SkipEmptyParts);
            if (splittedLine.size() < 3) continue;
            // see attributes order in psProcessParameters
            attributes.append(
            {splittedLine.at(2),
             splittedLine.at(0).toDouble(),
             splittedLine.at(1).toDouble()});
        }
        return attributes;
    }
};
ProcessInformation::ProcessInformation() :
    d(new Impl)
{
}
QList<ProcessAttributes>
     ProcessInformation::allProcessAttributes() const
{
  QProcess psProcess;
  psProcess.start(::psProcessName, ::psProcessParameters);
  if (!psProcess.waitForStarted(::waitForStartedInterval))
      return {};
  if (!psProcess.waitForFinished(
       ::waitForFinishedInterval))
      return {};
  QTextStream in(&psProcess);
  // skip header line
  in.readLine();
  return d->parsePsData(in);
}
```

Figure 6. Code that collects information about process

References

[1] Federal Law of the Russian Federation No. 152-FZ "On Personal Data"

[2] Federal Law of the Russian Federation No. 98-FZ "On Commercial Secrets"

[3] Business data, their characteristics, types and sources of income. http://www.zavtrasessiya.com/index.pl?act=PRODUCT&id=1875

[4] TestFairy - a platform for testing applications for Android. https://testfairy.com/

[5] SailfishOS Developer Documentation https://sailfishos.org/wiki/Testing

[6] Log4Qt for Jolla Sailfish Mobile OS RUS and in general http://www.codingsubmarine.com/tag/logging/

[7] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? An empirical study on logging practices in industry. May 2014.

[8] Titus Barik, Robert DeLine, Steven Drucker, and Danyel Fisher. The Bones of the System: A Case Study of Logging and Telemetry at Microsoft. May 2016.

[9] Colin Eberhardt, The Art of Logging, March 2014. https://www.codeproject.com/Articles/42354/The-Art-of-Logging

[10] P7 library documentation http://baical.net/p7.html

[11] 7 Good Rules to Log Exceptions. http://codemonkeyism.com/7-good-rules-to-log-exceptions/

[12] 7 More Good Tips on Logging. http://codemonkeyism.com/7-more-good-tips-on-logging/

[13] 10 Tips for Proper Application Logging. http://www.javacodegeeks.com/2011/01/10-tips-proper-applicationlogging.html

[14] Logging best practices http://dev.splunk.com/view/logging

[15] Logging Cheat Sheet. https://www.owasp.org/index.php/Logging_Cheat_Sheet

[16] Java Best Practices for Smarter Application Logging & Exception Handling. https://stackify.com/java-logging-best-practices/

[17] Discover Logging Best Practices. Part 1: Collecting Logs. https://logmatic.io/blog/beyond-application-monitoring-discover-logging-best-practices/

[18] Application Logging: What, When, How. https://dzone.com/articles/application-logging-what-when

[19] What Not to Log When Remotely Debugging Mobile Apps. https://bugfender.com/blog/what-not-to-log-when-remotely-debugging-mobile-apps/

[20] How to Remotely Debug Mobile Apps. https://bugfender.com/blog/how-to-remotely-debug-mobile-apps/

[21] What Mobile App Metrics You Should Look For? https://bugfender.com/blog/mobile-app-metrics-look/

[22] IBM MQ Telemetry. https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.0.0/com.ibm.mq.pro.doc/q002760_.htm

[23] Kibana – make sense of a mountain of logs. http://kibana.org