UNIVERSITY OF GOTHENBURG

Introducing the three tier model for app security and reliability in critical systems

An exploratory practical approach

Bachelor of Science Thesis in the Programme Software Engineering and Management

PER LUNDIN
Erik Kinding

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2011

Introducing the three tier model for app security and reliability in critical systems

An exploratory practical approach

Per Lundin
Erik Kundin

Examiner: Helena Holmström Olsson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2011

INTRODUCING THE THREE TIER MODEL FOR APP SECURITY AND RELIABILITY IN CRITICAL SYSTEMS

AN EXPLORATORY PRACTICAL APPROACH

Per Lundin
Dept. of Computer Science & Engineering
Chalmers Univ. of Technology
Forskningsgången 6
Gothenburg, Sweden
luper@student.chalmers.se

Erik Kinding
Dept. of Computer Science & Engineering
Chalmers Univ. of Technology
Forskningsgången 6
Gothenburg, Sweden
erikkinding@gmail.com

Academic supervisor: Carl Magnus Olsson
Industrial supervisor: Gunnar Andersson
May 2011

ABSTRACT

THE MODERN APP PHENOMENON HAS SPREAD FAST AND WIDE WITH THE MAJOR SPREAD OF SMARTPHONES. THIS PHENOMENA IS NOW TREADING NEW GROUND, AND STARTS TO SURFACE IN OTHER MARKETS. WHILE THIS IS HAPPENING, THERE IS LIMITED RESEARCH DONE ON THE SECURITY ASPECTS OF THESE APPS ON THE PLATFORM THEY INHABIT. LIKE THE PARASITES OF BIOLOGY THESE APPS COULD EVENTUALLY PROVE TO BE EITHER HARMFUL OR USEFUL FOR THE PLATFORM. THIS PAPER WILL LOOK AT SOME OF THE CHALLENGES FACING THE DEVELOPMENT OF APPS AND OF THE PLATFORMS THEY INHABIT.

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2011

INTRODUCTION

The in-vehicle infotainment(IVI) systems of modern cars are moving towards the rest of our mobile devices such as netbooks, tablets and smartphones in terms of functionality. A very clear example of this is the Android based platform recently released by Saab, called Saab IQon (Saab automobile, 2011). This platform includes the functionality found in any modern smartphone and even allows the customer to continually extend the functionality by downloading so called ‘apps’ (small lightweight applications, that usually fulfill a single purpose).

About a year ago, project MeeGo was launched (MeeGo 2010). It was a merge between two separate projects aimed at mobile platforms, Intel's Moblin and Nokia's Maemo. The new platform would also come to include a branch for in-vehicle devices, known as MeeGo IVI (Schroeder, 2010).

MeeGo is an open source project, running on a Linux kernel, which give developers and users good possibilities to configure it to their own preferences.

A company in the automotive industry (CarComp) was interested in if MeeGo IVI has the capability to provide a secure and reliable platform for third party applications. An investigation of this new platform would have to consider many factors. Two of these, which are common concerns with any software supposed to run in a car, are security and reliability. As understood by the mere existence of Saab IQon and the now noticeable success of the app concept , (Sharma 2010) it is of course highly interesting to investigate the possibilities of letting a user run third-party software, which they can download from the internet, in their car. The concern of this paper is how third-party software can be developed for MeeGo IVI in respect to security and reliability concerns? The security and reliability became the main focus based on discussions we had with an engineer at CarComp. An example that usually was stated was that an app malfunctioning should not be able to disable the breaks on a running car. It's from these discussions the concern for this paper was born.

In our effort to answer this we used a methodology called design research. This will be thoroughly explained in the method section of this paper. The data we collected was both qualitative and quantitative. Qualitative data that was collected was chronically written descriptions of the implementation effort. The quantitative data that was collected was in the form of security and reliability requirements. The requirements was linked and traced in the development effort as well as the prototype artifact.

The outcome of this research is in the form of a proof-of-concept app environment operating on the MeeGo IVI platform. The prototype should encompass all the security and reliability issues that will be presented in this thesis.

Following this introductory section we will present our theoretical framework, the lens through which we look at our problem. Further, we will present the method used, the result of the research process and then discuss these findings under the light of our specified framework. Lastly, we sum the paper up in a concluding and discussing section.

THEORETICAL FRAMEWORK

The research in this thesis was done at the infotainment department at CarComp. CarComp is interested in the new IVI platform, MeeGo. The application we produced will help CarComp evaluate MeeGo as a viable platform for future IVI systems.

CarComp is now facing questions concerning the security and reliability of the underlying system structure. How can an application execute in a safe environment, especially if it's developed by third party developers? The area looked at in this thesis was framed from a security and reliability perspective. McGraw(2004) describes some examples of handling security, one of which is sandboxing, which would play a major part of the development effort.

Saab has recently released an IVI system with the focus on letting third party developers develop applications for it. The idea for this thesis was to develop a secure application environment for third party developers. The focus was on how to do this in a secure and reliable fashion. We have concluded that normal software security and stability requirements, such as Sommerville(2007) suggests *authorization requirements, integrity requirements, intrusion detection requirements* are not viable for this kind of system. The same goes for reliability requirements and metrics. This is because of the nature of the application, and the fact that it is intended to run on a critical system. In the app environment that was produced, security and reliability were the main requirements.

Desmet et al.(2007) describe five important architectural requirements for this kind of system (1) secure execution of third-party applications (2) support for the security-by-contract paradigm (3) flexible integration of enforcement techniques (4) optimized for resource-restricted devices (5) compatibility with legacy applications. The first architectural requirement states that a software running should not be able break the underlying structure. The second focused on security-by-contract, that security of the system can be based on contracts. The third states that on-device security enforcements should be flexible. The fourth is about the resource requirements needed. And finally the fifth states that the architecture should be able to work with legacy software. We will focus on the first, third and fourth. This is because of the focus of this thesis, namely security and reliability. These three architectural requirements were the ones that was best suited for the application. Secure execution of third-party applications is also identified as the primary requirement for this kind of system. This does not mean that the other architectural requirements will not be satisfied. As Desmet et al.(2007) explains, the architecture of the underlying structure must make sure that the application can't jeopardize the reliability of the system it is run on. From this we extracted our two main focuses, security and reliability. We will use parts of the suggested architectural design for safe application execution that is suggested by Desmet et al.(2007) in our development efforts. The following two sections discuss the focus of the security and reliability concerns.

SECURITY

Every decision made in the development process was made from a security perspective. For the implementation process the security aspect would be evident in the following description: An application should not jeopardize the foundation system that it is run on, it should not be able to access data that is not intended for that application. The sandboxing plays a major role in the security aspect, as it will limit the execution of applications.

RELIABILITY

As with security the reliability aspect would color the entire development process, as well as the research process. When it comes to implementation process the following definition is used: if an application is badly written and crashes, the underlying structure should realize this and shut it down. Sandboxing would play a role in reliability, since applications run in the sandbox environment should not be able to halt the entire underlying structure. Sandboxing is a term used in computer science when an application is run in a closed off or simulated part of a system. Developers can usually create a sandboxed environment where an application cannot harm the underlying systems core functionality. This is done mainly to maintain reliability, but also for security, such as running untested or unsafe files (Prevelakis & Spinellis 2001). Sandboxing was one of the key areas that we explored. It was important to look at how sandboxing can contribute to software security and reliability.

Important to keep in mind is the fact that a completely secure system most likely is a system turned off or completely disconnected from the world around it. Thus, our efforts cannot be seen as solutions to perfect security but rather attempts to enhance security where needed and possible. We have mainly been keeping the end users (device owner) and app vendors in mind as key stakeholders of security aspects.

The implementation effort executed during our study attempted to shed light on issues related to the implementation of an app environment. Studying how other platforms solved this issue, we set out to implement and investigate the feasibility of solutions for the MeeGo platform. A solution should add value of some kind (in our case security and reliability) to stakeholders of the system. Along with the development oriented parts of the investigation, we wanted to see how the actual system could be tweaked and used. Looking at how it has been done previously, the concept sandboxing seemed to be a reasonable place to start.

We looked at different ways to solve the app environment problem for a critical system, we studied the way two of the major players in the app-market solve these issues. The two that we looked at are Android and iPhone (iOS). This is in an attempt to gather more information of how this problem has been solved before. Whether a mobile device such as a mobile phone is a critical system was not the focus of the study.

THE THREE TIER MODEL

This section will introduce a model that we call the 'three tier model' that was conceptualized during the development, it will be linked to requirements and show examples of how specific requirements were handled in the prototype artifact.

The application of this model is supposed to be in the area of critical systems. The model will make sure that the underlying operating system will not be harmed in any way by the third party software that the user run. The model describes three tiers of security, the idea is to make sure that if one of the tiers break there will at least be another one that will make sure that the underlying system is not harmed. First we will introduce the thought behind this model, by shortly describing each tier and give some examples how it can be implemented. Later we will show how it was applied to our prototype artifact.

The three tier model is comprised of three security levels. The first one is to set limitations on the development language of the apps, which means the specific language the apps are written in. We hypothesized that this would greatly enhance the security of the system, by constraining what the developer can do, but also force the developer to use specific purpose designed libraries.

Second level is to have a controller that handles the different processes that the apps run in. This affected both of the identified requirements, security and reliability. The security requirement benefit from this by having each app run in a separate process, so all memory and data that is used would be local to that app. The reliability requirement will benefit from this because if the app process crashes or freezes it will not interrupt the underlying operating system. The controller will also monitor the app process for errors. The hypothesis behind this level is that each app needs something that controls it, making sure it's not doing something it is not supposed to do.

The third part is to have a safe place to run the app in, that is away from the basic functionality of the underlying operating system. We hypothesized that this fulfilled most of the reliability requirement, by closing off parts of the underlying operating system, so it can't be modified in an undesirable way.

A detailed explanation of each tier will follow, with examples of how that tier can be implemented.

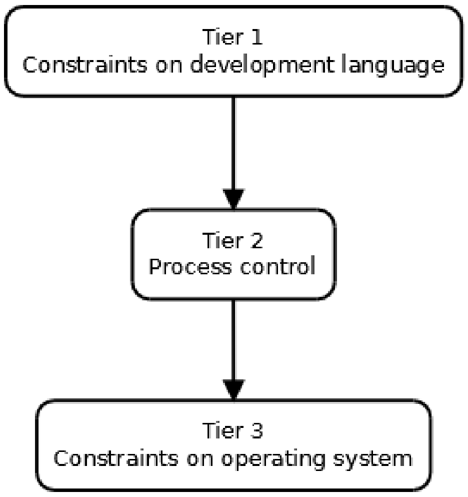

Tier 1
Constraints on development language
Tier 2
Process control
Tier 3
Constraints on operating system

FIGURE 1

TIER 1: CONSTRAINTS ON THE LANGUAGE

This tier focuses on the way that the apps are written and putting limitations on the specific language that is used. The idea behind this is to make sure that the author of an app can't call functions that are potentially harmful to the system. There are many ways that this is achievable, either from the use of a scripting language for the app implementation, or by forcing the developer to use specific, purpose built libraries. The idea is not to limit the app developer, but give more of a rigid framework for development. Since this model is meant for critical systems, the idea of having an open platform is not recommended. Forcing the developer to use an API that has limited functionality is important for the security aspect of the system. Requirements that are addressed by this tier is security. This tier covers the flexible integration of enforcement techniques described by Desmet et al.(2007), by letting the developer of the system easily change the way that security is enforced on the app development language. Desmet et al.(2007) discusses different ways this can be achieved, one of the ways to let the app developers have more access is to enforce the use of signatures. Where registered and trusted app developers can get a signature to distribute with their apps, that will allow the app to use functions that the general public cannot. This is one example of how such technique can be implemented in the model. Another example is enforcing the use of a specific API(application programming interface), this API can be formed to only let the app developer to use specific libraries when developing. This is more like the Android approach, where Java is used. Java is a general purpose programming language, but has a specific, purpose built programming API for Android development. A third example is to force the app developer to use a specific programming language, that is either built for the app development, or restricted for such use. This is more how iPhone have solved this problem, by not permitting development for their platform in other languages than C, C++ and Objective C. Any of these examples give a good solution to solving this tier.

TIER 2: PROCESS CONTROL

Even if the developers of the system can put limitations of how the apps are written, they cannot limit what the app actually do. This is why we felt the need for some sort of controlling mechanism. The idea behind this tier is to limit the execution of apps, and to monitor their behavior. If an app starts to use up too much resources, and starts slowing down the underlying system the controller should have the authority to shut the app down. The same goes for if an app crashes and becomes unresponsive. Not being able to control the underlying system could be considered an extreme reliability risk. So the main focus for this tier is the reliability of the system. An example of how this can be implemented is to have a program that spawns each app process and monitors them. This is a good solution, mainly because there is a separate process that keeps track of the spawned app processes. So as long as this process is not compromised the apps can't do any harm on the system. The process controller could also monitor its own process, and if it is compromised it will close itself down along with all the app processes. Another way of solving this tier is to use a native operating system constraint on processes. For a GNU/Linux system, this is quite easy, the system developer can limit how many processes that can be run at the same time. This means that no other processes can start until another process has been shut down. This can be achieved with commands such as ulimit in the GNU/Linux environment. Since MeeGo is based on Linux it's a viable solution for our prototype as well. A third way of solving this tier is only to have specific user-groups for specific functionality. For example having a user-group that is allowed to use the internet connection on a platform, and assigning the app rights to that user-group, and then monitor that group. If the app abuses the terms of the user-group it can easily be unregistered from that group, not allowing it access to that functionality any more.

TIER 3: CONSTRAINTS ON OPERATING SYSTEM

Sandboxing is a fairly abstract term which gathers many techniques used to enhance computer security and reliability. Using a sandbox is a common way for running untrusted software on a system without the risk of harming the underlying system. An app, or more generically third party software, can be considered to be untrusted software. Our research started out with investigating so called jails, which intuitively seemed to be what a sandbox was all about. Alternatives to this idea was discovered and investigated.

Examples from the previous section, Tier 2, also qualify as examples for this tier, more precisely the utilization of mechanisms in the Linux kernel and software usually found in GNU/Linux distributions. Creating a sandbox as part of this tier could mean setting up restricted user accounts under which processes run and thus have limited access to system resources, utilizing the Linux Security Modules (LSM) framework for mandatory access control (MAC) or virtualizing a separate and isolated filesystem using a software such as Linux-VServer.

The way users are handled in the Linux kernel is both a very viable and fairly easy mechanism to use for restrictions. Users can be allowed permission to different sets of system resources and also bundled into groups sharing the same permissions. This is for example used in the Android security model.

Mandatory Access Control. MAC is for instance used in the iPhone iOS sandbox where it is based on the TrustedBSD framework (Dwivedi et al, 2010). To briefly describe MAC, it is a way for the operating system to constrain what a program is allowed to do with a piece of data or other system resource. As presented in the result section later in this thesis, MAC can be implemented in MeeGo and GNU/Linux systems using software which utilizes the LSM framework.

So called jails, which immediately drew our attention to the chroot mechanism, is a isolated section of a filesystem where a piece of software is allowed to execute without interaction with the underlying system which is thus protected. Chroot in itself was however discovered to be a poor choice, but alternatives fulfilling the actual intention of the jail implementation was discovered and is presented in the result section.

This tier also covers the secure execution of third-party applications requirement described by Desmet et al.(2007)

METHOD

RESEARCH SETTING

A major part of the research effort presented in this thesis was executed at the infotainment department at CarComp. The intention was to be working in the same environment as the architects and requirements engineers at CarComp and benefit from their knowledge and help.

MeeGo is a platform based on the Linux kernel and is very similar to other GNU/Linux distributions, such as Ubuntu Linux. The aspects of the MeeGo platform this paper discusses, security and reliability, can in several ways be taken care of fairly easy by embracing standard functionality of a GNU/Linux distribution. MeeGo is bundled with an IDE based on QT-creator, which rapidly increased the setup time needed to get everything in order and to start developing our solutions.

To get some perspective of how one can solve the execution of apps on a system we looked at Android and iPhone.

In large, Android apps are only as secure as the user wants them to be. They rely heavily on security permissions that are presented to the user when the app is installed on the system. If the user accepts that the app will have access to a particular system functionality, the app will have complete access to it. When an app is installed on an Android device it is given a specific user id, Dwivedi(2010), this allows the app only to use functionality from each specific user group that it belongs to. The way this is done is by having specific user groups for different functionality(such as using internet connections, or the address book) and prompt the user if he or she will allow the app to be a part of the specified user group. This is not viable in a critical system, since the ability or non-ability to make an informed decision should not jeopardize the system.

iPhones take on security differs from Androids. Apple have the right to exclusively control what apps run on their system, unless the device is 'jaibroken', but this is outside the scope of this thesis. Apples exclusive domain of what apps are available to their system is based on the marketing model of their apps, Dwivedi(2010). Any developer can develop apps for iPhone, but if the app you produce violate any of Apples guidelines for apps the app will not be available through their appstore. All the source code for apps are approved by Apple before they are published, this kind of security makes it hard developers to sneak in security or reliability breaches. But it also hinders the development of apps, since every app has to go through the process of being reviewed. This rigid inflexible system that the iPhone is using can be viable, based on the corporate marketing model of the company that wishes to introduce the system. If the corporation that is implementing such a system does not have the resources to check all incoming apps for security breaches and unsafe code, it is not practical with such a solution.

It is in this middle-ground that the idea of our model was hatched. How can a company let third party developers develop apps for their critical system without compromising that security and reliability, and without putting the burden of these requirements upon the user. By experimenting with the already present concepts of app security on the MeeGo platform we managed to raise interesting issues to the surface and present insights regarding security and reliability concerns discovered during the research.

RESEARCH PROCESS

This thesis project was executed according to the design research methodology (Hevner & Chatterjee 2010,). Design research was identified as suitable for a couple of reasons, amongst them the fact that our research has been dealing with a fairly new and unexplored domain. The manner in which design research let us design, experiment, implement and redo as needed was very important in the choice of research process. This section will discuss the steps in the design research process, how each step is related to our research and by this justify the choice of methodology. Important to keep in mind while reading about these steps is that they were not followed exactly or in an as strict manner as one might think. Again, it was more the experimental nature of design research which was appealing rather than the proposed structure and order found below.

Five steps were identified in the design cycle (Hevner & Chatterjee 2010,):

AWARENESS OF PROBLEM

The automotive industry is pushing their in-vehicle infotainment (IVI) systems the same direction as other mobile platforms in terms of functionality. MeeGo is a new platform and one of its intentions is to be used as an IVI platform. Saab IQon, being based on the Android platform, is a great example of the emerging IVI field. An interesting problem is how third-party software could be handled on the MeeGo platform, similarly to the apps available for Saab IQon.

Further, a set of important software qualities concerning IVI systems were discussed. It became evident that security and reliability was of outmost importance. The output of this step is intended to be a proposal for a new research effort. We ended up with the following research question: "How can third-party developed software be run on MeeGo IVI and respect security and reliability concerns?".

SUGGESTION

From discussions with an employee at CarComp and by looking into existing knowledge on the area, an implementation of a sandboxing solution was suggested. Initially this was a spontaneous suggestion which seemed to be appropriate and literature confirmed this suggestion, Prevelakis & Spinellis(2001), McGraw(2004).

DEVELOPMENT

During this phase, we developed and researched a solution to our identified problem. Design is of big importance in this step and the goal was not only running code on the target platform but also ideas regarding architecture and further development. This together would constitute the model presented and discussed throughout this thesis.

EVALUATION

During this step, which in this specific project ended up to be very much like the next and final step, our main contribution to the thesis was reached. Collected data, assumptions regarding design, deviations from our expectations and personal reflections was gathered and discussed. Design research executed by the book would end this step by starting over at the suggestion phase with knowledge gained throughout the last loop in the design research cycle. Due to our narrow time frame the next step, reflection, was reached immediately after this step.

REFLECTION

Loose ends and still unsatisfied requirements from our development effort was noted and the discussion section of this thesis was finished up. Here suggestions for future research efforts on our topic was identified and presented. These suggestions can be found in the end of the discussion section.

DATA COLLECTION AND DATA ANALYSIS

As can be figured out from the example-like description of our work process above, essentially two categories of data was collected: logbook notes and data from the evaluation step as described in the design research methodology.

We documented our work progress on a daily basis. Each logbook entry was written in a structured, pre-defined, format to easily let us make use of the notes. Our thoughts during the project were important as our perception of the MeeGo platform became part of the evaluation process.

During the research, data in form of requirements which can ensure or help a system (of the studied kind) to be secure and reliable was collected. During analysis and also during data collection, requirements like this appeared. The key to gain such knowledge is through deeper insight, which in turn was given by interaction with the platform. Interaction in this case would mean development for the MeeGo platform. This knowledge that was analyzed using our theoretical framework would specify a model which can help to assure security and reliability of the studied system, but also systems sharing its technical characteristics, such as the fact that MeeGo is a GNU/Linux system.

LIMITATIONS

A system operating in a car needs to take a lot of different technical concerns in to consideration, especially from a safety perspective. Systems should not be able interfere with and cause each other to crash. This could have lethal consequences if, for instance, a system controlling the breaks all of a sudden would crash.

This project was carried out using hardware unattached to a car and nothing of what we implement or investigate is taking in to account the communication or infrastructure of in-vehicle electronics. This thesis puts a focus on software engineering and hardware aspects could not fit in the scope nor time frame of the project.

It was decided to keep the scope of the thesis to third-party software in relation to security and reliability. This is not a narrow scope, but it will keep the project on track while still being as permitting within its boundaries as our exploratory research would require.

RESULT

Here we will present the result of the development and research effort. In this section, the details of the implementation are not discussed in very much depth. If you wish to get a deeper understanding of the implementation, please see Apendix A for the entire source code, including example code for an app.

THREE TIER MODEL IN PRACTICE

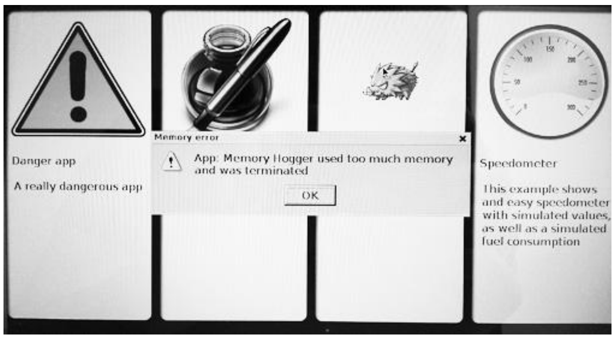
Here we will describe how we implemented the three tier model in our proof of concept prototype. We will go through each tier like the previous section and show how it was implemented.

TIER 1

In our prototype we took the first approach exemplified in the theoretical framework section, and implemented a Lua interpreter in the code that would run each app. Lua (Lua 2011) is a lightweight scripting language, written in C, and proved to be really useful for this application. And since MeeGo IVI is Linux based and the implementation language used for our app environment was C++, Lua provided us with the functionality to not include specific libraries that could be harmful to the system, such as IO and OS libraries. If such functionalities that the IO library provides would be needed, we would implement a function that would mimic that functionality and export it to be used by Lua. But the functionality of such a function would be controlled by the system, and not by the app. This also provided us with the ability to build a specific GUI controller for the app, which greatly improved the ability to write graphical apps. The more non-API functions that can be mimicked in the API of the app development language, the more secure the app environment will become. Since our prototype is just a proof-of-concept, we decided only to include about 10-15 functions, such as rendering, image handling and events. Another idea behind this approach is the ability to check all the input that the app handles and reject input that could be harmful for the system. What we realized after exporting functions to Lua, was that even though we exported a small amount of functions we could accomplish running apps in MeeGo. The result from this tier, proves the idea behind this tier, by allowing the developer to write an app that can't be harmful to the system, because of the limitations we made to the language. We tested this by creating an app that tries to use a library that we wouldn't allow it to access. This resulted in the app closing down with an appropriate error message.

TIER 2

We implemented a controller in to our prototype that would spawn each apps process and monitor the resources used by it. If an app would start using too much resources in the form of memory or process power the controller would simply close down that running app. The implementation of the controller was not a really hard task to complete because of the way MeeGo IVI is built. This made it really easy to spawn processes and control them using the native Linux implementation of the proc tree. The controller was imperative for an app running on a critical system, if the system is malfunctioning safety and security of the user can be at risk. Even a slowdown of such a system, for example a process consumes too much process power, can put the system at risk. Having each app run in its own process is a major benefit when working with multiprocessing operating systems, such as Windows or Linux. It gives stability that even if the process somehow crashes or freezes, the underlying system is not particularly effected, and can continue execution. We deemed this necessary because of the domain that this system would run in, which is a moving car. The idea of running each app in a separate process is used by other developers such as Android. This along with our prototype shows that it is a good solution. For additional security it's possible to have certain functionality tied to specific user groups, Dwivedi(2010) describes the way Android have solved this issue, but also points out some of the downsides with it, as mentioned earlier Android bases a lot of the security on the user's ability to make correct assumptions about the software. This was something we deemed as a security risk for our prototype, and we decided on developing the above mentioned interfacing with an API instead. Figure 5.1 shows our process controller in action. In this instance the app called memory hogger used up more memory than what was allowed and was forced to shut down. This shows what we intended for this tier. The app is able to run as long as it's not consuming more memory than we allowed it to consume. When the memory usage exceeded the threshold set for it, the app was closed down and an approriate error message was shown.


Memory error

App: Memory Hogger used too much memory and was terminated

OK

Danger app

A really dangerous app

Speedometer

This example shows and easy speedometer with simulated values, as well as a simulated fuel consumption

FIGURE 2

TIER 3

On an early stage, we decided to go with a mechanism called chroot ('change root'). This mechanism has been around for a long time and is a common feature in Unix and GNU/Linux systems. A chroot "jail" would let us run an app in an isolated section of the system and thus prevent it from harming the underlying system. However, it was discovered that there is really nothing preventing a running process from leaving the jail and cause damage or other kinds of unwanted behavior. Further reading on the topic would lead us to dismiss the idea.

"chroot is not and never has been a security tool"
(Alan Cox, Kerneltrap 2006).

This statement was backed up by quotes from the manual of chroot which states that the current working directory (cwd) of the process remains the same. Due to this fact, it is possible to break out of the jail. Also, the fact that only root (UID 0) has the ability to use chroot leaves the door to the jail fully open in practice. Examples of how this can be done is more elaborately shown at the homepage of the Linux-VServer project (Linux Vserver, 2009,). Our understanding is that chroot can be used to create a separate section of your system where you can run software you do not trust or want to test. However, the protection the chroot environment offers does not help if the software you do not trust knows that it is running in a chroot jail and can break out.

We spent quite some time (a major part of the time dedicated for implementation) trying to propose a chroot based sandbox on MeeGo. Once the problems, or facts, became apparent to us, we started to investigate related software and other ways in which sandboxing could be done.

Alan Cox mentioned in the same thread on kerneltrap that software has been built for the purpose of security upon the idea or concept of chroot, Linux-VServer and FreeBSD Jail are examples. There are more alternatives than these. Here we present four of them and briefly try to describe their purpose and functionality. All are developed for Linux systems which makes them possible candidates for the MeeGo platform.

Linux-VServer adds an extra barrier to overcome the previously mentioned issues with a chroot jail and thus make it more apt for a security oriented purpose. This software makes it possible to run several, separated, virtual servers on one physical machine. In the context of this paper, this functionality would more or less do the exact same thing we intended to do with chroot. One virtual server, which is isolated from the underlying system would constitute the jail. (Linux Vserver, 2009)

There are no sources indicating the feasibility of using Linux-VServer on MeeGo, but in a discussion on a Maemo project (one of the predecessors to the MeeGo project) forum (Maemo, 2011) it was mentioned as a possible candidate for setting up a jail environment.

Another candidate for this section was SELinux, but from our understanding based on various online discussions, it is perceived as hard to use and maintain (Linux 2006). An alternative to SELinux suggested on the Maemo discussion boards was AppArmor. It was only briefly mentioned but caught our attention. Apparently, AppArmor is easier to use than SELinux, one of the main selling points of AppArmor seems to be the ease of use (AppArmor 2011). The two differs on a fundamentally technical level, but this has not been investigated in any closer detail.

There seems to be no information on the possibility of using AppArmor on MeeGo and we have not tried this ourselves during the project but there is nothing indicating that an implementation would be impossible. As long as MeeGo is used with a kernel version supporting LSM this option is interesting.

Further, we found SMACK and TOMOYO Linux. Both are a bit more interesting than AppArmor since they seem to have been used in practice on the MeeGo platform in one way or another. Regarding TOMOYO Linux there is not much to say. It is a software used for MAC, similar to AppArmor. However, a tutorial (Tomoyo Linux 2011) on how to run TOMOYO on MeeGo was found. This is a life sign and TOMOYO might be a suitable tool for MAC on MeeGo. We have however not been able to try this out in practice during the project.

Finally, we started to look at SMACK. Edge (2010) reports from a conference for lwn.net that SMACK is decided to be the tool used for access control. This is further confirmed in the MeeGo projects wiki page on architecture (MeeGo wiki 2011) This option seems to be the official choice for MAC this far. The author of SMACK, Schaufler (2007) describes his software in more depth in his paper.

CONCLUSION

The main contribution of this thesis is the three tier model, but also the collection of thoughts and ideas that arose during the development and research processes. The model is not the complete solution to the problem of app security, but it can at least give CarComp, and others who are in the process of implementing an app environment, some idea of what to consider.

The suggestion of app environment on the MeeGo platform we have proposed manifests itself to work in practice and theory. Parts of it in mere practice shown by our implementation effort, parts in theory which led to conclusions based on research. The proposed environment is not a complete app environment, but it contains, along with the presented ideas in this paper, a foundation for a body of knowledge which could be used to further develop an app environment on MeeGo or, as far as we have seen, any other GNU/Linux system providing the same amount of functionality.

Because of the way we implemented Lua and the way we wrote our app-environment the entire system is really lightweight, which makes it usable for systems with limited resources. And since the apps are written in a script language the apps are lightweight as well. This is because all the heavy GUI implementations are already implemented in MeeGo. Furthermore the app environment that we produced covers 1 (secure execution of third-party applications),3 (flexible integration of enforcement techniques) and 4 (optimized for resource-restricted devices) of Desmet et al.(2007) architectural requirements, which we identified as the relevant ones for this thesis. So through our research and development of our prototype, we can conclude that these architectural requirements are the important requirements in developing an app-environment.

Working with MeeGo was not a challenge in itself, since it is based on Linux. Writing code and compiling for MeeGo was quite straightforward.

REFLECTION

The model that we have proposed in this thesis works in theory, and in practice in our application. This does not mean that it is totally fool-proof. Further research into each of the tiers focus area is needed for this model to provide a really solid framework for app-security.

What we have seen during the study is that there are many benefits with MeeGo when it comes to security issues. This is likely due to the fact that it is based on the Linux kernel. As presented in this paper, several tools with potential to aid a secure app environment on MeeGo exist as generic Linux applications. There is also an ongoing discussion regarding the various tools in the many Linux and security oriented forums, mailing lists and blogs. A lot of information on the subject remains to be collected and analyzed. Our research merely scratched the surface.

As shown in the MeeGo project wiki, access control is currently handled using SMACK. Time would not allow us to actually try its features in practice but it sure feels promising that SMACK is a default feature in the MeeGo platform. How it can be utilized in the design of an app environment remains to be seen, but the MAC concept taken care of by SMACK is used in other app environments, Apple iOS being one.

PROPOSITION FOR FURTHER RESEARCH:

During the project we came across the problem of how software is authenticated on the client side after being downloaded. Our contact person at CarComp expressed the importance of this functionality. Fully understanding this issue is not what this paper was about and time would not permit us to investigate the matter very closely. However, making sure that downloaded software is what one assumes it to be adds a fourth tier of protection to our proposed three tiers. It would also be a necessary step to take if further development of an app environment on the MeeGo platform is considered.

Guaranteeing software authenticity is not something new and has been discussed in many corners of the internet and by researchers. A lot of information is available on the topic. Something that came to our minds during the research process was that existing app environments, like Android Market and Apple's App Store share some fundamental characteristics with the package managers commonly found in GNU/Linux distributions (Aptitude, RPM, pacman, etc), namely that of distributing software. Recently, package managers even have nice graphical user interfaces (e.g. Ubuntu) and there is a resemblance to the app store clients when it comes to usage: search, find, download and install. A GNU/Linux package manager usually handles updates and dependencies between packages. Also, last but not least, software authentication is more or less handled automatically in these package managers.

To sum this up, our suggestion is that a future project our research effort would try to utilize a widely used package manager such as RPM or Aptitude and build an app solution based on it. Exactly what that needs to be added has to be figured out for the specific case. MeeGo IVI comes shipped with RPM installed so perhaps that is a good starting point.

This research effort could potentially benefit the free open source software (FOSS) community, which has to be considered a good thing for the general public, if the project would develop or modify the existing systems to fit the purpose of an app store solution. Further, the FOSS community could potentially benefit from the existence of such a solution as it would enable new ways of using FOSS commercially. The problem with making money from FOSS has been a common criticism but an app store for FOSS distribution might just be what could solve this problem. The general idea has already been adopted by Apple and the Mac App Store, which is an appstore for their laptops and desktop computers.

Another suggestion would be to look closer to mandatory access control, SMACK in particular, and how this can be used to solve issues regarding safety and reliability in an app environment. A potential study could compare different tools for MAC and evaluate how suitable they are for various kinds of systems. Several options exist and it might be of value to industry to read a comparison discussion them. Also, this could be interesting to the developers of the different software projects in order to get new ideas which could be used to improve their products.

REFERENCES:

Hevner, A.R., Chatterjee, S, 2010, *Design Research in Information Systems*, Integrated Series in Information Systems, Volume 22, Springer.

Prevelakis, J., Spinellis, D., 2001. Proceedings of the FREENIX, *2001 USENIX Annual Technical Conference*. Berkeley(CA, USA)

McGraw, G., 2004. Software Security, *IEEE Security & Privacy*, 2(2), pp.80-83.

Shroeder, S., 2010, Introduction to MeeGo, *IEEE Pervasive Computing*, 9(4), p.4

Desmet, L., Joosen, W., Massacci, F., Naliuka, K., Philippaerts, P., Piessens, F., Vanoverberghe, D., A 2007. Flexible Security Architecture to Support Third-party Applications on Mobile Devices, *CSAW'07 Proceedings of the 2007 ACM workshop on Computer security architecture.* New York(NY, USA)

Sommerville, I., 2007. *Software engineering*. 8th ed. Pearson Education Limited

Sharma, C., 2010, *Sizing up the global mobile apps market.*[online], Industry Study Commissioned by Getjar. Available at: < http://telecomcircle.com/wp-content/uploads/2009/05/Sizing_up_the_Global_Mobile_Apps_Market.pdf> [Accessed at 18 March 2011]

Saab automobile, 2011, *Saab IQon - infotainmentsystem genom öppen innovation.* Press release, 1 March 2011. Available at:<http://media.saab.com/sv/press-releases/2011-03-01/saab-iqon-infotainmentsystem-genom-ppen-innovation> [Accessed at 18 March 2011]

Dwivedi, H., Clark, C., Thiel, D., 2010. *Mobile application security*.MCGRAW-HILL

MEEGO, 2010, MEEGO WEBSITE, [ONLINE] AVAILABLE AT: <https://meego.com> [ACCESSED 9 MAY 2011]

Lua, 2011, *Lua Website,*[online] Available at: < http://www.lua.org/> [Accessed 10 May 2011]

Linux Vserver, 2009, *Secure chroot Barrier,* [online] Available at < http://linux-vserver.org/Secure_chroot_Barrier> [Accessed 10 May 2011]

Maemo, 2011, *Maemo website,* [online] Available at: <http://maemo.org/> [Accessed 10 May 2011]

Linux, 2006, *SELinux: Comprehensive security at the price of usability*, [online] Available at: <http://www.linux.com/learn/tutorials/305764-selinux-comprehensive-security-at-the-price-of-usability> [Accessed 10May 2011]

Edge, J., 2010, *The MeeGo security framework*, [online] Available at: <http://lwn.net/Articles/416771/> [Accessed 10 May 2011]

App Armor, 2011, *AppArmor wiki,* [online] Available at: <http://wiki.apparmor.net/index.php/Main_Page> [Accessed 10 May 2011]

Kernel Trap, 2006, *Abusing chroot*, [online] Available at: <http://kerneltrap.org/Linux/Abusing_chroothttp://lwn.net/Articles/416771/> [Accessed 10 May 2011]

Tomoyo Linux, 2011, *TOMOYO Linux on MeeGo 1.1,* [online] Available at: <http://tomoyo.sourceforge.jp/1.8/meego-x86.html.en> [Accessed 10 May 2011]

MeeGo Wiki, 2011, *Security/Architecture*, [online] Available at: <http://wiki.meego.com/Security/Architecture#Access_Control> [Accessed 10 May 2011]

Shaufler, C., 2008, *The Simplified Mandatory Access Control Kernel* [online] Available at: <http://schaufler-ca.com/data/SmackWhitePaper.pdf> [Accessed 10 May 2011]

APPENDIX A

Through this URL you can download a zip-file containing all code produced during the project. The archive does not include makefiles or anything helping you to compile the code, but this should not be too hard to manage if you have some experience with programming.

http://erikks.se/thesis/source.zip