FOX-IT
EXPERTS IN IT SECURITY

A Secure Access Control System
Glados Documentation

Final Report

Classification **PUBLIC**

Project ID	DOC-FINAL-REPORT-PUBLIC
Date	2011-08-10
Version	v1.1
Author	Jurre van Beek, Rutger Prins, Tristan Timmermans

Fox-IT BV
Olof Palmestraat 6, Delft
P.O. box 638, 2600 AP Delft
The Netherlands

Tel.: +31 (0)15 284 79 99
Fax: +31 (0)15 284 79 90
Email: fox@fox-it.com
Web: www.fox-it.com

ABN-AMRO
no. 55.46.97.041
Chamber of Commerce
Haaglanden no. 27301624

Fox Crypto BV

Olof Palmestraat 6
2616 LM Delft

P.O. box 638
2600 AP Delft The Netherlands

Phone: +31 (0)15 284 7999
Fax: +31 (0)15 284 7990
Email: fox@fox-it.com
Internet: www.fox-it.com

Table of Contents

i. Preface

To learn, and to fulfill the demands of our academic institution, we wrote this report on the design and implementation of Glados, a Secure Access Control System, and the process we went through to create it.

We are grateful to our mentors Adriaan de Jong and Christo Butcher from Fox-IT, and Andy Zaidman from the TU Delft for their awesome guidance during our 10 week internship at Fox-IT. We'd also like to extent a special thank you to Valve, for creating the video-game franchise Portal, which gave us the inspiration for the codename of this project.

If you want to grade our performance or have to work on this system, or are just interested in how to design a secure and maintainable access control system, you should continue reading!

Delft, 15-07-2011

Jurre van Beek
Rutger Prins
Tristan Timmermans

ii. Summary

We, three students from the TU Delft, have created a new Secure Access Control System, named Glados. The goal of this system is to limit the access to a building, and separate 'zones' in that building, to authorized persons only.

In 10 weeks, we created a prototype that can read an authentication token, and react with the permission from a central server. Each door in a building can be equipped with a small computing device and token readers (one for each side of the door). The device (called a 'Portal') communicates with the central server, where the information about people and their access rights are stored.

Our primary concerns were security and maintainability. Special care has been taken in identifying possible attack vectors and defending against those. This includes using high-end authentication chips, encryption of communication and centralizing sensitive data. Maintainability was achieved through test-driven development, using programming languages and frameworks well known to Fox-IT and a modular design.

Before the system can be used in practice, a lot of work still needs to be done:

- An audit system needs to be created for event monitoring.
- The current basic configuration interface needs to be refined.
- A small PCB board needs to be developed to connect the token reader (only prototypes exist)

iii. Glossary

text (reference)

A reference to an external file (See chapter 9) or appendix.

1-Wire

A bus protocol with one wire and a ground.

AI

Artificial Intelligence.

Big Endian

Most significant byte first. This is like writing numbers on paper.

Bit

1/8th byte.

Byte

8 bits.

Django

A python web framework.

Electric strike

Electric lock used on doors

Glados

Genetic Lifeform And Door Operating System, the codename for the entire project.

GPIO

General Purpose Input/Output

I^2C

A two wire bus protocol with additional ground.

LED

Light emitting diode. A fancy, shiny thing.

Little Endian

Least significant byte first. This is like with arithmetic operations on paper (right to left)

Open-Drain

Wire-OR GPIO. For a more detailed description you can use the internet to find examples.

Page

A memory page, usually the size of a memory block. The iButtons contain 32 byte pages.

Portal

The hardware and software used at the doors.

RTOS

Real time operation system. An operating system which has a level of consistency concerning the amount of time it takes to accept and complete an application's task.

Server

The central server, this includes the database and the server logic.

Serial

A serial interface, usually UART and/or RS232 compatible.

Token

A iButton device ('Druppel').

UART

Universal asynchronous receiver/transmitter, usually a serial port (see serial).

UI

User interface.

User story

Short description of what someone or something can or cannot do, in a single sentence.

Zone

A part of the building to which user have access or not. Zones do not overlap.

1. Introduction

In its essence, security is about controlling what people can and especially can not do. For people to do anything successfully they need information on how to do it and information about how the environment will respond. Controlling information should therefore be a primary goal in any security strategy.

A 'Secure Access Control System' can help with the most basic way to information security: Limiting physical access. It automates the checks on a person's identity and rights, which are required every time he or she wants to walk into the secured area.

This report shows what we, three Computer Science Bachelor students from the Delft University of Technology, have done at Fox-IT; how we planned, designed and implemented the new secure access control system 'Glados' (named after the AI that tries to prevent people from escaping the Aperture Science laboratories in the video-games Portal and Portal 2). The report serves two purposes: Giving our mentors an impression of how we performed, and providing a reference for developers who want to improve the product.

In this report we first give a short recap of the previous documents and reports we wrote during our project (2. Preliminary). This is followed by a description of the challenges associated with the project and an analysis thereof (3. Problem description and analysis). After we give an overview of the design of Glados and the design choices we made (4. Design). Then we discuss the planning we have made at the start of the project (5. Planning). This is followed by an overview of how the implementation process went (6. Implementation). After that we conclude the report (7. Conclusion) and give recommendations, of what work still can (or has) to be done (8. Recommendations). Last, but not least, we discuss how we look back at the entire in a personal evaluation (9. Personal project evaluation).

As a requirement of the TU Delft we need to publish this final report publicly. The report contains confidential information since it concerns a security system. The system might be used as basis for a future system by Fox-IT, and therefore we want to prohibit the public of getting highly detailed information about the structure, data, data flow and internal workings of the system. For this reason we have removed several parts from the public part of the report and placed them in appendix H: Confidential information. If a portion of information was left out intentionally we placed a reference to appendix H with a superscript section number. This appendix can be requested at Fox-IT if the necessity occurs and might be subject to a non-disclosure agreement and security screening. The grade and criticism we received was based on the original, fully integrated report.

2. Preliminary

This is the *final* report, implying that there were other reports. To assist under-informed readers and to refresh the memory of over-informed readers we summarize the most important parts of the previous reports here.

In this chapter we will summarize the following documents and reports (which have been added to the final report as appendices):

- Project description (Appendix A)
- Plan of action (Appendix D)
- Requirements Document (Appendix B)
  - User stories (functional requirements)
  - Non-functional requirements
- Research report (Appendix C)

2.1 Project description

Fox-IT offered the secure access control project as an internship assignment for students at university bachelor level. The duration of this project was estimated to be about 3 months for 3 students working together.

This project involves designing and implementing a proof of concept for a secure, networked access control system that offers a complete infrastructure for managing and monitoring access control within a high-security environment, such as an office or production facility.

The system consists of a central server that implements the core access control logic and ensures correct authorization decisions. It also offers a (web-based) user interface for administration of authorizations and real time system activity overviews, as well as audit information. Distributed (embedded) devices provide user authentication and access control (i.e. door opening) hardware, and communicate with the central server to determine a user's access rights.

This project includes the following tasks:

- Determine system requirements
- Design system architecture
- Determine communication protocols
- Develop a proof of concept implementation, containing the following components:
  - Central server's access control logic
  - Central server's web-based user interface
  - Embedded devices' access control logic

2.2 Plan of action

We used an agile development process, starting with a short research period. After which design and implementation will follow each other continuously (Just In Time design). (Automated) testing is an integral part of the development process. Documentation was kept to a minimum and was updated alongside the project.

We divided general project responsibilities:

- **Project management** : Keep track of the schedule and planning. (Done by Jurre)
- **Code quality assurance** : Ensure coding standards are adhered to and tests are run. (Done by Rutger)
- **Documentation assurance** : Ensure reports are complete and on time. (Done by Tristan)

2.2 Requirements document

2.2.1 User stories (functional requirements)

We did a number of interviews with users of Cerberus for our requirements elicitation. Most of the resulting user stories involved the web interface, with the exception of the obvious and most important story:

- 7.1: A user requests access at a door with her token.

We chose this story for our first development iteration, because it depends on nearly all system parts and so we could start working on those different parts concurrently.

The other user stories can be summarized as:

- Configuring zones and doors, users and tokens, etc.
- Managing user rights.
- Locating people in a building.
- Auditing events.
- Providing feedback at the doors (lights, sounds)

2.2.2 Non-functional requirements

For the non-functional requirements we took the categories from the ISO 9126 Product Quality standard and assigned priorities to those. The following list is thus in order of priority:

1. **Security**
   Security has our top-most priority. Implemented functionality that is not secure might as well not be implemented; this is a security system after all.
2. **Maintainability**
   Maintainability is of high priority for us, because the system will most likely not be finished in the time we have. Therefore, others need to be able to understand our code and adapt it and expand on it. Also, the reason this project was started in the first place was because the old system was too difficult to maintain.
3. **Reliability**
   Reliability is of high priority, because failure would have troublesome consequences: Either people cannot move through the building or security could be breached.
4. **Usability**
   Usability is (generally) of low priority, because only a small, constant group of users have to learn and understand the system. For the few use cases that are frequently used, usability is of high priority.
5. **Efficiency**
   Efficiency is of low priority. The problem of access control is in itself not remarkably data or processing intensive. If efficiency turns out to be a problem, the maintainable quality of the system should allow for optimizations in a later stage. We agree with Knuth that "premature optimization is the root of all evil" (Knuth74) .

6. **Portability**
   Portability is of low priority, because the system will initially be installed only on a single location. In the long term deployment elsewhere might be possible, but even then it's most likely that custom hardware will be installed to host Glados.

2.2.3 Quality Assurance

For the quality assurance we created a list of questions for each category. At any point in time, the answers to these questions are an indication of the state of the product and its quality.

Some example questions are:

- Security
  - Which user access points does the system have and how are these protected?
  - At which points can new data enter the system and how is this data verified?
  - At which points can data leave the system and how is this controlled?
  - What information could a thief/attacker get from a portal device?
  - etc..
- Maintainability
  - How many modules are there and how are they dependent on each other?
  - What is the test coverage percentage?
  - Do all use cases have a corresponding integration test?
  - etc..

2.3 Research Report

The research report focused on initial high-level architecture. Our highest priority and focus was the security architecture. We analyzed the security with the CIA method (confidentiality, integrity and availability) and identified possible threats. Choices were made about which technologies we would use to make up the architecture and provide technical considerations where choices are debatable.

Most of these preliminary design choices were also described in the Design chapter of the final report.

2.4 Available hardware

As already shown in the research report, we had a fixed choice in hardware. The hardware was selected so we had sufficient computing power available for the hardest tasks (i.e. AES encryption) and sufficient peripherals available through GPIO ports.

3 Problem description and analysis

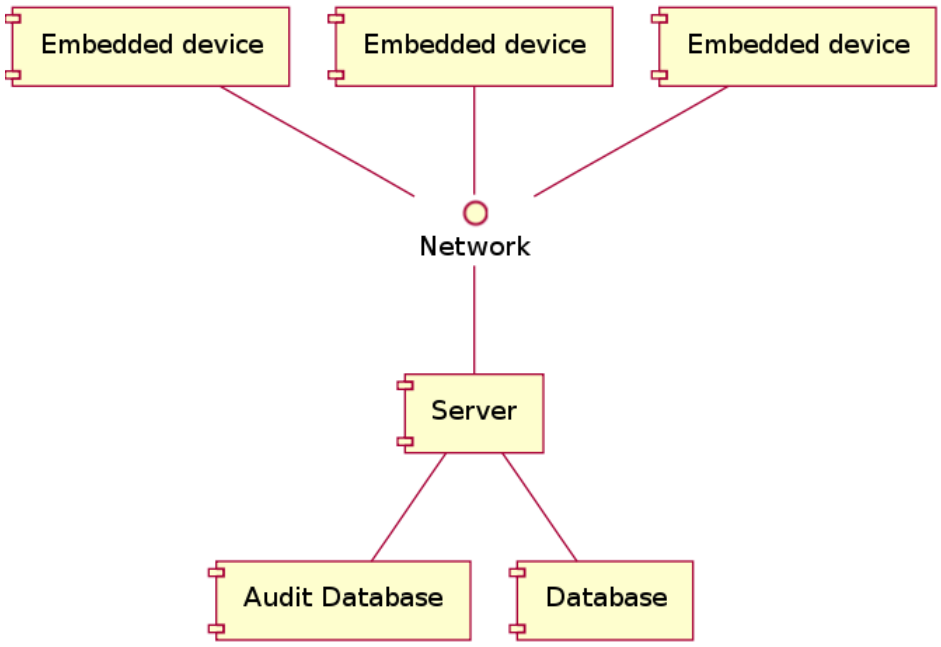
In this section we discuss what the project encompasses; What are the problems we need to solve in order to create a working prototype that meets the requirements? First, we will give a problem description, followed by an analysis.

3.1 Problem description

A access control system should be capable of dividing the building into different zones to which users can have access. This means the users should identify themselves when they enter or leave a zone to keep track of the user movement. This way a user can be checked to have access to the zone or not by allowing the user to pass through the door or be rejected. Because the user would be unable to go to another zone if the system failed, reliability should be high and the system must be robust.

Another advantage could be the tracing of people: If you need a specific colleague you can look up where this colleague is and go to him or call him at the right phone or place. To do this a system should be accessible from almost anywhere and have a responsive and easy to use UI.

Every door between two zones has an embedded device that controls an electric lock, reads tokens that employees use to authenticate themselves. The device uses a network connection to connect to a server. The server checks whether people are authorized to go through the door or not and communicates this to the embedded devices. All information needed for the server to operate is saved in a database. It also keeps track of all movement within the building in an audit database. The following diagram shows the basic setup of the system:



Since we were building the new system from scratch, there are a some technical challenges we had to overcome:

- Network communication should be as secure as possible, we don't want people to be able to eavesdrop or be able to grant themselves access to an embedded device, the server or a zone.
- There should be a robust protocol for network communication.

- The embedded devices need to be able to read the tokens employees use to authenticate themselves.
- What responsibilities do server and embedded devices have?

3.2 Analysis

To analyze the technical challenges we had to overcome, we wrote a research report at the start of the project. In this report we offered solutions for those challenges, which we summarize in this section. We also wrote a requirements document to give a clear view of the functionality the finished product should have, which we also discuss in this section.

3.2.1 Requirements

To develop a prototype that meets the requirements of Fox-IT, we first had to know what these requirements are.

We gathered the requirements by interviewing employees of Fox-IT who use different parts of the system and have different interests in using it. These interviews accumulated to a list of user stories, which we use as functional requirements. The interviews also led to a list of non-functional requirements.

The complete list of requirements can be found in the requirements document (User stories) (Non-func. requirements) .

3.2.2 Communication

As part of the research we did, we looked at ways to set-up a secure connection (Research Report p10) . We could use existing techniques discussed in the research report, which will probably result in a system with a secure form of communication with a good encryption scheme.

The server side should be capable of accepting multiple concurrent connections. There are multiple ways of doing this: e.g. listeners in different ports, synchronous/asynchronous connection handling. The server should also understand the (secure) communication from the portals and this can be easily done by a well documented security standard.

We also need to do our own communication with actual data. Since we needed a robust protocol, and also wanted some flexibility in case we want to add a new feature in the future, we decided to use a existing serialization framework. You can write your own messages, where you define how the data you want to send is structured and then compile it with the serializer. This way you don't have to write your own code to serialize data. Plus, if you define the messages in the right way, you can even extend the protocol in such a way that your application will keep working without having to alter it.

3.2.3 Token reader

We also looked (Research Report p9) at what technologies we need to be able to read the tokens Fox-IT uses for authentication. The token reader which we selected can be found in appendix H (3.2.3) .

3.2.4 Responsibilities of server and embedded devices

We assumed that there is enough computational power on the embedded devices to perform whatever task we want (there where bought with that in mind), then our main focus should lie on security when deciding what tasks should be done by the server and embedded devices. As part of the research we did, we looked at the confidentiality, integrity and availability of the system (Research Report p6) .

We know that the server will be in a secure room or zone (see glossary) , while the embedded devices could be in a less secure room or zone. So we must make sure that it is impossible to retrieve any sensitive data. The easiest way to do this, is to store no sensitive data on the embedded devices unless it's absolutely necessary.

This made dividing tasks between the server and embedded devices much easier, only tasks which absolutely needs to be done on the embedded devices should be done there (e.g. reading tokens and sending the token data to the server). All other tasks (e.g. authenticating the token data and deciding if someone can go through a door or not) should be done by the server.

4 Design

4.1 Introduction

In this chapter we describe how the Glados Secure Access Control System is built up and why we designed it this way. We think that a good high-level design is the most important factor in maintainability, so that is what we focus on in this chapter. Of course, we also consider security aspects and we will discuss those aspects at length in the design of the authentication process, amongst others.

4.2 System view

The full design of Glados on a system level can be found in appendix H (4.2) .

Functionally, the system can be divided into three parts.

1. **Door control system** The door control system handles access requests from users at a door. It was developed as two separate parts, the server and the portal, which share a messaging protocol.
2. **Web interface** The interface lets users find the location of other users, and provides configuration options for administrators.
3. **Audit system**

4.3 Data model

The data model represents objects from the real world that are directly involved in access control, i.e. users, tokens, doors, zones, etc. This model is used by both the door control system and the web interface.

It is important to note that this is not a SQL database schema. Instead, the data model presented in appendix H (4.3) is a class diagram. We use an object-relational mapping library (ORM) to manage the database schema for us. This library takes a collection of class definitions and constructs all necessary tables, including those for one-to-many or many-to-many relations. The ORM library is also used for all CRUD operations (Create, Read, Update, Delete), thus relieving us from writing cumbersome SQL queries and gluing the results to objects.

4.4 Glados server

The Glados server is the central part of the door control system, it services requests from Portals which are situated at the doors. The related code is contained in the `server.glados` package, including the data model. This data model is also used by the web interface (the `server.web` package). `server.glados` can be found in appendix H (4.4) .

4.4.1 Overall design

Since maintainability was our highest priority at this point, extra care was taken to lower dependencies:

- There are no circular dependencies between modules (i.e. they form a DAG).
- Translation from the messaging protocol to commands is contained within a single module. This way the connection management module can be completely purpose-agnostic; only having to call a handler function with incoming data. Also, the verifier needs no knowledge of the messaging protocol.
- Knowledge of the database model is limited to a single module.
- The server is stateless, apart from the database and the connection management.

4.4.2 Communications module

At the door, we require a low response time. Users presenting their token should get a very quick response. We don't want the Portal and server to shake hands and establish a new connection each time. Therefore the server has to maintain multiple concurrent connections.

We can achieve concurrency in several ways:

1. Multithreading: A thread for each connection.
2. Multiprocessing: A process for each connection.
3. Asynchronous event handling: Keep a list of connections and constantly check it for incoming events.

The biggest argument against multiple threads or processes is the complexity in managing shared memory. However, because the server is stateless this is not an issue. If this were to change in the future though, it would imply a high burden on maintainability.

Multiprocessing enables the server to run on different machines, which would make the system more scalable in the processing power dimension. However, we expect the server to be bound on network I/O, since no special memory or calculation tasks take place at the server.

Asynchronous event handling offers the advantages of multithreading, without actually forking threads. This works with the 'reactor' pattern: All possible events are stored in a list, the reactor iterates over this list and when it sees an event has fired it will execute any functions waiting for that event (this is called a 'callback'). Programming with callback functions adds a layer of complexity which is an argument against asynchronous event handling.

One difference in this complexity lies in its localization. Synchronizing resources over threads is something that might involve any module (e.g. a verifier might need to start keeping track of request sequence numbers). The problem of concurrent connection handling belongs to the connection manager and this should not spill over. This would lower reuse and maintainability considerably.

Asynchronous event handling localizes complexity. It also localizes benefits. If, because of performance issues, database access should also be done concurrently, then asynchronous event handling needs to be applied there seperately. Which isn't really a downside.

The definite choice for the asynchronous networking setup can be found in appendix H [(4.4.2)]

4.4.3 `MessageHandler` class

The `MessageHandler` forms the glue code between the messages and the verifier; message fields are directly mapped to functions and their parameters in the verifier. The only other purpose `MessageHandler` fulfills is exception handling; any exception during message handling results in an generic error message which is sent to the calling Portal.

4.4.4 Verifier

The verifier provides the commands that the Portals ultimately call. In this sense, the `MessageHandler` can be seen as a RPC layer. The verifier uses the ORM to communicate with the database. The ORM is the object-relational mapping library.

4.4.5 `hashing` module

The `hashing` modules holds the functions needed for the server-side calculation of hashed used by the portals.

4.5 Web interface

The web interface is used for configuring the door control system (see Admin interface), and for looking up in what zone users currently reside (see Overview page). The web interface uses the a web framework.

4.5.1 The rationale for the choice of the web framework

Details can be found in appendix H (4.5.1) .

The framework was chosen because Fox-IT has extensive experience with it.

There are many Python web frameworks, but the three most notable are:

- Django
- TurboGears?
- web2py

All frameworks have good security measures against things like cross-site scripting, SQL injection, file execution, etc. And they provide ORM libraries that talk with wide selections of SQL databases and object oriented databases.

Django has extensive and accurate documentation, including well-written tutorials which significantly lower the learning curve (which is important for such a short term project). Django also has many additional, tried-and-tested plug-ins.

TurboGears provides a more powerful ORM (SQLAlchemy) than Django, a more powerful template engine and there are claims of better performance. However, documentation seems to be occasionally inadequate or erroneous.

Web2py is a serious contender because of its easy learning curve. It was originally created for educational purposes. It is also more 'convention over configuration' oriented (which is a plus).

4.5.2 Admin interface

A detailed description of the admin interface can be found in appendix H (4.5.2) .

4.5.3 Overview page

The overview presents users per zone, with a search for specific zones or users. It also includes a refresh option to monitor changes. This page is accessible to all users, therefore we expect that this will be the most-used feature of the web interface

4.5.3.1 Mockup

We made a list of features the overview page should contain and created a mockup (using a mockup tool called 'Balsemiq'). The features are:

- The page title.
- A login box.
- A list of people per zone.
- A search field.
- A building map with zone names (optional).

A mockup picture can be seen in appendix H (4.5.3.1) .

Mockup rationale

- Keeping things simple.
- Search does not need any extra data from the server, so searching through the presented list can be done with a quick javascript function which doesn't require extra page loads.
- Search and list are in the center, since this is the primary function of this page.
- Upper right is a common position for log in forms.
- Building map provides additional orientation, as zone names could be uninformative. It should be a static image, for simplicity. The map is placed bottom left, because it's the least important thing on the page.

4.6 Authentication design

Since the main feature of the system is validation of user rights and monitoring, the authentication process is vital and must be both secure and robust. In the research report information on the global security and the possible threats can be found in sections 2.1, 2.2 and 2.3 (Research rep p5-7) . This part describes the process from a token entering until a valid or invalid response from the server to eventually the user and will leave the internal functions "as is" and focus on data transmission and calculations.

Detailed description of the tokens used and the way the authentication process takes place can be found in appendix H (4.6 - 4.6.2) .

4.6.1 Overview

The complete flow of data and extra information can be found in appendix H (4.6.1) .

4.6.2 Improving security

Security improvements are shown in appendix H (4.6.2) .

4.7 Portal design

The portal handles requests from the user and reports to the server what is happening and requests the server for verification of a token. The portal is set up as an interface to different components via kernel modules. These modules communicate via predefined input/output characteristics (see Functional spec) with the hardware.

The layout of the components and detailed descriptions can be found in appendix H (4.7) .

The portal logic handles the input and responds with requests to the server via network or by displaying a message to the user via the interfaces. Possible messages are a LED to show green when the user has access and red when the user has none. Otherwise a buzzer has been used for alarming the user and a lock can be unlocked (or locked again).

The portal does not respond to requests from the server. This means the portal must request all needed data and initiate all communication by itself. This is chosen to keep the server unaware of the portals. And therefore the portal has to request a status message every now and then (in the order of seconds).

The 'class' diagram can be found in appendix H (4.7) .

4.7.1 Portal logic

Functionality within the portal is handled by the portal logic which sends requests to hardware or the server via the interfaces. This logic is implemented as a finite state machine. Since a state machine has the possibility to end up in states via transitions which are not expected or end up in a state which has not been documented or meant to happen, we need to test for the so called 'hidden states' and 'hidden transitions'. Since we test for all 'hidden transitions' (which is

a finite amount) we can be sure the portal will not switch to an undesired state. This does not fix hardware failure, however and this might still occur but is manageable by resetting the portal after a hidden transition occurred.

The state machine can be found in appendix H (4.7.1) .

4.7.2 Interfaces

The interfaces are designed to accept a command per function. They should handle the request of the portal and translate this to any form of communication required by the kernel modules or the network. They only report a success or a specific failure to the portal logic and handle all setup/close actions needed. The interfaces can communicate with each other and should function as one library and not several separate libraries. While this is a constraint on maintainability, the interfaces are inexplicably linked as the token interface and the connection interface need to be aware of each other since they must pass data.

The interfaces communicate with the following:

- The hardware
- The network

Since most of the hard work is done by the kernel modules, the interfaces only need to translate the requests of the portal to decent function calls [Functional Spec. (p 2-5)] to the kernel modules. The interfaces are unaware of the underlying hardware, though the token interface has such specific functions that they are effectively need to be a one to one map of the functions in the token. To be independent of the portal logic the interfaces need to handle all file input/output (e.g. the kernel modules).

4.7.3 Kernel modules

There is an option to use the Linux sysfs [Sysfs] but this is not a good idea because of several reasons: For full access which the GPIO would need we should run as root and we want to avoid setting the entire sysfs system to normal user control. Another problem is speed; for the token and buzzer it is simply too slow.

Additional information about the hardware and kernel modules can be found in appendix H (4.7.3) .

Since Linux is not a real-time operating system (RTOS) and the realtime-extension for Linux is not available for our kernel, we will have to use spinlocks during reading/writing the token. As we do not know the time a context switch will take and cannot be certain the function will sleep for a specific amount of time, unless we busy wait and spinlock the system. This does slow down the system but not to much extent (max. ~2,5 ms).

5 Planning

During the first week of the project, we worked on a plan of action, which included the project approach and planning (Plan of Action) . The goal of writing a project approach and planning was to give us, and our mentors, a guide to follow during the project and is based on how we envisioned that the project should unfold. It also provided a good opportunity to reflect at the end of the project.

In this section we will first discuss the approach we took to work on the project. Finally, we will take a look at our planning in the form of a time schedule.

5.1 Approach

As discussed in the research report (Research Report) , we took an agile approach to this project. For us this meant that, after we finished the research report, we did some initial requirements and architectural design envisioning. The requirements were based on interviews we held with different employees of Fox-IT and the architectural design was based on these requirements and on the high-level design you can find in the research report.

After that, we started the development process using (preferably small) development iterations (see 6.1 Development iterations). Each development iteration included the following steps:

- Choose a little bit of functionality
- Do a little bit of model storming
- Do test driven development

The general idea behind this, was that we would finish a little bit of new functionality every week or every few days. After we finished an iteration, we started a new one which build functionality on top of the previous iteration. This way we gradually expanded the functionality and constantly evaluated the design we had made so far, instead of creating the entire design beforehand, then develop and see if everything works correctly at the end.

5.2 Time schedule

Below you can find the time schedule we have made, which you can also find in the plan of action (Plan of Action) .

Week	Dates	Planned Milestones
1	April 26 - April 29	- Plan of action
2	May 2 - May 6	- Research report - User stories
3	May 19 - May 20	- Initial requirements - Initial architectural envisioning
4	May 23 - May 27	- Start of first development iteration
5	May 30 - June 3	
6	June 6 - June 10	
7	June 13 - June 17	- Halfway code review - First draft of final report

8	June 20 - June 24	
9	June 27 - July 1	
10	July 4 - July 8	- Start of last development iteration
11	July 11 - July 15	- Intensive testing - User manual - Final report - Final code

6 Implementation

As can be seen in the planning (Plan of Action p 7) we planned to do several development iterations. These iterations should add functionality or do code revisions to improve previously written code. After each iteration there should be a working product, though it might not be in presentable order. While the target was to do an iteration every week, some took far longer and some shorter.

Iterations consisted of one or more user stories, and can be divided into three major parts: portal code (C), network code (C & Python) and server code (Python). These three major components were divided evenly among us:

- Portal code: Tristan Timmermans
- Network code: Jurre van Beek
- Server code: Rutger Prins

While this gave the opportunity to dive into your own code and leave the rest as it is, it did present a clear outline about what you needed to do and with whom you needed to talk. Since the project only has three participants this was decent option despite the problem which might arise when some workload is more at one specific part of the code-base. To solve this we did hand over some coding to others. Generally the workload was evenly divided.

In this chapter we will first discuss how we set-up the test and build environments. After that we will discuss the three development iterations we worked on. Finally, we will reflect on the implementation process.

6.1 Test and build set-up

Before we could actually start developing, we needed set-up and configure some tools for testing and building the implementation. Because Python doesn't need to be compiled, we don't need a build tool for it. We did need a build tool for C however, to smooth the build process.

At the start of the project we planned to use the following development tools and libraries:

- Setup of build environment:
  - Using CMake
- Setup testing:
  - For Python: Using existing frameworks.

6.1.1 Build environment

Server

The server environment consists of Python, the web framework and the database. A question we faced was: How do we separate the code for the web interface and the door control system (Glados)?

The web interface is hosted in a site directory, and the data model is also in that directory because it relies on the ORM. The web interface and the Glados door control both use that data model. Having Glados in a directory separate from the site, created import difficulties and it also made unified test running nigh impossible.

In conclusion, separating the door control system and the web interface would make a spurious division; they both rely on the web framework for testing and database access.

Portal

Since we are testing, running and compiling on both the portal and the x86-64 machines we used every day, we could not use the same build setup. To avoid strange compiling problems, different versions of make and other environment problems, we decided to use a universal build system: CMake. CMake files can be found in almost every C code directory and are recursive with the CMake file in the root directory. CMake automatically finds the correct compiler, sets the options correctly for the build environment and therefore we did not have to double up for every machine or instruction set. The kernel modules are board specific and should not be compiled on our x86-64 machines. You can cross-compile them but, since it's easier, we decided to build them on the portal.

We completed the environment setup quite quickly. The configuration in appendix F shows the necessary steps to create the basic environment. CMake files became more complex as the code-base became larger and the complexity increased. To simplify the CMake files, we first used a single large file. After two weeks this became large and cumbersome and we changed to a directory based system as can be seen in the code-base.

6.1.2 Test environment

Our testing goals are:

- Unit test for critical or complex functions.
- 100% coverage for functional tests per module.
- An integration test for the portals and server connection (currently done manually).

Server

Details about the and frameworks can be found in appendix H (6.1.2) .

Portal

At first we planned to use a existing test framework for the C code. Because the portal has the ability to compile code on site we do not have to cross-compile which saves time. The problem is that our original framework is mainly written in C++ and compiling it with our own code created severe memory issues. To avoid problems we switched to a small, self written, test framework. This can be seen by the preprocessor macro functions in the test directory. Using CTest we validated the tests during compiling/building. CTest is integrated with CMake, so tests can be compiled automatically and then can be run by simply running `make test` or `CTest` on any machine. The tests return either 'Success' (0) or 'Fail(1)

For the portal we wrote functional tests for every interface. This represents a way of unit testing but due to lack of the physical board or the token while testing some tests will have to be run twice: Both with and without a token present or on and off the board. Especially (and obviously) the token interface requires a token present to complete its test and should be run manually on the board. In the third iteration this has been fixed by specific identifiers which execute only common code on all devices and specific code only on the board.

In the third iteration automated tests for kernel modules have been made to be run when compiled on the boards.

6.2 First development iteration

All details about the development iterations can be found in appendix H (6.2-6.4) .

6.2.2 Time window

Planned:	23-05	until	10-06 (3 weeks)
Actual:	23-05	until	24-06 (Network, Server) and 28-06 (Portal) (~6 weeks)

6.3 Second development iteration

6.3.2 Time window

Planned:	27-06	until	01-07 (1 week)
Actual:	27-06	until	01-07 (1 week)

6.4 Third development iteration

6.4.2 Time window

Planned:	04-07	until	08-07 (1 week)
Actual:	04-07	until	08-07 (1 week)

6.5 Realization of Planning

Below you can find how the planned milestones stack up to the realized milestones. As you can see the first development iteration took more than five weeks to complete, instead of few days or a week.

Firstly, this is because we chose to include most of the functionality in the first develop iteration (see 6.1.1 First iteration). We did this to have a platform to build on with iterations that have only a little bit of functionality and that can be finished within a week. Secondly, this is because we still needed to do some reading on the tools and methods we used and to settle in the new environment we were working in.

Because the first iteration took longer than planned, we only had time to do three development iterations in total, but we did manage to develop a working prototype with enough functionality with those three iterations.

Although we did continuously document what we were working on via trac [TracWiki] , we started working on the final report a little later than planned. So instead of having the fist draft of the final report finished in week 7, we had the first draft ready in week 10 and we actually spent that entire week working on the final report.

All in all, we think we managed to realize most of the things we planned on time. More importantly, the project approach and planning gave us a good guideline to follow during the project. We also think we made the right choice to include a lot of functionality in the first development iteration. Although, we should have thought about that when we wrote the project approach and planning.

We also could have started writing the final report a week earlier. I think it took more time to write it than we initially thought it would, although we managed to finish it in time. If we had started earlier, this would have meant that the draft we sent to Adriaan and Andy, would be more complete. This, in turn, would have meant we would have gotten more feedback we could have used to make the final report better.

Week	Dates	Planned Milestones	Realised Milestones
1	April 26 - April 29	- Plan of action	- Plan of action

2	May 2 - May 6	- Research report - User stories	- Research report - User stories - Initial requirements
3	May 19 - May 20	- Initial requirements - Initial architectural envisioning	- Initial architectural envisioning
4	May 23 - May 27	- Start of first development iteration	- Start of first development iteration
5	May 30 - June 3		
6	June 6 - June 10		
7	June 13 - June 17	- Halfway code review - First draft of final report	- Halfway code review
8	June 20 - June 24		- End of first development iteration - Start of second development iteration
9	June 27 - July 1		- End of second development iteration - Start of last development iteration
10	July 4 - July 8	- Start of last development iteration	- End of last development iteration - First draft of final report
11	July 11 - July 15	- Intensive testing - User manual - Final report - Final code	- Intensive testing - User manual - Final report - Final code

7 Conclusion

In the project description we summed up the goals of the project:

- Determine system requirements
- Design system architecture
- Determine communication protocols
- Develop a proof of concept implementation, containing the following components:
  - Central server's access control logic
  - Central server's web-based user interface
  - Embedded devices' access control logic

The Research report and the requirement documentation (User stories) (Non-func. Requirements) fulfill the first point and part of the second. These documents were guidelines in the development and were made to aid in selecting the choices made in the different iterations. While this looks more like a waterfall model of development and not the agile way we wanted, it did give a stable base to continue upon which is a plus. This 'hybrid' model of development is in our opinion a normal course of action while most agile developers would say it is not. It provides a base to work on and show you what big problems might show up.

Most of the real design had not been done in the research report, and we did those during the iterations. This worked out pretty well and we had no major problems in designing the components (e.g. the database, authentication and communication). As can be seen in this document, the system design is mostly complete (missing the audit database) and can be used in a real product.

Due to the use of standard components for secure communication we had little difficulty to set up communication: The server and portal were developed separately but since they used the same protocols they communicated out of the box. The decision to use those tools sped up the project significantly.

While the project description, develop a 'proof of concept' as implementation goal, it was clear from the beginning that the design and implementation should be extendible and deploy-able. Therefore, as a proof of concept is usually a tech-demonstration, we also had to keep in mind that all code could be used in a final product.

The server and portal (embedded device) logic have most of the functionality available and the functions which are not active have either their implementation on one side (i.e. portal status messages) or simply lack function calls from the main routine due to missing hardware at time of writing.

While the web-based user interface is not finished, the basis is in place with the web framework, the database and examples like the overview of users. This should make the development of a decent interface easy enough to be done by any employee with sufficient Python knowledge. Since the missing functionality can be implemented without changing the database format or altering code in either the portal logic or server logic, a integrate knowledge of the entire system is not needed. This should make developing the rest more easy and independent of our continuation with Fox-IT.

8 Recommendations

In this chapter we will offer some recommendations. Some of the points we discuss are about work we weren't able to do because of the time constraint of the project. Others points are more about what we think would be good additions to the system. Most details can be found in appendix H [8.1 - 8.3] .

8.1 Server

Further develop the admin interface

The default administration interface needs to be improved before the system can be deployed and used. The user rights needs to be more fine-grained. E.g. letting a someone only control her own fields. Second, usability needs to be increased by explaining the meaning of fields, or hiding fields that users should never set or change.

Implementing auditing

The auditing system will need to hook into the current code. Verification results, opening doors and other events need to be registered.

It might be prudent to then also create a `SerializationHandler` so messages are not unnecessarily serialized and deserialized in every handler.

8.2 Portal

Several improvements need to be made:

- Configuration files and reading them.
- Remote configuration by deploying configuration files.
- Remote updates of the portal logic.
- Improve auditing.
- Improve maintenance mode.
- Interrupt usage for detecting tokens.

Configuration

At the moment most of the settings are hard-coded into the portal logic, interfaces and kernel modules. This prevents a simple roll-out of all components since it cannot be changed without recompiling. While all the portals have the capability to compile their own code, you might want to change things during operation. On the other hand, recompiling means root access is needed and this is another step to take if you want to abuse the portal.

If a means of configuration with config files is adopted, some way of distributing the files is needed. We advice to keep the network communication one-side oriented and let only the portal initiate connections. This would be possible with a periodic check. Otherwise logging in to the portal via might be a option.

Updates to portal logic

While kernel modules require more access, a way (i.e the configuration files) could be adopted for replacing the program/portal logic. Since the software should always keep at least the basic functionality, you can do this by letting the

portal periodically check for updates since no functionality should be lost with an upgrade (the system keeps working with minimum capabilities). Future improvements could therefore be distributed.

Improve auditing

Since no error messages are send and the are stored locally, it might be a good idea to send the errors to the server for logging and monitoring. This does however give a boost in traffic when a power outage ends and the systems boot or when the network was down and went up again. This sould be considered when building such a auditing/loggin system.

Improve maintenance mode

The current maintenance mode is a separate file which can send data to the token and turn on/off the LED's. We can try to make this possible remotely, but since only the administrator will be doing this we can easily keep it a command line option. There should however be a maintenance function for all the commands except opening the door. The maintenance function should also audit every change or action which it does not do at the moment.

Interrupt usage for token detection

At the moment the portal checks quite a lot of times per second for a token. When this is the case, the line will change and we can detect this with a trigger on the falling edge of the signal. This can fire an interrupt to the kernel driver which can handle the interrupt and alert the user space portal logic. This way we do not need lengthy and inefficient spinlocks to check the presence of a token with a reset signal and can thus save time and cpu usage.

8.3 Hardware

Another improvement might be the usage of a timer output for the buzzer which stops much interrupts being fired. Currently this is not done since the timer output at which the buzzer is connected is not accessible due to the Linux kernel blocking any access (by simply crashing). There are however sufficient timers and external pins available to do so.

Future improvements might be the use of a smaller board with just a Ethernet link (chips, power control and connector), a connector for the I/O ports, 5v to 3,3v step-down (or linear) regulators and some other minor components. This removes optional interference from other components and give the option to remove several modules from the Linux kernel. It might also lower costs for large scale deployment. On the other hand you do need a capable electrical engineer to design the peripherals and find a default board at a supplier.

9 Personal project evaluation

At the end of a project like this, it's always good to look back at what was achieved and how it was achieved. Even if things didn't go as planned, if in retrospect you can say what went wrong, you can prevent that from happening the next time you work on a project. In this chapter we will give a personal evaluation of how we experienced the entire process, one for each member of the group.

9.1 Jurre van Beek

Doing a project like this is always a challenge one way or the other. Most of the time the first challenge is to effectively work together, but this wasn't really the case for us. I think we managed to work together and communicate very effectively. Of course there were some discussions here and there, but you need that in order to do a successful project and we always kept it respectful and to the point. Meetings were concise, dividing tasks went very natural and we were all on the same page for at least 99% of the time.

The most challenging part at the start of any project is making a planning, especially when you are working on something you've never done before. On the one hand I do feel that planning is very important, but on the other hand it's so difficult to predict how long something will take that planning in some cases looks a bit arbitrary. That said, I think compared to Rutger and Tristan I'm a little bit more of a planner. This means that there is at least one person keeping an eye on the planning, which is probably more than enough for a project like this.

Then there is of course the issue of too much to do and not enough time to do it. This means that choices have to be made. What do we do first, what will we do at a later time or what are we not going to do at all? One of the choices we made, is to not implement auditing. This is one of the major requirements for the secure access control system, but we felt that we didn't have time to implement that along with the things we did implement. Besides, we have designed the entire system to be extendible, so implementing and integrating that functionality at a later time shouldn't be much of a problem. I think we made the right choices during the project to do certain things and not do other things.

I enjoyed working on this project very much, we had a good time working at Fox-IT. I think we made the right choices along the way and did a good job. I've learned a lot about the frameworks we used. I also managed to become a lot more fluent programming in C. So I think I learned a lot that could be very useful in future projects

9.2 Rutger Prins

The challenge in such a project for me is learning what is the most effective way to work. How can we get done the most and honestly learn the most in these few weeks? Software development is a creative, chaotic and unpredictable activity, and managing this process is the hardest part.

My view on managing the development process the right way, is that control should remain short term and reactive. Instead of the military general that draws up the grand strategy, managing should be like the helicopter scout that keeps a constant overview and who can point into the right direction at any time. If motivation and priorities are well maintained during the project, our productivity is the highest and the most effective, regardless of planning.

I'd say that our level of professionalism was high. Our team spirit was great and the discussions were focused. Then again, we are not three average Bachelor students; we all have quite a bit of programming experience, and our age helps too. ;)

What I've learned (but also already knew) is that combining development and process management is hard. Every new functionality or technology you are implementing is both actual progress, and research in the balance of time needed versus value gained. In other words, to know if something is worth the effort, you have to do some of the effort. Doing that effort is also progress and investment, so at what point does one say: "I've spent this amount of effort, but from what I learned, spending more effort is not worth it."?

As a case in point, I encountered difficulties while working on the testcases for our communications module. The implementation of those testcases turned out to be complex, especially when compared to the very simple code it was testing. However, testing was of high priority for us and I did not know how much time it would take me to finish those tests. This is a problem inherent to programming; the one that makes planning so impossible.

I've learned several new frameworks, languages and a whole caboodle of smaller packages. Python is a great language. I find its strongest points are its idiomaticity, its terseness, the thought that has gone into every detail via the Python Enhancement Proposals, the available libraries, and the REPL (Read-Eval-Print-Loop) with its live documentation.

Ultimately, one could not have expected more from this project. We have a working demo, written copious amounts of reports and achieved a high code coverage. In 10 weeks, limited to business hours, without previous experience in secure access control systems! I think its time for beer.

9.3 Tristan Timmermans

The project looked quite big: Far more than we could do in roughly three months time. There was too much to make and lots of thing to take into account which made the initial part, the research report, somewhat longer and more complicated than strictly necessary. The fact that we all had other events besides the project, like the business-tour of the study association, did not help time wise.

This is directly to the main problem (as always): time. To make sure we could do as much as possible we decided to do the 'global' thinking together, and split the coding and code design in three parts. This is in my opinion the best solution: Rutger had more experience with template engines, web development and high level languages. My experience was with C and embedded platforms and Jurre did the interconnect between the two: the network layer. This might look like putting everybody in a specific corner and letting them mess around, but we were still dependent on each other. The fact that integrating the different parts took about one week while we were still doing other things in the sidelines, gives me the idea this option worked pretty well.

The biggest problem with the approach we took was the possibility to 'submarine'. Sometimes we did not now well enough what the progress of the other two was. Since we were only with three this could be solved quite easily by looking at the other persons screen but in larger groups this would not have worked. More precise assignments and more control would be needed.

The work load, which was determined by the three-way split in code, was in my opinion well divided: Some things required more research, like the secure communications, while others just required more code, like the other programs written in C. At the time when documents needed to be produced, I sometimes went on developing code for the token reader while the others went on writing documentation or reports. All in all the 'time' problem was expected and I think we all did a sufficient amount of work in the limited time we had.

During the development of the entire system I learned a lot about security, and especially about the 'not so security' of different approaches. The main problem usually is the human factor, since losing a key or failing to use proper passwords is a human problem. It can though be attempted to be avoided by guidelines. While I had plenty of experience with using embedded platforms, my experience with Linux as a embedded operating system was limited. This turned out to be a blessing and a curse: The ability to run Linux provided easy access and the ability to compile the

code on the device, it also provided the problem with Linux not being a real time operation system. When this problem was overcome it was easy to use and the learning process of writing more Linux code was welcome.

One disadvantage (or advantage) which I have not mentioned yet is the following: We could not take our work home. This provided the problem that we could not 'work late' and finish a piece by working really hard, but on the other hand we had to plan ahead and I think this meant we had to work more efficient. This idea of not being in control of our working hours provided a nice challenge and I think we did pretty well concerning this.

My final opinion is that we did quite well, despite limited time. We had a challenge and too much to do by ourselves and we took the right choices.

10. References

Since many references are for specific techniques used, the references are placed in appendix H [10] .

Appendici

Appendix A

Project description

A description from Fox-It about the project. We received this as our 'contract'.

Appendix B

Requirements documents

The requirements, both functional and non-functional. The functional requirements are effectively the user stories.

Appendix C

Research report

The research report. This report has been handed in on the 24 of May 2011. The research report contains all research done before the project.

Appendix D

Plan of Action

The plan of action as handed in in the first week of the project.

Appendix E

Functional Specifications

The functional specifications of several parts of the systems.

Appendix F

Configuration

The configuration for the server and portal before you can use them.

Appendix G

Token specification

Appendix H

Confidential Information