

Introduction

This report outlines the design and implementation of a social networking website named RaptorDating. Its main purpose is to securely allow users to socialize and share their personal information with each other. The system is designed with security and privacy as the primary design priorities, with relaxed security where necessary to allow for usability.

We begin by outlining our business goals for this product, before exploring the possible business and technical risks. We then continue by defining our architecture strategy, after which we define functional and non-functional requirements for the system. Afterwards, we dive into the architecture and implementation of the solution in terms of security measures and steps taken to fulfill said design principles and requirements. Finally, we test and evaluate the solution as a whole in order to provide a basis for a final conclusion.

Business Goals and Risks

As a group of master's students doing a project for a course, our situation is different from one a regular business would find itself in. We are not interested in generating revenue, and our only investment is time. Because of this, our business goals are also different from the goals of a typical business. Instead of focusing on revenue or user retention, our goals are to use what we have learned in the course to create a system and a report that lives up to (or exceeds) the expectations of the examiners. With this as our context, we have devised the goals, risks and mitigation strategies presented in this section.

Business Goals

- BG1: To create a system that lives up to the requirements specified in the project description.
- BG2: To create a secure system, that can withstand attacks from the other groups.
- BG3: To live up to the Service Level Agreement (SLA) of 95% uptime.
- BG4: To learn how to design and implement secure systems.
- BG5: To obtain an acceptable level of usability in spite of necessary security measures.
- BG6: To strengthen our project by successfully attacking other groups.
- BG7: To meet the deadline for project hand-in.

Risks

We divide the risks to the project in two: business risks and technical risks. In the following sections, we cover only the risks deemed most important. A complete list of ranked business and technical risks can be found in Appendix A. To rate the severity of each risk, we have adapted the impact definitions of McGraw¹ to suit our project as follows:

High: Enough negative impact to cause us to fail the exam.

Medium: Enough negative impact to significantly lower our grade, but still pass.

Low: Enough negative impact to prevent us from reaching the highest grade, but not enough

¹ Gary McGraw. *Software Security: Building Security In*. Addison-Wesley, 2006.

to have significant impact on our grade. The impact of several of these occurring would be a significantly lower grade.

Business Risks

Business risks cover any risks that threaten the business, but are not directly related to any technicalities of the implementation. In this section, we will address the most important business risks from the full analysis, in descending order of significance to our project: Lack of time (BR1), servers becoming unavailable (BR5) and opposing groups successfully attacking us (BR3). The remainder of our business goals are not irrelevant, but have a likelihood or impact too low to be mentioned in this report.

BR1: Lack of time

A chronic lack of time could stop us from fulfilling the business goals we have listed, which could cause us to fail the exam. This gives it an impact rating of **High**. However, given that we have controls in place to mitigate this, its likelihood is rated as **Low**.

Mitigation strategy

To mitigate this risk, we should arrange weekly working sessions as well as have intermediate deadlines. As our final deadline is strict, there is no mitigation strategy should we run out of time before the project is completed.

BR5: Servers become unavailable

If the servers become unavailable, fulfilling our SLA requirement of 95% uptime might not be possible. While not critical to passing the exam, it might still have an impact on our grade, earning it an impact rating of **Medium**. Given the realities of software development, and the possibility of attacks from other groups, we give this risk a **Medium** likelihood.

Mitigation strategy

There are two primary ways in which the server could become unavailable: Attacks from other groups, and software or configuration errors leading to unavailability. This mitigation strategy will cover both causes. To protect against attacks, we should design a secure architecture, and follow best practices when implementing our designs. To mitigate software crashes, we should set up a system that can recover from failure, and we should test heavily before deploying new versions to avoid introducing bugs to the production server. Finally, we should use a service to alert us if the server stops responding so we can take appropriate action.

BR3: Opposing groups successfully attacking us

This risk is likely, as the opposing groups are encouraged to attack us. However, it is not something that will have a significant impact on our grading, and it is something that we actively create controls to prevent. This earns it a likelihood rating of **Medium** but an impact level of **Low**.

Mitigation strategy

To mitigate this, we will specifically address the attacks presented to us in lectures and lab classes to prevent these from happening. While other attacks may be possible, focusing the

defenses on the ones we have learned of will likely suffice as our attackers have roughly the same knowledge as we do in terms of black-hat thinking.

Technical Risks

Technical risks consist of the risks directly related to the technicalities of technology. As a result, most of our technical risks are given **Low** likelihood as we have been taught how to protect the system in a technical way. They do, however, have varying impact levels. As with the business risks, only the most important technical risks are listed here, the rest can be found in Appendix A. Technical risks include (in descending order of significance): improper password hashing (TR11), tests not covering our requirements (TR10) and development bottlenecks caused by unfamiliar technology (TR1).

In addition, a number of technical risks also exist for specific types of attacks. These include cross-site scripting (XSS), injection-based, and cross-site request forgery (XSRF) attacks. As we are well equipped to handle these due to our knowledge from the lectures and lab sessions, we consider these as negligible in terms of likelihood.

TR11: Improper password hashing

Due to the inherent complexities of cryptography, there is a risk that we might improperly hash our password, which could lead to theft of private data should an attacker get through. This would break our requirement of creating a secure system (BG2), although we do not believe it to be something that would cause us to fail the exam. It is, however, still a very important area and as such could reduce our final grade. For that reason, it is given an impact and likelihood rating of **Medium**.

Mitigation strategy

In order to ensure that our password hashing is secure, we should use a peer reviewed and trusted hashing algorithm (such as BCrypt), and use a trusted implementation of this algorithm. We should under no conditions try to implement these ourselves.

TR10: Tests do not adequately show that the requirements have been fulfilled

This risk is relatively unlikely due to controls and procedures in place, but could have a severe impact on project fulfillment if it occurred. It could mean that some functionality or requirement was either not working or non-existent. This could potentially lead to failure of the course if it was a large number of critical functions. Because of this, it has been given a likelihood rating of **Low**, and an impact rating of **High**.

Mitigation strategy

In order to show that tests adequately show that all requirements have been met, we will cross-reference our tests with the list of requirements provided for the project as we go along.

TR1: Development bottlenecks caused by unfamiliar technology

If this risk was to materialize, we could suffer significant time delays, which could potentially lead to us not having enough time to fully implement all required functionality. This could have a significant impact on our grade. It is not very likely, however, as we are aware of the

problem and have controls in place to avoid bottlenecks. Due to the possible effects of this risk, it is given a impact rating of **Medium**, and a likelihood of **Low**.

Mitigation strategy

To mitigate this risk, we aim at working with technologies that most of us had experience with prior to the beginning of the project. When tasks come up that require learning an unfamiliar technology, we will attempt to isolate these to a separate subsystem in the architecture, such that the remaining subsystems can be worked on simultaneously.

Architecture Strategy

This section describes the strategies underlying our architecture. In the definition of the architecture for our system, we will strive to adhere to the following principles:

- Defense in Depth
- Reluctance to Trust
- Least Privilege
- Failing Securely
- Economy of Mechanism

First, we will describe the goals for our architecture. Following that, we will define strategies for attaining these goals.

Goals

AG1: Security

We need to have a secure architecture in order to protect the privacy of our users, and to protect the system against attacks from malicious entities. This architecture goal is directly related to BG2 and BR3.

AG2: Stability/fault tolerance

The architecture must enable the system to be stable and fault tolerant. Due to our business goal concerning uptime (BG3), it must be resilient to attacks.

AG3: Simplicity

The architecture must be simple, such that potential attack vectors can be easily identified. A simple architecture will allow us to easily make sure that all components are protected against attacks. Because of BR1, striving to keep the architecture simple will help us keep the development process efficient.

AG4: Modularity

We will keep the architecture modular, such that each component can be in development simultaneously. This is important with regards to BR1. Also, a modular architecture allows us to implement security measures for each module, compartmentalizing possible breaches.

Strategy

The following outlines our strategies for attaining each of our architectural goals.

AG1: Secure

In accordance with the Defense in Depth principle, we must secure every component of our architecture, so that a breach in one component does not by default mean a breach in any other component. This means that we will establish redundant security measures and make sure that different components do not trust other entities, neither internal nor foreign.

Adhering to the Defense in Depth principle will help us mitigate attacks from opposing groups (BR3), as well as help us protect sensitive data (BR6). It will also help act as a protection mechanism against injection attacks through input validation in every component (TR4).

Furthermore, we will secure our architecture by following the Reluctance to Trust principle. Our system will aim at having little to no trust in other systems, its users, or the environment in which it is run. We will, however, not focus on social engineering attacks, which means that illegitimate actions performed through authentication granted with the use of social engineering tricks is out of scope for this project. Following the Reluctance to Trust principle will also help us mitigate attacks from opposing groups (BR3), as well as inhibit illegitimate access to sensitive data (BR6).

Whenever possible, authentication will follow the principle of Least Privilege. More specifically, user accounts for operating systems, databases, etc. will have only the necessary access rights to do their intended work. This helps us alleviate the damage caused by potential attacks (BR3) and lessens the chance that an attack on one component leads to the failure of others (BR5 and TR2). By not allowing unnecessary access, we also gain some protection against theft of sensitive information (BR6).

AG2: Stable/fault tolerant

In accordance with the Failing Securely principle, we aim to make the system fail securely by not exposing unnecessary information to its users, and by utilizing secure defaults, such that users are not granted permission if authorization fails.

This helps us avoid illegitimate access by attackers (BR3), as well as ensures that servers do not go down if failures occur (BR5). It also plays a part in protecting sensitive data (BR6).

When calling APIs from other groups, we will make sure to run these requests in such a way that our own service will not be affected by an attempt to flood our server with data, or if the request times out.

AG3: Simple

By adherence to the Economy of Mechanism principle, we will focus on keeping the code simple with few choke points between components, which allows for increased code reuse, which is in accordance with the Don't Repeat Yourself² pattern. This will help us save time during development by avoiding development bottlenecks as all components can be developed

² Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

independently (BR1 and TR1). Also, simplicity will protect us against attacks by way of choke points (BR3).

AG4: Modular

We will develop the system as several coherent but interdependent modules, such that each team member can work on development and securing of a specific module. This is especially important due to BR1. However, modularization is also important in connection with the Defense in Depth principle. With a general lack of trust in other modules, we can establish a separate and self-contained wall of security measures around each module. Consequently, a potential attacker will have to break down each wall individually in order to breach the entire system. This will become important in order to mitigate BR3.

Design Principles for Implementation

Besides the design principles listed above, there are several design principles that do not affect the architecture as a whole, but rather should be followed when implementing the discrete components of the architecture. These are listed below.

Separation of Privilege

We will build the solution so that permission is granted by checking multiple privileges when necessary for security reasons. For instance, forms submitted will utilize two-factor authentication; a key that determines if a user is indeed the user claiming to be submitting, and a key that proves that the form is indeed meant to be submitted by that user.

This helps us against attackers attempting to gain access illegitimately (BR3), helps protect sensitive information through increased authorization-requirements (BR6 and TR12). It also helps guard against XSRF-attacks (TR7).

Complete Mediation

While we realize the implications and benefits of complete mediation, it is a principle that must be carefully weighed against the usability of the system. A completely mediated system would, for instance, not use session cookies and therefore be very annoying to use. Thus, we try to keep it in mind during design and development, but realize that adequate usability is almost impossible to achieve if the principle is fully adhered to.

This approach to the principle helps us provide a certain level of usability (BR4), and helps us create stronger password policies by helping us aim at making them as strict as possible (TR8).

Promoting Privacy

As is standard in authenticated systems, we will encrypt or hash sensitive information such as passwords whenever possible, so unintended access to the underlying data does not lead to leaking of sensitive information.

This helps us against the damage an attacker can cause (BR3), helps protect sensitive data (BR6), and helps remind us to properly utilize password hashing (TR11). It also helps us authorize users more properly before allowing them access to private data (TR12).

Requirements

The envisioned solution is a social networking site through which users can add an online identity and interact with other users on the site. Interaction can happen through establishing different kinds of relationships to other users, discovering common interests or hobbies as well as sending virtual hugs. As emphasized in our design principles, security is of paramount importance. Where usability and security concerns collide, security will be prioritized in order for us to meet our business goals (especially goals BG2 and BG4), unless it means a substantial decrease in usability.

Functional Requirements

Based on the user stories provided by the client (see Appendix B), we arrive at the functional requirements presented in this section.

We divide the users of the site into two categories: normal users and administrators. An administrator is just a normal user with added privileges, e.g. the ability to create and remove users at will. Below are the functional requirements for the system.

Normal users

A normal user must be able to:

Identifier	Description	Notes
FR1	Create a user on the site.	A name, a valid email address, and a password must be specified when creating a user.
FR2	Confirm account email.	To prevent unauthorized creation of accounts by arbitrary parties, the email provided when creating a user must be confirmed.
FR3	Log into account.	Logging in requires a valid email and a password.
FR4	Log out of account.	

FR5	Edit data <ul style="list-style-type: none"> • Edit name • Edit address • Change password • Add a hobby • Remove a hobby 	A name is required, so the name cannot be left empty. An address is optional. It should be possible to change password by providing a current password. Adding hobbies is optional.
FR6	Request a relationship to another user.	We define a relationship as either a Friendship, a Romance, or a Bromance.
FR7	Send hugs to friends.	
FR8	View hugs from friends.	
FR9	Accept or reject a relationship request.	Any relationship request received must be either accepted or rejected.
FR10	End a relationship.	Any user should be able to end a relationship with any other user at any time.
FR11	View profile information of other users.	A user must be in a relationship with another user before the address can be viewed. All other information besides email addresses should be available to all authenticated users.
FR12	Search for other users.	A user should be able to find other users' profiles by searching on their names.

Administrators

An administrator must be able to do anything that a normal user can do.

In addition, an administrator must be able to:

Identifier	Description	Notes
FR13	Create new users.	An administrator should be able to create arbitrary users for various purposes.
FR14	Remove any user.	An administrator should be able to remove any user from the system.
FR15	View all profile information.	An administrator is not required to be in a relationship with a user before viewing all the information pertaining to that user's profile, including email addresses.
FR16	Promote another user.	An administrator should be able to promote a normal user to have administrative rights.
FR17	Demote another administrator.	An administrator should be able to revoke any administrator's administrative rights.

APIs

Identifier	Description	Notes
FR18	Another authorized service similar to ours must be able to access our user data.	Other services should be able to interface with our service.
FR19	Our search function must show data from another site.	We should be able to interface with other services.

Non-Functional

The following non-functional requirements are based on both our design principles and our risk analysis. Seeing as this project revolves mainly around building secure systems, we have a strong focus on security requirements.

Security

Below is a list of our security requirements. Each of these have been deemed necessary to

include either as a consequence of one of our design principles or as a means to realize the previously defined mitigation strategies.

Identifier	Description	Notes
NFR1	The system must be behind a firewall.	Due to the Defense in Depth principle, the system must be behind a firewall.
NFR2	Any coherent component in the architecture must do input validation.	The Defense in Depth principle encourages that all systems have defenses against possible attacks. Consequently, no component should rely on other components to do input validation for them.
NFR3	All input must be validated against a <i>whitelist</i> relevant to each type of input.	Checking against a whitelist instead of a blacklist ensures that no input contains unexpected values. Injection and XSS attacks are stated as risks in BR3, TR3, and TR4.
NFR4	Authorization must only be granted after successful execution of validation.	Because we want the system to adhere to the Failing Securely principle, access to any object must only be granted after <i>successful</i> validation.
NFR5	No assumptions must be made about the validity of the content of any request.	No input should be trusted at any time prior to validation, i.e. any component should be reluctant to trust other components.
NFR6	All passwords must be hashed with a salt.	Due to the threat of opposing groups attacking our system, identified in BR3, we should protect the system against rainbow table

		attacks by hashing all passwords using a salt.
NFR7	All HTTP POST Requests to the server must provide a server-generated request key.	To protect against XSRF attacks (BR3, TR7), we should make sure that the system does not trust the browser to protect it against requests from unauthorized sources.
NFR8	Users must be logged in to interact with the system.	Due to privacy concerns identified in BR6, all users must be logged in before interacting with the system.
NFR9	Private information must only be viewable to users with proper authorization.	Also due to privacy concerns (BR6), users must be authorized to see other users' private information.
NFR10	Session keys must not be predictable.	Because of TR5, session keys must be handled in a non-predictable way, such that malicious users cannot predict the session keys of other (legitimate) users.
NFR11	Whenever session keys must be stored in a cookie, these must be stored in a secure way.	Following TR5, whenever we need to store a session key in a cookie, it must only be transferred over HTTPS.
NFR12	Passwords must have a length greater than 8 characters.	To make passwords harder to crack, they must be at least 8 characters long. This is a consequence of TR8.
NFR13	All external communication must happen over an encrypted connection.	As identified in TR9, all communication with the server must happen over a secure connection.

Reliability

The following reliability requirements have been deemed necessary to provide a proper service.

Identifier	Description	Notes
NFR14	The system must have an uptime of at least 95% after November 1, 2013.	This requirement is stated in the SLA.
NFR15	The server should not crash.	Faulty input from a user should not take down the server.

Usability

While usability is not our primary focus, the site should still be usable by a user with average experience with computers and social network sites. Based on this we have defined a set of usability requirements.

Identifier	Description	Notes
NFR16	A user must be able to do basic operations on the site, utilizing its features without assistance.	Without this our site would not be user friendly enough to be usable.

Performance

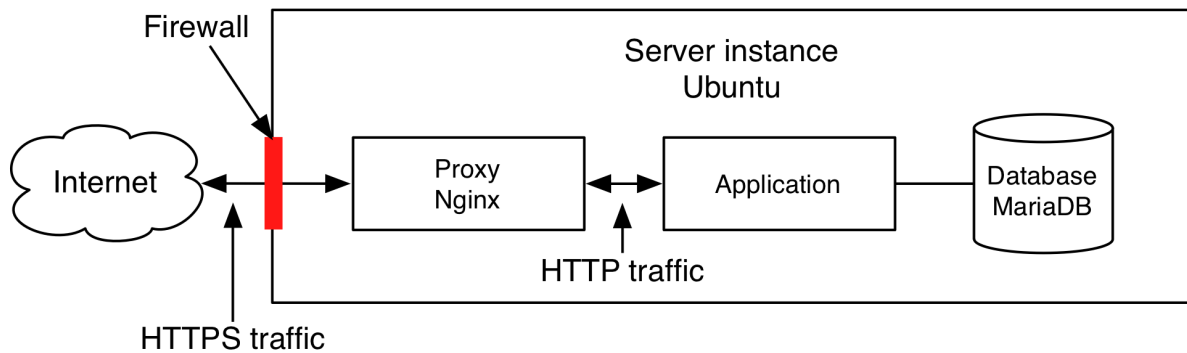
The last part of our non-functional requirements pertains to performance. These define the minimum performance of our server for it to provide a usable service.

Identifier	Description	Notes
NFR17	We must support a small number of concurrent users.	Since we support multiple users to be registered on the site we must also be able to support multiple users to be using the site at the same time.
NFR18	All HTTP requests must be handled and fulfilled within a reasonable time.	If the service is too slow it is not very usable.

Architecture

The system has a relatively simple architecture, with three main components: a reverse proxy, the application itself, and the database. These run on a single virtual private server (VPS) instance running Ubuntu 12.04.3 LTS³, and all communication between the components occurs locally (i.e. not over the network). This section will describe the architecture and security considerations for these three components.

³ "Ubuntu: The world's most popular free OS." 2002. 12 Dec. 2013 <<http://www.ubuntu.com/>>



Reverse proxy

Nginx⁴ serves as a reverse proxy to the application. It listens for incoming HTTP and HTTPS connections, and forwards them to the application, while redirecting all HTTP requests to HTTPS. Using a trusted web server for HTTPS access means our application does not have to implement HTTPS. As HTTPS is hard to get right, this is a security improvement, conforming to AG1 (security). If we wanted to implement protection against DDoS attacks, the reverse proxy could also act as a cache and load balancer, possibly mitigating the attack.

Application

The application is built in Scala, using the Spray.io⁵ web framework. We chose Scala for its strong, static type system, and the functional programming model it encourages. We believe the extra static checks and focus on immutability lead to safer, more secure code, and thus helping us attain AG1 (security). Spray.io was chosen because it is lightweight and modular. It also is relatively low-level compared to many other frameworks, such as Django⁶ or Play⁷. This was desirable, as one of our primary business goals is to learn how to implement a secure system (BG4). We decided that we would learn more by implementing as much of the system as possible ourselves. Thus the only parts of Spray.io we have used is the HTTP server and the routing system, allowing us to build “routes” that tell the Spray.io server how to handle requests. Spray.io is built with Akka⁸ actor framework, implementing the actor concurrency model. Akka actors let us achieve concurrency in a relatively simple manner. Akka actors also have powerful features for handling failures, letting us automatically restart actors that crash, helping us accomplish AG2 (stability/fault tolerance).

The application has four main layers, as well as several utility modules. The main layers are the service layer, the view layer, the model layer, and the database layer.

⁴ "nginx." 2009. 12 Dec. 2013 <<http://nginx.org/en/>>

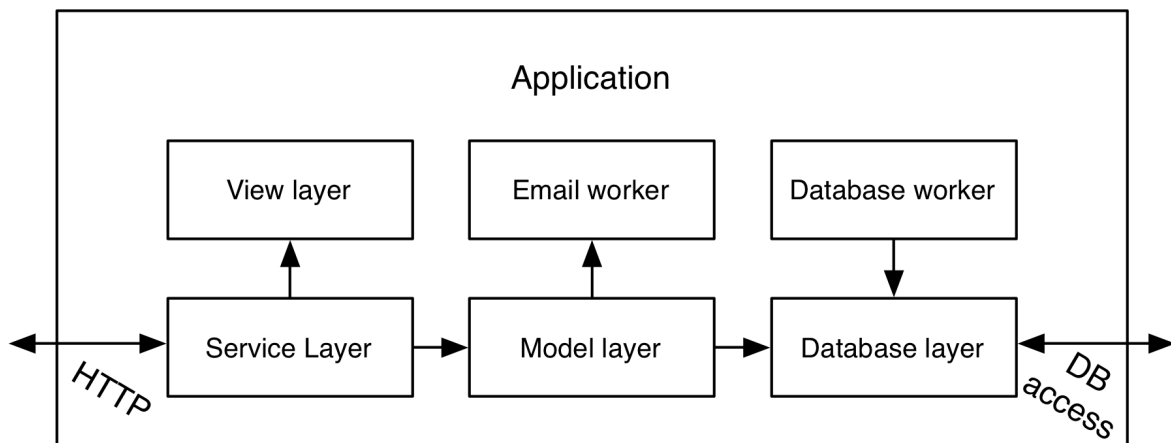
⁵ "spray | REST/HTTP for your Akka/Scala Actors." 2012. 12 Dec. 2013 <<http://spray.io/>>

⁶ "Django: The Web framework for perfectionists with deadlines." 2011. 12 Dec. 2013 <<https://www.djangoproject.com/>>

⁷ "Play Framework - Build Modern & Scalable Web Apps with Java and ..." 2008. 12 Dec. 2013 <<http://www.playframework.com/>>

⁸ "Akka." 2010. 12 Dec. 2013 <<http://akka.io/>>

Splitting the application into these layers and modules help us reach AG4 (modularity). This also means each layer can be smaller and more focused, as described in AG3 (simplicity). Each of these layers will be distrustful of the others, in accordance with the Reluctance to Trust principle. Each layer will also have its own security checks, from the outward facing service layer, to the innermost database layer, implementing the Defense in Depth principle. In the following subsections, each of these layers will be described, along with the utility modules.



Service layer

The service layer is the topmost layer and faces the internet at large, and as such will be the most exposed. It handles incoming requests, and fulfills them by using the model layer, which mediates all access to the database, and the view layer, which formats the information from the model layer as HTML, CSS and Javascript. The service layer is further split into three sub-services: the public services, the user services, and the admin services. Each sub-service only allows requests with the proper access rights to be processed. This makes it easy to offer a tiered access model, and is in accordance with AG4 (modularity).

All information retrieved from the model layer is validated before use, so even if something dangerous (like an XSS attack) is smuggled into the model layer, the service layer will produce an error before serving it, implementing Reluctance to Trust, and furthering AG1 (security). Services are run by a pool of actors. If any actor running a service crashes, it automatically restarts. This helps us achieve AG2 (stability/fault tolerance).

View layer

The view layer receives requests from the service layer to produce HTML pages that the service layer can return to the client. The view layer makes sure that all displayed information is valid and properly escaped before inserting it into the code. The view layer also holds the Javascript used for client-side validation of user-input information. It should be noted that the client-side validation is only used to display error messages before requests are made, as a service to the user, and is not expected to provide any additional security, as the client can simply disable Javascript. All this helps us achieve AG1 (security).

Model layer

The model layer mediates access to the database. It includes classes for representing users, sessions, and so on. All methods that can change the state of given model include consistency checks, and also validates and sanitizes any data before committing it to the database. The models also verify all data read from the database, so even if the database has been compromised, the data within it cannot be used for XSS attacks, for instance. This follows the Reluctance to Trust and Defense in Depth principles, and helps achieve AG1 (security). Encapsulating all state changes in this one layer also helps achieve both AG3 (simplicity) and AG4 (modularity).

Database layer

All direct database access is done through this layer using Slick, an ORM-like library for Scala. Among other things, Slick makes sure SQL injection is not possible. Slick lets us write database queries directly in Scala, using native functions. This gives us static checking, as well as an easy way to write queries. The static checking makes it much less likely to encounter SQL errors at run time, helping us achieve both AG1 (security) and AG2 (stability/fault tolerance).

Email worker

Emails used for confirming users are sent from the email worker. It sends these emails asynchronously from a queue. The worker runs in a dedicated thread.

Database worker

Elements like sessions and email confirmation entries need to expire after a given amount of time. To enable this, a database worker runs at set intervals, and executes various database tasks. It is run in its own thread, and can also carry out asynchronous database queries. Having all database maintenance run by a single worker helps us attain AG3 (simplicity), as it gives us a single place to define new tasks.

Utility modules

Besides these major components, the application also includes several smaller utility modules. These are described below.

Settings module

All settings for the application are stored here. This includes settings like minimum and maximum password lengths, regular expressions for valid user names, and so on. All these settings are loaded from a single configuration file at startup. Centralizing all settings in a single place makes it much easier to make changes without introducing inconsistencies or errors in the application. These settings are used throughout the application, and helps us attain both AG1 (security) and AG3 (simplicity).

Security module

The security module has two responsibilities: hashing and checking passwords. This is done using the jBCrypt library⁹, a widely used library implementing the BCrypt hashing algorithm. This algorithm was chosen due to its resistance to brute force attacks. All passwords are hashed with unique salts generated from a cryptographically secure random number generator. This module helps achieve AG1 (security).

Validation module

This module verifies whether different types of user input, such as names, hobbies, and emails, are valid. This is done using parameters read from the settings module. To verify whether an email is valid, we use the Apache Foundations EmailValidator¹⁰. This module is used throughout the application when validating data. This central implementation of validation ensures that different parts of the system cannot disagree on whether data is valid or not, and is helpful in accomplishing AG1.

Database

To comply with our Least Privilege design principle, the database is run as an less privileged user, and all access to the database is done through a MySQL user with as few privileges as possible, so if the application is compromised, the potential damage is limited. Further, a firewall is set up so database access is not possible from outside the local machine. Thus, an attacker must either compromise the application or somehow get full access to the VPS to be able to access the database.

Security Model

The systems security and rights model is separated into three layers of access; public, users and administrators. Naturally, the further in you get, the more rights you have.

Three Layers Of Access

Each client is assigned a unique session ID when they first visit the site. The state of this session dictates which access rights the user has. If the session is tied to a user with regular rights, it is in the user layer. If it is connected to a user with administrative rights, it is in the admin layer. If no user is attached, it is in the public layer.

Public

Public access is granted to all visitors to the website who are not logged in. They are allowed access to only the front page. This access layer covers functional requirements FR1 (Create a user on the site), FR2 (Confirm email account) and FR3 (Log into account). It should be noted, naturally, that confirming an email is only possible for the creator of the user tied to that email, as it requires the password to be reentered.

⁹ "jBCrypt - strong password hashing for Java - mindrot.org." 2006. 12 Dec. 2013
<<http://www.mindrot.org/projects/jBCrypt/>>

¹⁰ "Commons Validator - Apache Commons - The Apache Software ..." 2007. 12 Dec. 2013
<<http://commons.apache.org/validator/>>

Users

Users are any visitors that are logged in to the system. They can view and edit their own profile, add relationships and hug other users they are connected to. They are also able to search for other users and view their profiles, including friend-only information (such as address) if they are connected. This level covers the requirements set forth in NFR8 (Users must be logged in to interact with the system), NFR9 (Private information must only be viewable to users with proper authorization). It also covers all functional requirements set forth for normal users except for FR1, FR2 and FR3.

Administrators

Administrators are any user who have been upgraded to the administrator status. They are capable of seeing any users information, including the friend-only information. They are also capable of accessing the admin page, which has administrator-only functionality such as creating users. This layer covers all functional requirements set forth for Administrative users (FR13-17).

Validation of User Data

In order to validate any data input by the user, the validation module contains functionality to check and validate information. It utilizes the regular expressions and constants defined in the settings module to decide the validity of input. By centralizing this functionality in a common module, all separate components can easily agree on what constitutes valid data, and fixing holes in the validation is narrowed down to a single place in the codebase.

Testing Strategy

In order to validate the requirements put forth at the beginning of this report, we define strategies for evaluating the implementation success of the various types of requirements. In this section, the testing strategy for each category of requirements is described.

Testing of Functional Requirements

For testing the functional requirements, unit tests are to be used to verify the correctness of the functionality implementing the various requirements. These are to be implemented in ScalaTest. By cross-referencing these tests with our requirements, full coverage can be achieved. This also helps avoid programming errors during implementation, as unit tests can be run after each addition or alteration to verify the correctness of existing code.

Testing of Non-functional Requirements

While functional requirement testing is covered relatively easy with unit testing, non-functional requirements are not as straight forward to verify. Each type of non-functional requirement has its own strategy, which is covered below.

Security Testing

First, with a basis in the technical risks, unit tests will be constructed where applicable for functionality such as injection protection and input validation.

Second, to not only test our components from the inside out, but also from the outside in, a penetration test will be performed. This helps to ensure the security of the system, and helps locate any possible weak points. For this, we will follow the nine steps from the guide on penetration testing from the website pen-tester.dk¹¹: 1) footprinting, 2) scanning, 3) enumeration, 4) penetration, 5) escalation, 6) getting interactive, 7) expanding influence, 8) cleaning up, and 9) reporting.

We will also perform a penetration test simulating an attack from an internal threat, i.e. one of the group members.

Reliability Testing

We only have two reliability requirements; an uptime requirement (NFR14) and a requirement that the server must never crash from external input (NFR15). The uptime requirement will be covered by utilizing the website monitoring website Pingdom¹². It regularly pings the web-server for a response and saves the reply for later analysis. From that, we can extract the total uptime percentage. The second requirement is covered by our penetration test that will try to break the system with malicious input.

Usability Testing

As usability is not of the utmost importance to this project, it is the least tested requirement. We will perform simple informal usability tests where a few test subjects, unfamiliar to the project but familiar with other social networking sites, will attempt to use all functionality of the website available to non-administrative users.

Performance Testing

Our performance requirements consist of only two requirements; that the site covers a small number of concurrent users (NFR17), and that all requests are responded to in a reasonable amount of time (NFR18).

NFR17 will be covered by manually visiting the site on several computers concurrently and ensuring that no problems arise. NFR18 will both be covered by Pingdom and through manual checking. Pingdom logs the response time of requests and thus provides proof that the page does not occasionally suffer from slow responses on the whole, while manual tests where the response time is measured by the browser will ensure that all functionality responds within a reasonable timespan.

Evaluation

Architecture

In this section, we will evaluate our architecture as it is implemented against our architecture goals.

¹¹ "Pen-tester.dk - Penetration Testing." 2007. 12 Dec. 2013 <<http://pen-tester.dk/pentest.htm>>

¹² "Pingdom.com - Website Monitoring" 2013, 12 Dec. 2013 <<http://pingdom.com/>>

AG1: Security

We believe we have achieved this goal, primarily by adhering to the design principles mentioned in the Architecture Strategy section. Each layer validates and sanitizes all information it receives and sends. No component ever assumes that the data it receives has been validated or sanitized. This complies with the Defense in Depth and Reluctance to Trust principles.

One area where these principles are not followed is with access rights to the model layer. For example, the User model exposes a delete method, which deletes the user from the database. Only administrators are allowed to delete other users, so you could argue that the delete method should take an administrator as an argument. This would make it impossible for someone implementing a deletion feature to delete a user without also having a user with administration privileges at hand. This, unfortunately, would make the delete method's semantics unclear. Is it a method on the user object, or a static method deleting the user passed as an argument? Adding these kinds of checks to all the methods in the model layer would make it more difficult to work with, and we decided to instead keep these kinds of checks on the service layer. As there is no way for an attacker to access the model layer directly, we argue that this is an acceptable tradeoff, although it means we must be vigilant when accessing the model layer, making sure the user has been properly authorized.

We follow the principle of Least Privilege, running all software components as special users with few privileges. Further, the application only has limited access to the database, and cannot create or drop tables.

When something goes wrong, we make sure to follow the principle of Failing Securely. We make sure to never show the user sensitive details if an error occurs, and since all our database operations are atomic, we do not leave the database in an inconsistent, possibly dangerous state. If a request causes a critical error, causing some part of the application to crash, the affected component is restarted. Since the application is stateless, this will cause the request to fail, but will not have other negative effects.

By following our architecture strategy along with the relevant design principles, we believe we have implemented an architecture secure enough to accomplish this goal.

AG2: Stability/fault tolerance

As mentioned above, our architecture follows the principle of Failing Securely. Not only does our system fail securely, it also fails in a recoverable manner. Any component of the application that crashes will be restarted. The reverse proxy and database are also configured to restart if they crash. Finally, if everything should go wrong, and the server went down completely, we would receive a notification from the Pingdom service when it could not contact the server, allowing us to quickly start working on a solution.

We have also implemented a suite of unit tests covering the most critical parts of the application, increasing our confidence that the application will not crash. Based on these points, we believe we have achieved this goal.

AG3: Simplicity

Through inheritance and constant avoidance of repetition, we have achieved a large level of reuse. Additionally, through the use of a small amount of clearly separated modules, we have kept choke points between components to a minimum. This allowed for a faster development cycle and a smaller codebase, and follows the principle of Economy of Mechanism. Further, we have a small number of different kinds of components, keeping the overall system relatively simple. Thus, we believe we have achieved this goal.

AG4: Modularity

We have achieved a modular architecture by keeping separate concerns separate in the architecture. Accordingly, we have split the architecture into four coherent layers: The view layer, the service layer, the model layer, and the database layer. As such, each of these layers implement their own security measures, and do not trust components of other layers to perform input validation for them. Based on these properties, we believe we have accomplished this goal.

Evaluation of Functional Requirements

In order to evaluate our implementation with regards to the functional requirements, we need to show that all of these have been fulfilled. We achieve this by performing unit tests on our model and service layers. Naturally, running unit tests does not rigorously prove that our system correctly provides the specified functionality, but it increases our confidence in the system significantly.

Our unit tests focus on testing that our data access objects and our exposed services are well-behaved and functioning. For the full list of unit-tests, see Appendix C. Recalling our requirements from earlier, this is how each requirement is covered by the tests. This cross-check helps us ensure that our tests cover the required functionality (TR10).

Requirement	Test
FR1: Create a user on the site	T1 T2 T30
FR2: Confirm account email	T2 T3 T5
FR3: Log into account	T19 T31
FR4: Log out of account	T20
FR5: Edit data	T15 T16 T22 T34 T35
FR6: Request a relationship to another user	T9

FR7: Send hugs to friends	T8 T12 T36
FR8: View hugs from friends	T14
FR9: Accept or reject a relationship request	T10 T11 T18 T32 T33
FR10: End a relationship	T17
FR11: View profile information of other users	T24
FR12: Search for other users	T7
FR13: Create new users	T25
FR14: Remove any user	T26
FR15: View all profile information	T27
FR16: Promote another user	T28
FR17: Demote another administrator	T29
FR18: Expose API	T37

Note that FR19 is missing from the above table. Testing FR19 would involve creating test data on another site, and since this is not possible for us to do we have not been able to test this programmatically. Therefore we have tested that we fulfill this requirement through black-box testing.

Evaluation of Non-Functional Requirements

The following tables list, for each non-functional requirement, the measures we have taken in order to fulfill it.

Evaluation of Security Requirements

Identifier	Fulfillment
NFR1	We have used the most recent version of the firewall software <i>iptables</i> to ensure that only the necessary ports are open for communication.
NFR2	We have made sure that every component in the view, service, model, database layer use the validation functions from the settings module to validate any input before use.
NFR3	The validation provided in the settings module provide positive validation checks. This means that input is only validated if it matches the range of expected and whitelisted input. For simple input, regular expressions are

	used to define the range of acceptable values. For more complicated validation, such as email validation, external but widely used libraries have been used.
NFR4	We make sure that all input validation is checked using the settings module, and that all checks actually using functions from this module only grant appropriate access if these functions return a positive result.
NFR5	We ensure that all components implement validation on all input. As such, no assumptions of the validity of any input is made.
NFR6	All passwords are hashed with a unique salt.
NFR7	All websites generated for the user come with a freshly generated <i>form key</i> . When a <i>POST</i> request is handled this form key is then checked for validity. If it is not valid, the request is rejected. Each form key can only be used once after which it is revoked.
NFR8	Without being logged onto the system, a user is only allowed to view the front page, create a new user, login as an existing user, and confirm an account. All other services at least require a valid session.
NFR9	We previously defined <i>private information</i> as a user's address. If two users are not in a relationship, their addresses will not be visible to each other. All <i>public information</i> is available to all registered users, but not to people without an account.
NFR10	All sessions are generated independently using Java's <i>UUID</i> library.
NFR11	All cookies have the <i>Secure</i> flag set.
NFR12	All passwords are validated against a set of requirements, amongst these a length requirement of at least eight characters.
NFR13	All communication with external sources use a secure connection (<i>SSL</i>).

Evaluation of Reliability Requirements

Identifier	Fulfillment
NFR14	See NFR15.
NFR15	All internal exception are caught and handled properly. Only the database has state and all communication with the database is done through transactions. As such the server cannot end up in a bad state.

Evaluation of Usability Requirements

Identifier	Fulfillment
NFR16	We have carried out usability tests on two individuals. While this is generally considered too few for proper usability testing, we have deemed it sufficient as usability generally is not the focus of this project.

Evaluation of Performance Requirements

Identifier	Fulfillment
NFR17	The server handles several requests currently by using to pool of actors. Whenever a request comes in, an appropriate actor is handled the request. Since actors work concurrently, our system is able to handle several concurrent users.
NFR18	Based on manual tests for every page, the response time is between 600ms and 1 second. This is deemed reasonable. The average response time from Pingdom is about 600ms as well (see Appendix E). We believe we fulfill this requirement.

Testing of Security Requirements

Risk-Based Testing

For security testing specific components of our system, we have carried out risk-based unit tests. We have done two tests checking part of our model layer for XSS attacks (*T38*, *T39*). Only having two tests is not sufficient for our risk-based testing to hold much value. Ideally we should have tested for more types of attack, and also on the service layer. Sadly, due to time constraints we did not do this. Luckily, penetration testing covers the same attacks and as such should suffice to provide confidence in the security of the system.

Penetration Testing

External threats

As previously mentioned, our penetration tests follow the nine steps described in the penetration testing guide found on the website pen-tester.dk.

1. Footprinting: Revealed information such as location and which hosting service was used. This was done using the `whois` command along with a geolocation tool¹³. The most important information, namely the IP address, was given beforehand.

¹³ "View my IP information: Geo IP Tool." 2006. 15 Dec. 2013 <<http://www.geoiptool.com/>>

2. Scanning: Using the nmap¹⁴ tool showed that we do not have any ports open apart from 22 (SSH), 80 (HTTP), 443 (SSL).
3. Enumeration: Identification of operating system and services was done by examining HTTP headers received from port 80 and 443. These showed that the system uses an Nginx HTTP server. It should be noted that this only revealed our reverse proxy, and not the actual application server. The operating system was not revealed.
4. Penetration: For this, we have carried out the same attacks on our own system as we did in our attempts to breach the security of the other groups. These include:
 - XSS attacks, both persistent and ephemeral.
 - XSRF attacks.
 - SQL injection in all form elements.
 - Replay attacks.
 - Packet sniffing.

None of our penetration attempts were successful. As none of our attacks succeeded, we were unable to gain deeper access to the system.

Internal threats

Since, threats do not always come from the outside, we also evaluated our internal security by simulating an attack coming from within the group. The simulated attack was a success, gaining full access to the system, and the ability to lock out the other group members. This is because each group member has full root access to the server. A solution could be to only give one group member full access, and give all other group members fewer permissions. This would add much overhead to daily work though, and could lead to bottlenecks in development, especially if the user with full access is unavailable, thus potentially affecting BG1, BG3, and BG7. Of course, the user with full access can still compromise the system. Instead, all group members have full access, even though this might compromise BG2.

Testing Usability Requirements

We have performed usability tests of the web interface of the system on two users with general familiarity with social networking websites, but otherwise have no technical background.

Specifically, we have tested that the test users were able to easily perform the following tasks:

- Creating a new user
- Logging onto the system
- Editing profile
- Searching for another user
- Requesting a relationship with another user
- Accepting a relationship with another user

In terms of results, we found that all test subjects could perform the tasks without assistance. The full results of the usability tests can be found in Appendix D.

¹⁴ "Nmap - Free Security Scanner For Network Exploration & Security ..." 2003. 15 Dec. 2013
<<http://nmap.org/>>

Business Goal Fulfillment

BG1: To create a system that lives up to the requirements specified in the project description

Through our through our explicit linking between requirements and testing, we have achieved a system that lives up to the requirements specified in the project description.

BG2: To create a secure system, that can withstand attacks from the other groups

We have only been susceptible to two attacks as will be mentioned in the next section.

However, these attacks were either mitigated after the fact, or were of such a nature that it is virtually impossible to defend against. Thus, we argue that we have adequately withstood attacks from other groups.

BG3: To live up to the Service Level Agreement (SLA) of 95% uptime

As is shown in Appendix E, our uptime has exceeded 95% as documented by Pingdom. Thus, we have met our SLA.

BG4: To learn how to design and implement secure systems

As evidenced by the our evaluation, our system is adequately securely implemented. As this was the result of a careful plan, we argue that this is proof that we have learned how to design and implement secure systems.

BG5: To obtain an acceptable level of usability in spite of necessary security measures

Through some, albeit not very extensive usability tests, we have shown that an average social networking user can navigate and utilize the website. This should be proof that we have met this goal.

BG6: To strengthen our project by successfully attacking other groups

As is documented in the next section, we have successfully carried out two attacks on other groups which are arguably fair and direct attacks with straight-forward countermeasures that were not in place.

BG7: To meet the deadline for project hand-in

As of the writing of this report, we have no way of knowing whether we have fulfilled this goal. If you are in fact reading this, you can assume that we have.

Hacking

We performed a number of attacks on other groups, most of which were unsuccessful. In the following, we explain the ones that worked.

Team 10: Session-hijacking through XSS

Vulnerability: Lack of input validation for registration field.

Attack: XSS with access to session cookie.

Users vulnerable: Any logged in user who visits our malicious webpage.

Time of attack: 6th of November

Description: We discovered a lack of input validation in the registration page of their website. That meant that with a simple input string that ended the current control (“/”), we could write any HTML or Javascript we wanted and it would be rendered on the page.

Although it was not possible to create a user with a script for a username in order to obtain stored XSS, we could still perform regular XSS through this method.

We utilized this to produce a webpage that would post a form request to the registration page, which was possible to post to even though a user was already logged in, and have that form request inject some Javascript which would redirect the user to our website, in which the session key was posted as a query string parameter. This session key can then be used to access the users account. Basically, all a user would have to do to give up his account was visit a seemingly innocent link on our web-server while he was logged in to their site.

Although this attack is successful in Firefox browsers, Chrome and Safari both provide some XSS protection. Googling how to avoid these XSS filters provides a lot of workarounds, however, so circumventing these filters is possible.

Mitigation Strategy: Either enable HttpOnly so the cookie cannot be accessed and/or sanitize input fields.

Team 1: Cookie-theft through packet sniffing

Vulnerability: Incorrectly configured cookie (Secure disabled)

Attack: Packet sniffing on unswitched network allows reading of session cookie.

Users vulnerable: Any logged-in user who visits any non-SSL page on same unswitched network.

Time of attack: 6th of November

Description: The team accidentally deployed a version of their website in which the Secure-Only attribute of their session key was disabled for testing reasons. This meant that a packet sniffer on an unswitched network would catch HTTP requests and have the session key displayed in cleartext. This attack could be performed by setting up an open hotspot in a public setting and waiting for someone who is logged in to the site to connect. For pages such as Facebook, this should not take long.

Other teams susceptible:

- Team 2: No SSL, no Secure attribute as of November 8
- Team 3: No SSL, no Secure attribute as of November 8
- Team 5: No Secure attribute as of November 8
- Team 6: No SSL, no Secure attribute as of November 8. Has session id in URL parameter so SSL does not matter.
- Team 7: No SSL, no Secure attribute as of November 8.
- Team 8: No Secure attribute as of November 8 (However, signup does not work so it may change the cookie-attributes once you sign in).
- Team 10: No Secure attribute as of November 8.
- Team 4 did not have their server running at the time of attack.

Mitigation Strategy: Enable Secure attribute for the session cookie.

Successful attacks on us by other groups

While no conventional penetration attacks was successful (as far as we've been informed) by other groups, two groups did manage to abuse the system.

Group 1: Session prediction

Initially, a session was generated for clients who weren't logged in. This session would then be tied to the user after a successful login. However, this meant that an attacker with physical access to a computer could note the session key, wait for another user to login using the computer, and then access their account from another computer using the same session key which was now tied to the user. This attack was stopped by regenerating a key on login.

Group 8: Man-in-the-Middle attack through ARP poisoning

This group performed a Man-in-the-Middle attack by performing an ARP poisoning attack on the victims computer through which all traffic destined for our site was directed to their computer. They used an SSL-stripper to proxy our website and thus allow the user to be presented with a non-secure version of our site. The victim would then post his login information through an unsecured connection, which could then be sniffed to obtain the password.

This attack is on the ethernet layer of the victims computer, so it is effectively impossible for us to ward against completely. However, using the *Strict-Transport-Security* (STS) response header will provide some protection against SSL stripping by forcing clients to only view the page through a secure connection. Users who have already visited the page prior to the attack would have the presence of the STS header cached, and thus be protected through this counter-measure. However, a prerequisite to this defense is that our SSL-certificate is trusted, i.e. signed by a CA authority, which is not the case for this project.

Conclusion

In this report, we have outlined the design and implementation of the social networking site RaptorDating. We began by defining the goals of the project, which allowed us to identify the inherent risks to these goals. We then linked these risks to requirements, which made it possible to define an architecture and design philosophy that would help us meet these requirements.

We also showcased our offensive abilities through attacks on other groups, and noted attacks from other groups that had an effect on our solution.

To evaluate the solution, we outlined testing strategies and evaluated our solution based on these strategies. As our evaluations showed, all of our requirements are satisfactorily met, and we achieved all of our business goals. We argue, that this was possible by maintaining a tight coupling between not only our tests and requirements, but also the requirements and our business goals.

Appendix

Appendix A – Risks

Business Risks

- BR1: Lack of time
 - Threatens BG1, BG2, BG5, BG6, BG7
 - Likelihood: M
 - Impact: H (Not all requirements fulfilled)
 - Severity: M
- BR2: Internal issues in the group
 - Threatens BG1, BG2, BG3, BG4, BG5, BG6
 - Likelihood: L
 - Impact: H (Lack of trust in other team members and less manpower)
 - Severity: L
- BR3: Opposing groups successfully attacking us
 - Threatens BG2, BG3, BG6
 - Likelihood: M
 - Impact: L (The consequence of being hacked is very low)
 - Severity: L
- BR4: Usability too low for intended users
 - Threatens BG5
 - Likelihood: L
 - Impact: M (System might become unusable)
 - Severity: L
- BR5: Servers become unavailable
 - Threatens BG1, BG3, BG5
 - Likelihood: M
 - Impact: M (We might not meet SLA requirements)
 - Severity: M
- BR6: Improper privacy-protection mechanisms
 - Threatens BG1
 - Likelihood: L
 - Impact: L (We might be hacked)
 - Severity: L
- BR7: Insufficient implementation of required functionality
 - Threatens BG1
 - Likelihood: L
 - Impact: H (Not all requirements are fulfilled)
 - Severity: L

Technical risks

- TR1: Development bottlenecks caused by unfamiliar technology
 - Relates to BR1, BR2
 - Likelihood: L
 - Impact: M (We might not have enough time)
 - Severity: L
- TR2: Insufficient Denial-of-Service protection
 - Relates to BR5
 - Likelihood: L
 - Impact: L (SLA not fulfilled - not “legal” attack in project though)
 - Severity: L
- TR3: Insufficient XSS Protection
 - Relates to BR3, BR6
 - Likelihood: L
 - Impact: M (Security requirements not fulfilled)
 - Severity: L
- TR4: Insufficient protection against injection-attacks
 - Relates to BR3, BR5
 - Likelihood: L
 - Impact: M (Security requirements not fulfilled)
 - Severity: L
- TR5: Improper session handling
 - Relates to BR3, BR4, BR6
 - Likelihood: L
 - Impact: M (Security requirements not fulfilled)
 - Severity: L
- TR6: Improper separation of privilege
 - Relates to BR3, BR5
 - Likelihood: L
 - Impact: M (Breach of security principles)
 - Severity: L
- TR7: Improper XSRF protection
 - Relates to BR3, BR4, BR6
 - Likelihood: L
 - Impact: M (Security requirements not fulfilled)
 - Severity: L
- TR8: Improper password policies
 - Relates to BR3, BR4, BR6
 - Likelihood: L
 - Impact: L (Improper handling of sensitive data and security policy breaches)
 - Severity: L
- TR9: Wrongly configured or missing SSL

- Relates to BR3, BR6
 - Likelihood: L
 - Impact: M (Security requirements not fulfilled)
 - Severity: L
- TR10: Tests do not adequately show that the requirements have been fulfilled.
 - Relates to BR7
 - Likelihood: L
 - Impact: H (Requirements may not be fulfilled due to not detecting it through testing)
 - Severity: L
- TR11: Improper password hashing
 - Relates to BR6
 - Likelihood: M
 - Impact: M (Potential leaking of sensitive data)
 - Severity: M
- TR12: Improper client authorization
 - Relates to BR6
 - Likelihood: L
 - Impact: M (Security requirements not fulfilled)
 - Severity: L

Appendix B – User Stories

Story	Description
Prepare the platform	Secure your slice server. Install necessary software.
Release the URL	Your applications landing page has to be accessible from the following URL your_ip_adress/ssase13 (you can still call your application whatever you want)
Profile	Expanded functionality from Part I A user can add data to his/her profile about: <ul style="list-style-type: none"> • Name (*) • Adress (*) • Hobbies • Friends /Romances Fields marked with (*) are only allowed to be viewed with proper authorization
Admin	Expanded functionality from Part I An admin can view all the profile information, can delete and create new users
Sign up	A user can decide to create an account, which enables her to add a profile
Virtual Hug	Expanded functionality from Part I Our marketing needs to run a campaign: Implement a virtual hug, that can be exchanged between users that are friends.
Protect against attacks	Expanded functionality from Part I Our users are very concerned about the security Marketing would like us to state proudly on the front page which attacks we have secured them from, that the competition haven't.

	Follow this up with an abuse case, and a functional test which demonstrates, that your application is secure from these particular sins.
Be Online	Marketing wants you to commit to a 95% up-time SLA for the time period 01-11-2012 – 15-12-2012. Make sure you are online BEFORE 01.11.2012. Document that you live up to the SLA with a log-file, or similar.
Penetration attack	Apply your security test (or attack) for at least 1 peer slice (No form of brute force is allowed, as we live on shared infrastructure).

Appendix C – Unit Tests

Functional Testing

The purpose of our functional tests is to follow up on our functional requirements from earlier. Our unit tests focus on testing that our data access objects are well-behaved. To do this we have the tests

T1: Creating user
T2: Newly created user is unconfirmed
T3: User is confirmed
T4: Creating second, confirmed user
T5: Second user is confirmed
T6: Set user to admin
T7: Search for a user
T8: You cannot hug a stranger
T9: Request friendship
T10: Receive friendship request
T11: Accept friendship request
T12: You can hug a friend, many times!
T13: You can mark a single hug as seen
T14: You can see unseen and seen hugs
T15: User can add hobbies
T16: User can remove hobbies
T17: Remove friend
T18: Reject friendship
T19: User can log in
T20: User can log out
T21: User can't log in with wrong password
T22: Change name
T22: Change email
T23: Delete user

Service tests:

T24: User can view profile page of other user
T25: Admin can create new users
T26: Admin can delete a user
T27: Admin can view a profile page of a user
T28 Admin can promote another user to admin
T29: Admin can demote another admin to regular user
T30: User can sign up and confirm
T31: Login works
T32: User can request friendship, and user can accept it

T33: User can request friendship, and user can reject it

T34: Added hobby appears as hobby on user

T35: Removing a hobby causes a hobby to be removed

T36: You can hug a friend

T37: API keys can be created and work. Stops working after being revoked.

Risk-Based Testing

T38: XSS attack in hobbies

T39: XSS attack in name

All of the above tests have been carried out and are successful.

Appendix D – Usability Tests

Results of Usability test for User 1

Age: 24

Sex: Female

Occupation: Student

Familiarity with social networks: High

Task	Result
Creating a new user	Passed
Logging onto the system	Passed
Editing profile	Passed
Searching for another user	Passed
Requesting a relationship with another user	Passed
Accepting a relationship with another user	Passed

Notes: She accomplished all tasks without any help. She was slightly worried when it took a while for her confirmation email to arrive.

Results of Usability test for User 2

Age: 32

Sex: Male

Occupation: Student


Familiarity with social networks: Medium

Task	Result
Creating a new user	Passed
Logging onto the system	Passed
Editing profile	Passed
Searching for another user	Passed
Requesting a relationship with another user	Passed
Accepting a relationship with another user	Passed

Notes: Accomplished all tasks with no help. Was met with a XSRF error message when pressing the “Back” button in the browser after searching and then trying to do another search.

Appendix E – Uptime and Average Response Time

Can be viewed at: <http://stats.pingdom.com/f7ogpdbhlowg/1012340/history>



15-12-2013 10:23:36 (GMT +1:00)
The shown time zone is the same as yours

Raptor Dating (history)

RecentHistory

Name ▼	Uptime	Response Time
December 2013	99,99%	584 ms
November 2013	99,91%	613 ms
October 2013	99,97%	605 ms

Uptime monitoring provided by [Pingdom](#)[Get your free account](#) and monitor your uptime

Appendix F – Work Log

Work log

Week 1 (13/09)

Everyone

- Agreed on software and frameworks
- Set up basic environment on server

Week 2 (16/09 – 20/09)

Christian Harrington, Emil Bech Madsen, Thomas Didriksen

- Wrote team contract
- Wrote project plan
- Wrote requirements
- Designed database
- Designed architecture
- Setup database on server

Week 3 (23/09 – 27/09)

Christian Harrington

- Worked on database layer
- Worked on tests
- Worked on validation

Sune Alkærsig

- Worked on service layer

Thomas Didriksen

- Worked on HTML, Javascript
- Worked on page templates

Week 4 (30/09 – 04/10)

Christian Harrington, Sune Alkærsig, Thomas Didriksen

- Worked on integration of different layers
- Worked on bugfixes

Week 5 (7/10 – 11/10)

Christian Harrington

- Worked on refactoring services into smaller components
- Improved logging

Week 6 (14/10 – 8/10)

- Autumn break

Week 7 (21/10 – 25/10)

- No work on project

Week 8 (28/10 – 1/11)

Christian Harrington, Sune Alkærsig

- Added hugs to the model and database
- Wrote better deploy script

Emil Bech Madsen

- Added validation of output from database

Thomas Didriksen

- Added hugs support to pages

Week 9 (4/11 – 8/11)

Christian Harrington

- Fixed bugs in hugs
- Fixed security bug with sessions

Christian Harrington, Emil Bech Madsen, Sune Alkærsig

- Worked on attacks on other groups

Thomas Didriksen

- Fixed page bugs
- Encode all output from model

Week 10 (11/11 – 15/11)

Everyone

- Worked on report

Week 11 (18/11 – 22/11)

Christian Harrington

- Worked on user API

Thomas Didriksen

- Worked on pages and design

Emil Bech Madsen

- Worked on hugs and email confirmations

Week 12 (25/11 – 29/11)

Everyone

- Worked on report

Christian Harrington

- Fully implemented API
- Implemented search using Brocial Network API

Emil Bech Madsen

- Added constraints to hobbies
- Added tests

Sune Alkærsig

- Implemented service-level tests

Thomas Didriksen

- Improved tests with random test data
- Added API key management to admin interface

Week 13 (2/12 – 6/12)

Everyone

- Worked on report

Christian Harrington

- Cleaned up services

Week 14 (9/12 – 15/12)

Everyone

- Worked on report

Appendix G – Team Contract

Structure

Flat structure with a designated team coordinator. Team coordinator has no special powers, but is in charge of making sure everyone in the group gets the information they need, and has something to work on.

Roles

- Christian Harrington — Team coordinator
- Emil Bech Madsen — Code monkey
- Thomas Didriksen — Code monkey
- Sune Alkærsig — Code monkey

Place and times to meet

Place: ITU

Time:

Weekly work sessions following the lecture from roughly 10-12, as well as minimum of one work session during the week, arranged through outlook-calendar invites.

Work norms

Team members get assigned tickets on Github with a deadline attached, and must complete them by the deadline.