IT 10 036

Examensarbete 15 hp
Augusti 2010

Authentication and Authorization

Achieving Single Sign-on in an Erlang Environment

Fabian Alenius

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Authentication and Authorization

Fabian Alenius

Forcing users to enter their credentials every time they want to use a service is associated with several problems. Common problems include lowered user productivity, increased administration costs and security issues. As companies and organizations are adding more services to their networks, it is becoming an increasingly important problem. By allowing users to sign on once and centralizing access control decisions, it is possible to reduce or completely mitigate this problem. This paper describes how a service written in Erlang was modified to allow for single sign-on and centralized access control.

Handledare: Erik Stenman
Ämnesgranskare: Björn Victor
Examinator: Anders Jansson
IT 10 036
Sponsor: Kreditor

Tryckt av: Reprocentralen ITC

Chapter 1

Introduction

In this paper we will study the problems associated with authentication and access control in corporate networks. We retrofit an existing service to enable single sign-on and centralize access control. The design described in chapter 4 will show how to enable single sign-on, by using a network authentication server to verify user credentials. We will also show how to centralize authorization decisions with the help of a directory server. The problems that were encountered during the implementation are discussed together with their solutions.

1.1 Background

Because password authentication is so easy to implement, it is the most commonly used authentication method [9]. When using password based authentication, the relative security of individual passwords can vary. For example, a longer password is more secure than a short password. Yan et al. [18] conducted a study on password security where they found that good passwords should be long and random. While good for security, these restrictions increase users difficulty in remembering their passwords. Users who forget their password need help from the helpdesk to retrieve or reset their password. If one user forgets his or her password, it's typically not a problem. However, in a large corporation or organization it's not uncommon to have several password-related issues every day [1]. There are also security problems related to forgotten passwords. Users who often forget their passwords sometime resort to writing them down. Passwords lying around for anyone to see is a disaster to network security. The problem with passwords only worsens as the number of services on the network increase.

There are very tangible costs related to password authentication. Fortunately there are solutions, as we will see in chapter 3. However, in a study that surveyed IT administrators, Ahlborg and Hålsjö [1] found that while administrators were aware of the problems, they were unable to fix them. The reason cited by the administrators was that the problem of retrofitting existing services range from difficult to impossible. For example, the source-code for old legacy services that are no longer in active development is often hard to obtain. In practice, this makes retrofitting these services an impossible task.

1.2 Problem definition

A typical corporate network offers a wide range of applications and services, but before a user can utilize the services, the user has to login. The user has to *authenticate* themselves (sec. 2.2) . A simple way to authenticate users is to supply them with usernames and passwords. The problem with passwords is that they are hard to remember. In general, the better a password is, the harder it is to remember [18]. This is especially true if users need to remember a different password for each service. The problems associated with authentication have greatly increased with the introduction of web-services in corporate networks. Today it's not uncommon for users to authenticate themselves 25 times per day [13]. One way to reduce this number is by enabling *single sign-on*(SSO) on the network. SSO enables users to sign on once and not have to retype their passwords during the session, even as they utilize different services.

Another problem on large networks is that of *authorization*: all users should not have the same access rights. For example, a customer service representative should not have access to financial data, while someone working in the financial department should not have access to personnel files. In a network that provides a wide range of services, management of access control can become a serious burden. By centralizing authorization decisions, management can be simplified, while at the same time alleviating the problem of managing user accounts across different services. In this paper we conduct a case study in which we retrofit a service to use SSO. We also centralize authorization decisions with the help of a LDAP server (sec. 3.3). Specifically, we will show how to:

- Enable single sign-on for a web service using a Kerberos server (sec 3.1).
- Provide fallback to username/password authentication for users that can't directly communicate with the Kerberos server.
- Centralize authorization using a directory server.
- Enable single sign-on for a service written in Erlang.

1.3 The case

Kreditor AB, which we will refer to as simply Kreditor, is a Stockholm-based financial organization that offers financial services over the internet. Kreditor is growing very fast: in the last year alone, they have increased from 70 to 130 employees. As the number of employees has increased, so has the effort required to manage the users. Kreditor uses a directory server to store employee records. The records hold information such as employee names, email addresses, passwords, and more. The directory server makes it possible to look up employees and see what departments they belong to. Kreditor uses a Kerberos server to authenticate the users on their network, Kerberos is described in section 3.1.

Kreditor has one primary service that almost all employees use, the *kred system*, which we will refer to as simply *kred*. Kred is written in Erlang, a programming language that according to its creator Joe Armstrong [2] was designed for writing concurrent programs that "run forever". Because Kreditor has very high reliability demands on their services, this is an important property. Embedded in the kred system is a web interface named Kreditor Online (KO). KO does not use existing network infrastructure for authentication and access control. Instead, KO stores user accounts that are unique to KO and not in any way synchronized with the accounts in the directory server. Accounts in KO hold a subset of the information stored in the directory server, including a separate username and password. As a consequence, users have to remember two different username/password pairs.

Kreditor wished to integrate the authentication and access control in KO with the existing infrastructure in order to consolidate IT management. Enabling SSO would increase user productivity and cut help-desk costs [1]. As kred is written in Erlang, a language that isn't widely used, there is no existing literature or study that covers this exact problem. However, the general solution outlined in [15] proved applicable with some small modifications. The specification from Kreditor is outlined in section 4.3 and the subsequent design in section 4.4.

1.4 Motivation

According to a study by the Network Applications Consortium (NAC) [12] up to 44 hours a year is spent by employees solving authentication related problems. This is more than one week every year that could be spent doing something more productive, a huge cost to companies that virtually every company has. Centralizing authentication is often done as an afterthought when the need becomes apparent. Refitting an enterprise network to use single sign-on can often be problematic and become very expensive. Because of this, studies examining this problem should be helpful to companies and organizations that are faced with this problem. Because Kreditor's environment is somewhat unique, the implementation details in this case study are likely to be particularly interesting.

1.5 Related research

Xiong [17] reviews popular web SSO solutions and conducts a case study where he presents a design that uses Central Authentication Service[1]. The design presented in this paper is more secure than Xiong's: it primarily relies on SPNEGO(sec. 3.2.2) for authentication. When CAS is used for authentication, the username and password have to be sent over the network every time a user wishes to authenticate his or her identity to a service. With SPNEGO, this is not necessary. Instead, a *token* is sent. The details of the design are explained in chapter 4.

Kabay [10] has written a general introduction to identification, authentication and authorization on the World Wide Web, but does not include a case study. Kabay's paper also reviews commercial products that are used in this context.

A common network authentication service used by many companies is Kerberos (sec. 3.1). Tagg [14] has

[1] `http://www.jasig.org/cas`

written an informative paper describing the details of Kerberos, and how it can be used to provide SSO on a corporate network. Samar [13] has written a paper about cookie-based authentication as an alternative to Kerberos.

Chapter 2

AAA

The three As in computer security are authentication (sec. 2.2), authorization (sec. 2.3) and auditing (sec. 2.4). The three As are crucial parts of any successful network security scheme. This chapter attempts to define the concepts associated with the As and introduce them to the unfamiliar reader. We will also see how they relate to single sign-on (sec. 2.5), but first we need to define identification, because it is important to each of the three As.

2.1 Identification

Before a user can be authenticated, he first have to go through the process of identification.

Definition 1. *Identification is the process that enables recognition of a user described to an automated data processing system. This is generally by the use of unique machine-readable names [10].*

Essentially, it means that the user announces who he is, for example, by providing a username. This is analogous to presenting yourself to someone you have never met before. It is important to understand that the identity can still be fake if, for example, the user is lying. To be sure of a user's identity we need some way to verify it. In the real word we can force a person to show an ID-card. Unfortunately, ID-cards are not very useful in a computer-network setting, but alternatives exist.

2.2 Authentication

Authentication is the process of verifying a provided identity.

Definition 2. *Authorization is a positive identification, with a degree of certainty sufficient for permitting certain rights or privileges to the person or thing positively identified [10].*

In computer systems, an identity need not be tied to an actual person. Instead, the identity can represent a system or a service. Verification can be done in a variety of ways, depending on the context, but all verification methods rely on some *factor*. A factor is an attribute or property that makes it possible to positively identify an individual. Below, we outline different factors.

What the user knows

This factor, *what the user knows*, indicates that we authenticate based on some information that only the user has knowledge of. We can give the user a secret password that can be used to verify the user's identity. Passwords can be assigned and distributed in different ways. For example, the passwords can either be randomly generated or chosen by the user. Based on how the passwords are generated and distributed, you can divide password authentication into different categories, called *password schemes* [9].

What the user has

The second factor, *what the user has*, can be combined with the first factor or used alone. Verifying a user's identity based on this factor means that we authenticate using something the user has in his possession, such as a security token or an ID card.

What the user is

In a computer security context, *what the user is* typically refers to the biometric properties of the user. If the properties are unique each user, it is possible to authenticate based on them. For example, fingerprint scanning, retina scanning and face recognition exploit the fact that certain properties are unique for each person.

Summary

Any of the factors above is sufficient to authenticate a user, and they are all used to varying degrees. Because regular username/password authentication falls into the category of *what the user knows*, it is the most common method [8]. Online banks commonly employ security tokens in their security solutions, which is why *what the user has* is the most common factor in banking [7].

2.3 Authorization

Users who are authenticated can proceed to make requests to the system. However, just because a user is authenticated, it does not mean the user can perform any operation. In order for a request to succeed, the request has to be authorized. What this means is that the request has to be allowed according to some policy or control mechanism. One such mechanism that can be used is an Access Control List (ACL). The ACL specifies what operations users are allowed to perform [4]. ACLs typically associate an object with a list of users and groups. For every combination of object and user, there is a list of operations for each object that the user can perform. Every time a user wants to perform an operation, a lookup is performed in the ACL to see if the operation is allowed.

2.4 Auditing

Auditing is the process of logging activities in the system. For example, failed and successful authentication attempts are often logged. When there is a problem or error, it is often helpful to review the logs in order to determine the root problem. It is possible to detect attacks against the system by analyzing the logs.

2.5 Centralization

When users want to log in to a system or service, they have to go through the process of authentication. If a user wants to authenticate against two different systems, the user needs to authenticate twice. In a larger setting, such as a company or organization, this becomes impractical very fast as the number of services grows, especially if the users have to remember separate usernames and passwords for each system. Another problem is that the management of user accounts becomes complicated as we need several user accounts for each user. For example, if we want to update the address of a user, we could potentially need to update a user record in all of our systems.

The solution is to centralize authentication and authorization. This means that every time a user has to be authenticated or authorized, it is done at a central location. The benefit of centralization is that all the user accounts are stored in one place so there is no unnecessary redundancy. For centralization to work in practice, the services on the network have to be able to delegate authentication and authorization responsibility to a central server. There are different protocols and applications that can be used to delegate responsibility. For network authentication, Kerberos (sec 3.1) is probably the most commonly used protocol. To delegate authorization responsibility, the OAuth protocol can be used [11].

One benefit of using centralized authentication is that it's potentially possible to allow for single sign-on (SSO), so that users who have entered their credentials once do not have to do so again during the session.

Definition 3. *A session is an activity for a period of time; the activity is access to a computer/network resource by a user; a period of time is bounded by session initiation (a form of logon) and session termination (a form of logoff) [10].*

One way to implement SSO is through the use of *tickets*. Users that has been authenticated can make requests for tickets. The tickets enable the users to utilize services. The services can verify the tickets, and thus verify the identity of the user. The requests of tickets and the subsequent transmission of the tickets is done automatically without the user's knowledge. From the end user's perspective, he enters his credentials once and is not bothered again during the session.

One problem with centralization is that it can create a single point of failure. If there is only one server doing authentication and authorization, then failure of that server will cause all services to be inaccessible. Fortunately, most tools that are used for centralization support replication, which enables creation of a second server that contains the same information. When the first server goes down, the second server will take over and continue as if nothing had happened. The second, and probably biggest, problem with centralization is that of retrofitting existing services to support delegation. In some cases it is even impossible, such as, if the source code is no longer available or if the company that originally delivered the code has failed.

Chapter 3

Tools and Protocols

This chapter describes some of the tools and protocols that are commonly used in companies and organizations for authentication and authorization. Section 3.1 describes Kerberos, a network authentication protocol. Other protocols attempt to standardize authentication; they are described in section 3.2. Finally, in section 3.3 we look at LDAP, a directory protocol.

3.1 Kerberos

Kerberos is a computer network authentication protocol developed at MIT. Its name refers to a character from Greek mythology: Cerberus is a three-headed dog responsible for guarding the entrance of the underworld. In a similar way, Kerberos guards a network, allowing entry only to those who are authenticated. In this section we will describe the Kerberos protocol and the concepts associated with it. The goal is to avoid getting bogged down in technical details. Instead, we attempt to provide an overview that will allow us to understand how Kerberos can be used as a central authentication server. Tagg [14] gives a more detailed description.

3.1.1 Terminology and concepts

The first step towards understanding Kerberos is to get a grip on the terminology.

Principals

In Kerberos literature, the term *principal* is used to describe an entity such as a user or a service. Every user, computer or service that needs to be authenticated is associated with a principal. The principals are authenticated based on a long-term key or password that is associated with every principal.

Realm

Each Kerberos installation defines an administrative realm that the principals belong to. In other words, a Kerberos realm is the set of Kerberos principals that are registered within a Kerberos server. The common convention is to use a DNS domain name, converted to upper case, as realm. For example, *helloworld.com* forms the realm HELLOWORLD.COM.

Tickets

Tickets are a very central concept, because they are used to verify identities. The tickets are essentially data constructed in such a way that it becomes possible to authenticate the sender. There are two types of tickets, Ticket Granting Tickets (TGTs) and service tickets. The TGT is a form of master ticket that is used to ask for service tickets, and service tickets are used to access the services that are provided on the network.

Key Distribution Center

The Key Distribution Center (KDC) consists of three logical parts: a database of all principals, an Authentication Server (AS) and the Ticket Granting Server (TGS). A principal who wishes to start a session asks the AS for a TGT. This ticket is then used to ask for service tickets that can be sent to individual services. The TGT is encrypted with the principal's key so that it is not usable by anyone else. It is important to note that the Ticket Granting Server (TGS) and the Ticket Granting Ticket (TGT) are two completely different things.

3.1.2 Putting the pieces together

We will now study the Kerberos protocol to see how users can authenticate themselves to services on the network.

Kerberos actually ensures mutual authentication, meaning that the service is authenticated to the user as well. Figure 3.1 shows the steps involved when a client authenticates with a service; below we outline the steps in more detail.

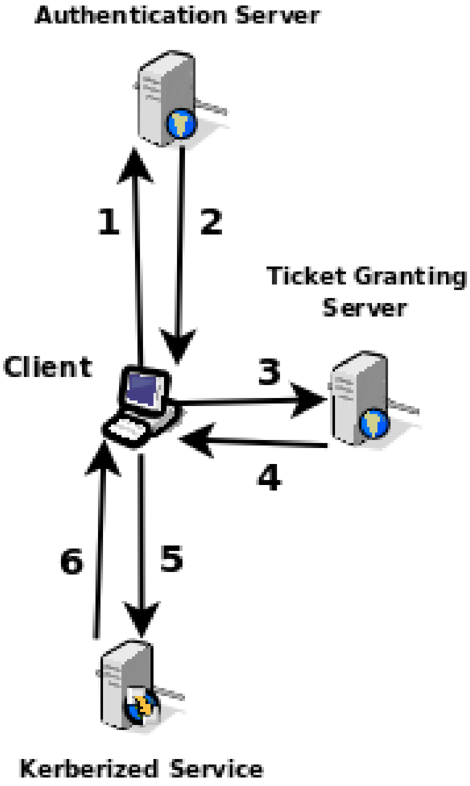


Figure 3.1: Kerberos authentication sequence

Step 1

The client begins the authentication process by requesting a TGT from the AS.

Step 2

The AS returns the TGT. Because the ticket is encrypted with the client's password, the ticket is not usable by anyone else. This means that the ticket can be sent over an unencrypted channel.

Step 3

The client sends the TGT to the TGS to request a service ticket for a specific service.

Step 4

When the TGS receives the TGT, it verifies the TGT and returns a service ticket to the client.

Step 5

The client sends the service ticket it received in step 4 to the service in order to begin a session.

Step 6

The service verifies the ticket and sends back a message encrypted with the session key. Because the ticket is encrypted with the session key, the client can verify the identity of the service.

3.1.3 Kerberization

Refitting existing applications and services to use Kerberos is a common task; in fact the task is so common that it has become a verb. To add Kerberos support to an application is called to "Kerberize". Specifically how this is done depends on the particular application. The easiest way is to use one of the standardized protocols or APIs. Alternatively, if a customized protocol is used, it may be possible to fit Kerberos into the customized protocol framework.

3.2 Standardized authentication

Many protocols and APIs attempt to standardize authentication. In this section we will present some of them and give brief descriptions. The focus will be on protocols and APIs that can interact with Kerberos.

3.2.1 GSS-API

GSS-API is an abbreviation for Generic Security Services Application Program Interface. This API provides security services to application programmers. GSS-API makes it possible to call security services in a standardized way. It also shields the programmer from the potential complexities of the underlying protocol. Most Kerberos implementations provide support for GSS-API, which means that all programs that support GSS-API also support Kerberos. The messages sent in a Kerberos session are encapsulated by GSS-API before being sent over the network.

3.2.2 SPNEGO

The biggest problem GSS-API faces is that the applications must agree beforehand what protocol to use. SPNEGO, which is an abbreviation for Simple and Protected GSSAPI Negotiation Mechanism, sets out to solve this problem. Using SPNEGO, a client and server can jointly determine what protocol to use.

The server sends the client a list of supported protocols and the client picks the strongest protocol it can use. SPNEGOs greatest benefit is that it is supported by all major web-browsers. By using an SPNEGO enabled web server and Kerberos it becomes possible to Kerberize web services. This method can be used to integrate web-services into a single sign-on framework, something that is otherwise very difficult.

3.2.3 PAM

Pluggable Authentication Modules is an API that allows integration of several low-level authentication schemes. By defining a standard interface for authentication, different applications and authentication methods can easily be integrated. The biggest problem with PAM is that it only supports username/password authentication, which means that it's not possible to authenticate Kerberos tickets using PAM. It is, however, possible to authenticate against a Kerberos server if a user has the principal name and password. Therefore, PAM can be used to provide a fallback method for users that do not have Kerberos tickets but do have a valid principal name and password.

3.2.4 SASL

SASL stands for Simple Authentication and Security Layer, which is a generic protocol framework for doing various sorts of authentication between a client and a server. The idea is to separate the authentication mechanism from the application protocol. What this means is that the application protocols specify an "application profile" that describes how to encapsulate SASL messages. Within the SASL framework different authentication mechanisms may be used, such as:

- DIGEST-MD5
- OTP One-time Passwords
- Kerberos
- GateKeeper (Microsoft)

SASL is more generic than PAM. For example SASL supports Kerberos mutual authentication using tickets. Note that application programmers need to explicitly add support for all underlying mechanisms that they want to utilize, so in order to add Kerberos support to an application using SASL the programmer needs to explicitly add support for GSS-API.

3.2.5 Summary

It is easy confuse all the different protocols and it is sometimes difficult to understand their purposes. In table 3.1 we provide a summary intended to make it easier to understand the relationships between the different protocols and APIs.

Name	**Purpose**
Kerberos	Network authentication protocol.
GSS-API	Generic authentication protocol, supports Kerberos as underlying mechanism.
SPNEGO	Enables GSS-API to be used over the web.
PAM	Generic API for authentication using username/passwords.
SASL	Similar to PAM, allows more complex mechanisms.

Table 3.1: Summary of tools and protocols

3.3 LDAP

An LDAP server is a directory server; its most common use is as an enterprise management tool [6]. LDAP is an abbreviation for Lightweight Access Directory Protocol. An LDAP server can be used as a central directory for users, groups and accounts, which stores information such as usernames, passwords, real names, telephone numbers and email addresses.

3.3.1 History

At its core LDAP is a network protocol designed to retrieve and store data in a X.500 directory (a directory server architecture, designed and standardized in the 1980s). The original LDAP servers were thus just gateways to the X.500 directories [6]. Today, however, LDAP has grown into a directory service in its own right, and now includes the standards needed to operate as a full-fledged directory server.

3.3.2 Directories

A directory can, for example, be a phonebook or a catalogue. Because a phonebook is organized in alphabetic order, it may be searched efficiently. A digital directory is not very different from its analogue cousin: the purpose of the directory is to organize information and make it searchable. It should be possible to add and remove information, but it is assumed that modification occurs less frequently than searching. A directory server contains one or more directories, and is optimized so that searching is very effective [6]. A typical entry in a company personnel directory might look something like this:

- Name: Joe Average
- Position: Vice President
- Street: Example-street 112
- City: New York
- Postal Code: 23423-3433
- Phone Number: 555-123321
- ObjectClass: person

3.3.3 Distinguished Names

To be able to uniquely identify an entry, we need a way to distinguish it from the other entries in the directory. This is accomplished by setting a Distinguished Name (DN), which is a concept similar to a key in a relational databases. The DN is a set of attributes associated with the entry that allows us to uniquely identify the entry. The ObjectClass attribute in the example above is similar to a type declaration and specifies some attributes that the entry must include. Here is an example of a DN: *CN=Peter, OU=Sales, DC=Example, DC=COM*

3.3.4 Directory Information Trees

Entries in a directory are organized into hierarchies. Consider an example company Acme. At the top of the hierarchy is the company itself, below are the different departments such as Accounting and Marketing, and under the departments are the employees working in the departments. In Figure 3.2 we can see an example directory.

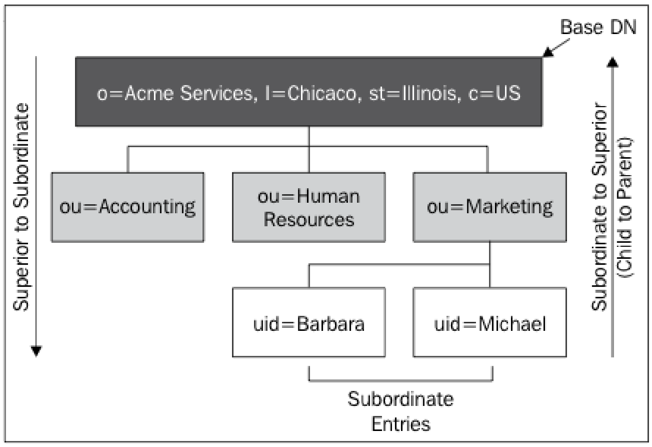


Figure 3.2: Sample Directory Information Tree

3.3.5 LDAP authentication

In order for a user to interact with an LDAP server, the user must first be authenticated; this action is called *binding*. LDAP supports two methods of binding, simple binding and SASL binding. Simple binding is exactly what it sounds like, very simple to set up and to administer. Unfortunately, simple binding is not very secure. With Simple Binding, the DN and password are sent in plaintext over the network [6]. Sometimes the LDAP server is configured to allow for anonymous binding, which makes it possible for anyone to view information without binding first. To allow for more secure communication, most LDAP implementations allow for traffic encryption using SSL/TLS. Because LDAP includes support for SASL, LDAP also supports every authentication mechanism that can be used with SASL.

Chapter 4

Case study

This chapter contains a case study of authentication and authorization centralization. It describes how single-sign on was enabled at a company called Kreditor. Some fundamental information about Kreditor and Kreditor's environment is given in section 4.1 and 4.2. The requirements from Kreditor are described in section 4.3. The design that was developed to meet the requirements is outlined in section 4.4. Finally, section 4.5 contains implementation details and solutions to some of the problems that were encountered.

4.1 Kreditor

Kreditor is a company offering financial services over the internet. Two services offered are *fakturera mig* and *dela upp betalningen*[1]. The services allow web shops to delegate credit payments from their customers to a third party, namely Kreditor. Kreditor is growing very fast, in the last year alone they have increased from 70 to 130 employees[2]. As the number of users has increased, so has the effort required to manage the users. Internally, Kreditor use a Kerberos (sec. 3.1) server for network authentication and a LDAP (sec. 3.3) server to organize their employees.

At the heart of Kreditor is the *kred system* that provides Kreditor's customers with services, and allow Kreditor's employees to do their daily jobs. In subsequent text the kred system will be referred to as simply *kred*. The problem with kred is that it doesn't use the existing infrastructure for authorization and authentication; instead, it stores separate user accounts. If kred could be integrated with Kerberos and the directory server, user management would be simplified and security increased (sec. 2.5).

4.2 Environment

Kred is written in Erlang, a functional concurrency oriented programming language. The system runs on a master/slave pair architecture to improve fault tolerance and availability. Kreditor has two LDAP servers: `ldap1.internal.machines` and `ldap2.internal.machines`. Kreditor also has one Kerberos server: `lithium.internal.machines`. Users get TGTs from the Kerberos server when they login to their computers. Information about employees and groups is stored in the directory server, queries can be sent to the directory server to retrieve this information.

4.2.1 Erlang

The development of Erlang began in 1986 at the Ericsson Computer Science Laboratory to design a language to program telecom switches. Erlang was designed for writing concurrent programs that "run forever" [2]. The requirements were virtually zero downtime and to allow for hot code swapping, upgrading running code. Erlang meets these by structuring programs using concurrent processes that have no shared memory and use asynchronous messages. The requirements of a telecom switch and a financial system are very similar, which is why Erlang suits Kreditor well. A good book we recommend to anyone who wants to learn Erlang is the introductory book by Armstrong [3].

4.2.2 Yaws

Kred uses a web interface to allow its employees and customers to interact with the system. The web pages are served Yaws, a high performance web server written in Erlang. The main benefit of Yaws is that it scales extremely well. In Figure 4.1 we can see a comparison of

[1] `http://kreditor.se/tjanster-las-mer/index.html`

[2] `http://computersweden.idg.se/2.2683/1.219743`

Yaws versus Apache[3]. On the horizontal axis is the number of sessions and on the vertical axis is the throughput. Yaws is the red curve while the green and blue curves show different Apache configurations. On the graph we can see that Apache simply dies at around 4000 sessions while Yaws can handle over 80 000 parallel sessions.

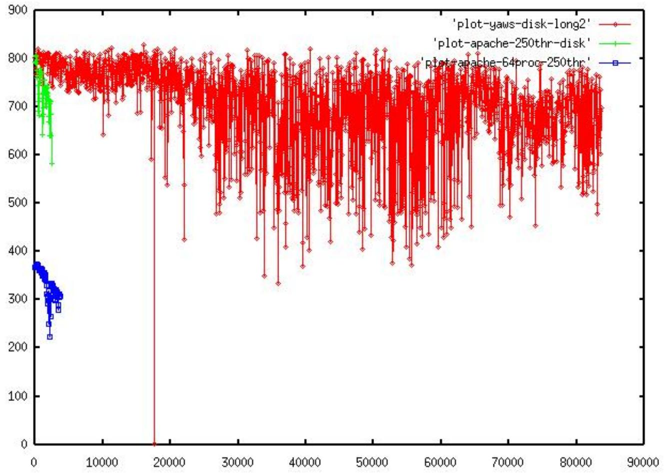


Figure 4.1: Apache vs Yaws comparison

Yaws authentication

Yaws supports quite a few authentication mechanisms, including username/password, SPNEGO and PAM. It is also possible to write customized authentication modules to add new authentication mechanisms. The version of Yaws that was available at the start of this project was 1.81, which does not allow for different authentication methods to be used concurrently.

Yaws uses the methods embedded in the HTTP protocol, the protocol used to transfer web pages from the web server to the client, to perform authentication. First the client sends a request; when Firefox makes a request it looks similar to the contents of Table 4.1. The server response can be seen in Table 4.2. The first 5 lines of the response are HTTP headers, the headers are followed by the content that the web-browser should display.

Marked in bold is the HTTP status, used to indicate the result of the request. For this request the status is 200, which means that the request was successful. The web server can use the status to indicate that the client needs to authenticate its identity; this is done by setting the status to 401. The web server also includes a list of alternatives that the client can use to authenticate its identity, they are indicated by a preceding ”WWW-Authenticate”.

Sample browser request
GET /example-page.html HTTP/1.1
Host: example.com
User-Agent: Firefox/3.0.11
Keep-Alive: 300
Connection: keep-alive

Table 4.1: Stripped down version of Firefox GET request

Yaws OK response
HTTP/1.x **200** OK
Server: Yaws/1.81 Yet Another Web Server
Date: Sat, 04 Jul 2009 13:21:35 GMT
Content-Length: 57760
Content-Type: text/html
<html> .... </html>

Table 4.2: Yaws ”OK” response

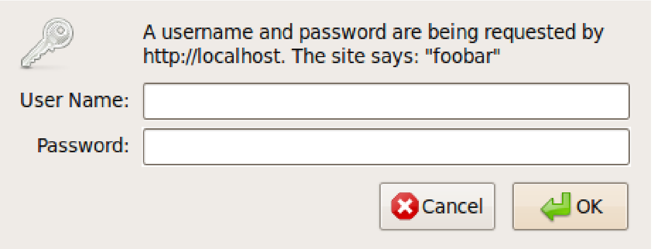
Yaws unauthorized response
HTTP/1.1 **401** Unauthorized
Server: Yaws/1.81 Yet Another Web Server
Date: Sun, 22 Mar 2009 03:47:47 GMT
Content-Length: 63
Content-Type: text/html
WWW-Authenticate: Basic realm=””
<html>
<body>
<h1>401 authentication needed</h1>
</body></html>

Table 4.3: Yaws ”unauthorized” response

When the web-browser sees that the status is 401, it attempts to authenticate using the alternatives available. If the web-browser fails, the content will be rendered to the user, in this case the browser will render a *401 authentication needed* message. When Yaws attempts to authenticate using a method that requires a username and password, it includes the header *WWW-Authenticate: Basic realm=””*. This will make the web-browser display a password prompt where the user can enter his username and password. A sample of what this looks like in Firefox can be seen in Figure 4.2. Once the user has entered his credentials, the credentials are added to the HTTP headers

[3] http://www.sics.se/ joe/apachevsyaws.html

and sent back to the web-server.



A username and password are being requested by http://localhost. The site says: "foobar"

Figure 4.2: Firefox password window

When Yaws is configured to authenticate using SPNEGO the header sent to the client is *WWW-Authenticate: Negotiate*. When the browser sees the header it will attempt to fetch a service ticket from the Kerberos server for the principal HTTP/domain@realm. For example, if the domain is example.com and the realm is COMPANY, then the browser will attempt to fetch a service ticket for the principal HTTP/example.com@COMPANY.

Yaws authorization

Yaws has no explicit notion of authorization, if a user is authenticated then he is essentially authorized. This means that if a more sophisticated authorization scheme is needed, the developer will have to design it himself.

4.2.3 Mnesia

Kreditor uses Mnesia as their backend database, Mnesia is a distributed DataBase Management System (DBMS) written in Erlang. It was developed at Ericsson to be used in soft real-time, distributed, and high-availability applications. Mnesia was designed to solve the typical data management problems of telecommunications applications. Data is stored in tables similarly to relational databases, with the difference that the data is schema free. The data is stored as {Key, Value} pairs and lookups are done based on the key. The values can be of any format.

4.2.4 Kerberos

The Kerberos realm at Kreditor is called `internal.machines`, it runs on the MIT Kerberos implementation[4].

4.2.5 LDAP

Kreditor has two LDAP servers to achieve redundancy. If one server crashes, the other server will remain operational. Because the second server is a replication of the first server, the content on the two servers is identical. The base of the directory is *DN=INTERNAL.MACHINES*. Under the base are *OU=People* and *CN=Groups*. Employees are placed under *People*, while the different departments are placed under *Groups*. For example, the employee John might have the DN *UID=john,OU=People,DN=INTERNAL.MACHINES*. Employees can become members of groups by having their DNs added to the *uniquemember* field of the group. An illustration of the directory layout can be seen in Figure 4.3.

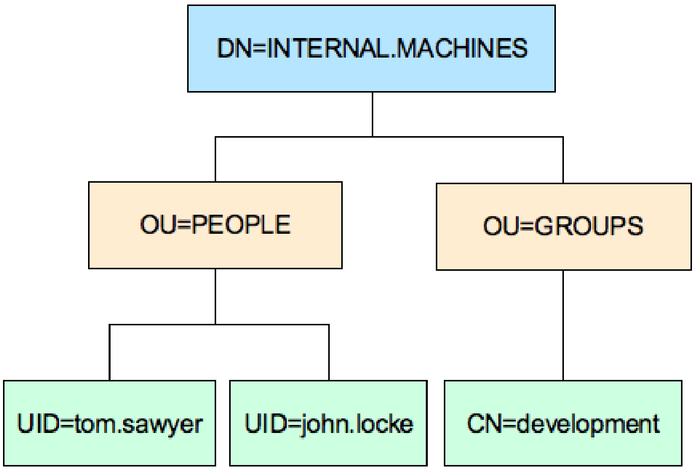


Figure 4.3: Kreditor LDAP layout.

4.3 Specification

The requirements from Kreditor were loosely formulated. The core requirement was integration of authentication and authorization in kred with the existing network infrastructure. SSO was desirable, but not absolutely necessary, which allowed more freedom in the design of the solution. The goal for the design was to allow for SSO and to capture the advantages of SSO while mitigating the disadvantages.

[4] http://web.mit.edu/Kerberos/

4.4 Design

In this section we describe the design that was presented to Kreditor. The design uses the existing Kerberos and LDAP servers at Kreditor to perform authentication and authorization. The solution has been implemented and tested. Below we will describe how the different goals of centralized authentication, authorization and SSO were achieved.

4.4.1 Authentication

Since Yaws already included support for SPNEGO, the protocol was quickly included in the design, which meant the design allowed for SSO from the start. However, Kreditor required that there be a fallback method so that users without access to the Kerberos server could still login using their username and password. One potential solution to this problem was to use the PAM support in Yaws, together with a Kerberos PAM module, to authenticate usernames and passwords. A flowchart of the full authentication process used in this design can be seen in Figure 4.4.

One problem with this design was that Yaws version 1.81 did not allow for multiple authentication methods to be used concurrently. However, since Yaws is open-source, it was possible to write a patch that would add this feature. Writing the patch would also give an opportunity to get acquainted with Erlang.

Another problem was related to the fact that the PAM support in Yaws uses HTTP Basic authentication. With Basic authentication, the client authenticates its identity using a username and password. The web browser asks the user for a username and password and then sends a new request that includes this information in the HTTP headers. If the username and password is correct, the web server responds with status 200 and the requested content.

The problem is that the prompt that the web browser displays to the user does not allow feedback to be passed back to the user. A better approach is to use a login page to provide user feedback and allow for an overall more pleasant experience. However, in Yaws 1.81 it was not possible to change the content shown with HTTP status 401. This feature would have to be added as well. Sequence diagrams of how SPNEGO and PAM authentication works can be seen in Figure 4.5 and 4.6 respectively.

To encrypt all the traffic and mitigate the disadvantages of Basic authentication, all traffic is sent using Secure Socket Layer(SSL). The fallback method is less se-

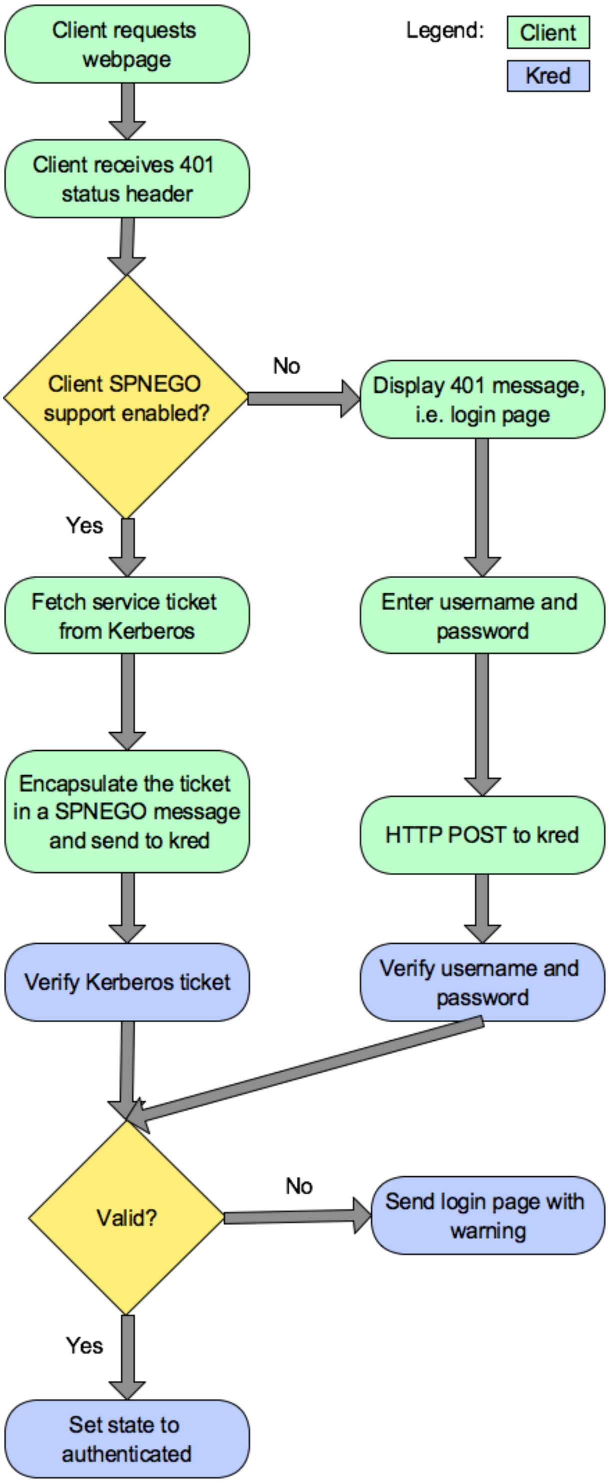

Client requests webpage
Legend:
Client
Kred
Client receives 401 status header
Client SPNEGO support enabled?
No
Display 401 message, i.e. login page
Yes
Fetch service ticket from Kerberos
Enter username and password
Encapsulate the ticket in a SPNEGO message and send to kred
HTTP POST to kred
Verify Kerberos ticket
Verify username and password
Valid?
No
Send login page with warning
Yes
Set state to authenticated

Figure 4.4: Flowchart of the login process.

cure since the username and password has to be sent every time a user wants to authenticate themselves to kred.

4.4.2 Authorization

In Yaws there is no real notion of authorization separate from authentication: if a user is authenticated then he is authorized. This means that in order to get a more flexible authorization scheme, a developer will have to add it. Because kred already had a working authorization scheme, it could be extended to integrate with the LDAP server. Authorization in kred is based on the information stored in four different Mnesia tables called *kuser*, *kuser_level*, *kuser_group* and *kuser_page*. The finest granularity that authorization can be based on in kred is at the kuser_page level. Kuser_pages represent the pages in the GUI and are stored as Erlang atoms[5], but could in theory represent any item or concept. Each kuser_page is mapped to at most one kuser_group. The kuser_groups on the other hand can map to more than one kuser_page.

The kuser_groups have an attribute called *bit* that is used to uniquely identify the group. Since Erlang supports arbitrarily large integers, there is no limit to the number of groups. The next level of granularity is at the kuser_levels layer, kuser_levels map to one or more kuser_groups. Kuser_levels have the ability to inherit groups from other kuser_levels, which means that kuser_levels can easily be extended. The kuser_levels also have a bit number used to uniquely identify the level.

Finally, each user in kred is associated with a kuser record. The kuser records contain attributes such as username, password and the kuser_levels that the user has access to.

[5] An Erlang atom is essentially a named constant.

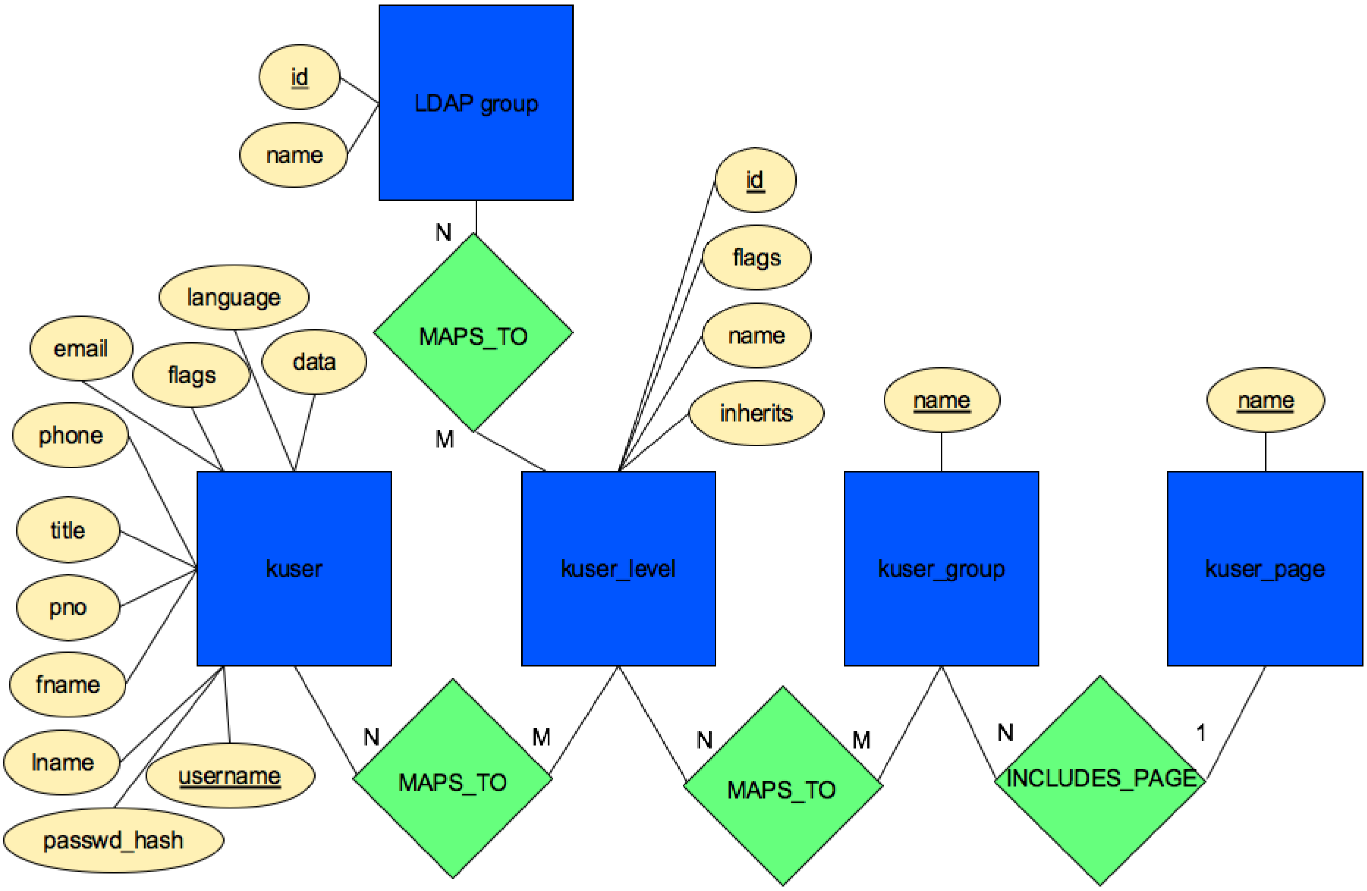


Figure 4.7: ER diagram the authorization scheme in kred.

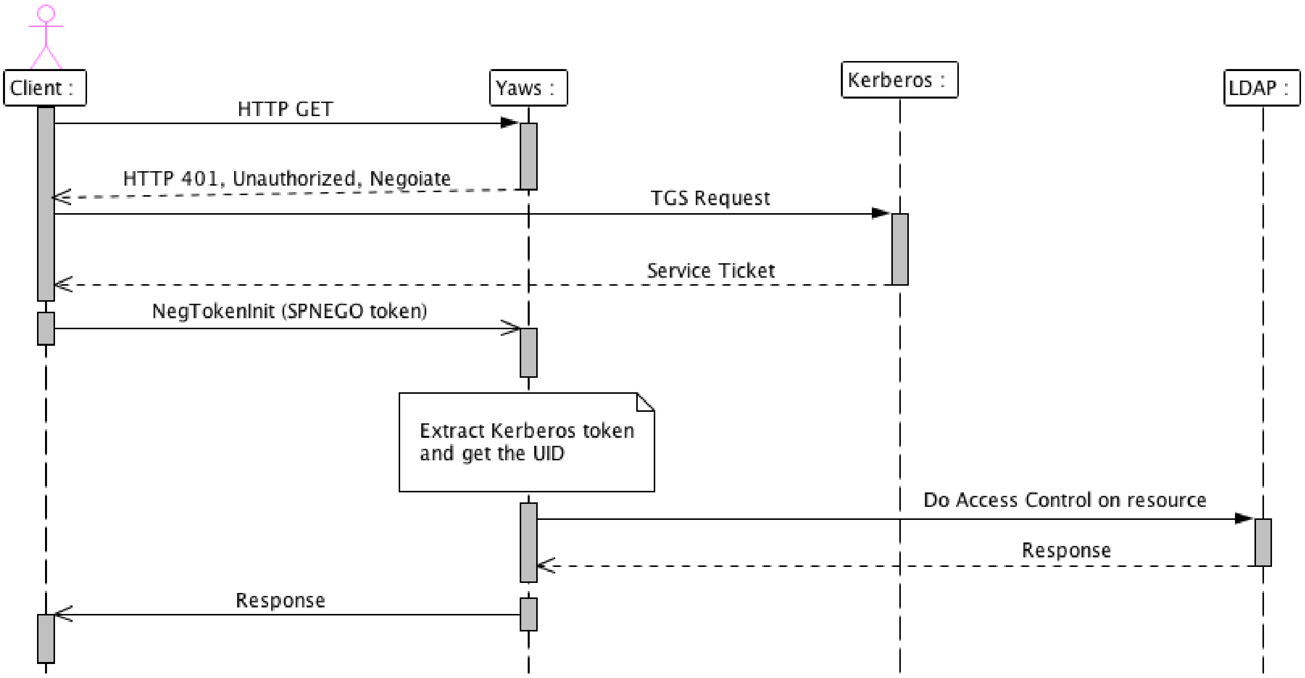

Client :
Yaws :
Kerberos :
LDAP :
HTTP GET
HTTP 401, Unauthorized, Negoiate
TGS Request
Service Ticket
NegTokenInit (SPNEGO token)
Extract Kerberos token and get the UID
Do Access Control on resource
Response
Response

Figure 4.5: SPNEGO authentication in kred.

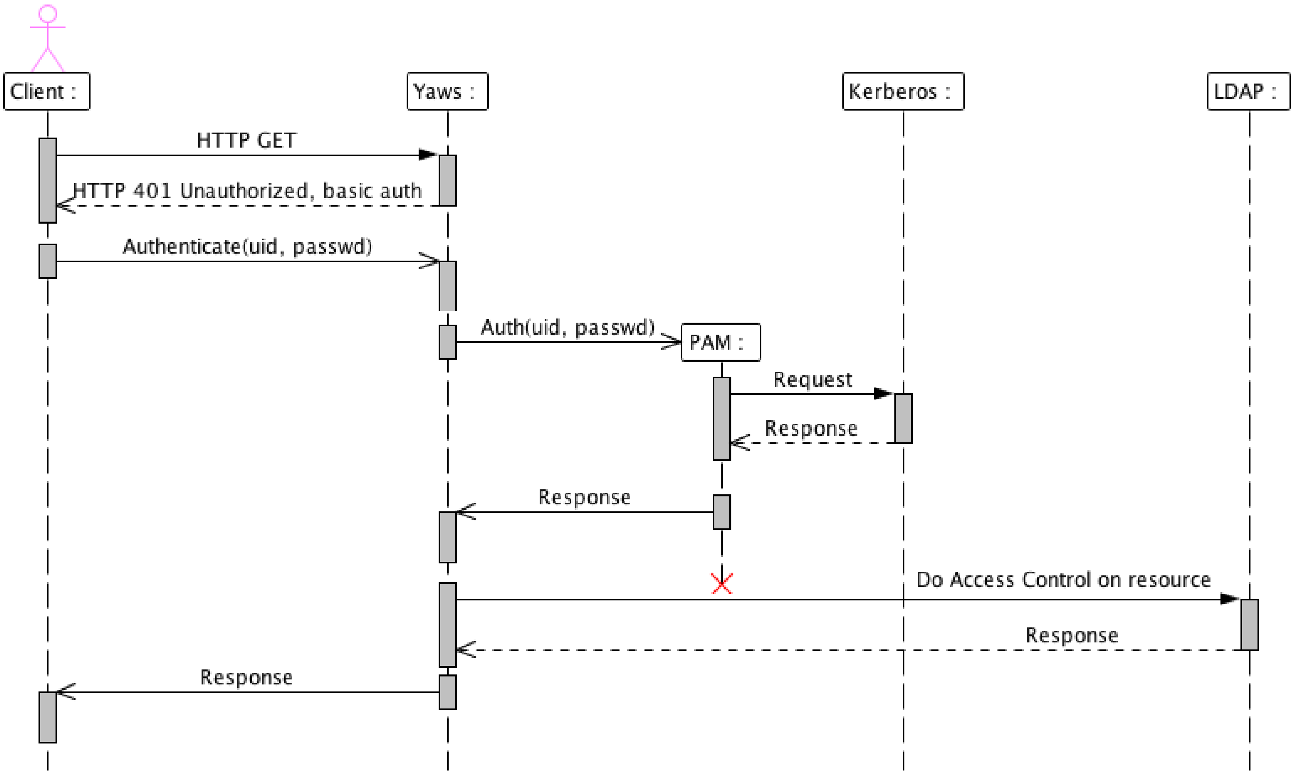


Figure 4.6: username/Password authentication in kred using PAM.

An entity-relationship diagram with the relationships between the tables can be seen in Figure 4.7. The kuser_level has an attribute called *perms*; this is a bitmap where the groups that map to that level have their ID bit set to 1. Kusers are mapped to kuser_levels using the same method.

To illustrate how this works, we will outline what happens when a user wants to visit a page called *sample_page.yaws*. First the user has to be authenticated, which is done by matching a username and password against the entries in the kuser table. Every user session is associated with a site state that is stored in kred. The site state contains the permissions of the user. If authentication is successful then a bitmap with the group IDs that the user has access to is built and stored in the site state.

First all the kuser_levels that the user has access to are retrieved, then all the groups that are mapped to the levels. The ID bits of the groups the user has access to are set to 1 to construct a bitmap, which is then stored in the site state. When the user visits *sample_page.yaws*, that page is mapped to a group. If the bit associated with that group is 1 in the bitmap, then authorization is granted.

The easiest way to extend this scheme, with the goal of integration with the LDAP server, would have been to add an attribute to kuser_level that maps a kuser_level to a LDAP group. However, to update the kuser_level table, the table would have to be locked down. On a live system this is very problematic. Instead, the solution proposed was to add a new table called kuser_ldap_map that would map a kuser_level to a set of LDAP groups. This relation is included on Figure 4.7.

4.4.3 Single sign-on

Adding SSO proved to be an easy task because Yaws included support for SPNEGO. When the web browser sees *WWW-Authenticate* in the HTTP headers, it automatically fetches a Kerberos ticket. For this to work, SPNEGO must be enabled in the web-browser. Instructions on how to enable SPNEGO in Firefox and Internet Explorer can be found in Appendix A.

4.5 Implementation

The implementation proceeded smoothly and took 7-8 weeks. The first step was to set up a test server that could be used to test the patch needed for Yaws. Setting up Yaws, Kerberos and Pam took longer than expected, Kerberos especially was difficult to set up. When the patch was finished, Yaws had the features needed to implement the design.

One interesting problem was that Firefox does not seem to pick the strongest authentication method provided in the HTTP headers. Instead Firefox seems to always pick the first method provided, regardless of what other methods are available.

The patch for Yaws was officially released with Yaws version 1.82 and the details can be found at GitHub[6]. GitHub is a web-based hosting service for projects that use the Git revision control system.

The next problem was to run kred with the new version of Yaws. The version of Yaws used in kred was a modified version of 1.52. Upgrading to 1.82 introduced some bugs that had to be fixed before the system could be fired up. After some tweaking, both SPNEGO and PAM authentication worked.

Authorization required less time to implement. The implementation follows the design described in section 4.4.2. Some Erlang libraries that can be used for authentication and authorization are briefly described in Appendix B. The Eldap library was used to communicate with the LDAP servers at Kreditor. Because of the way authorization was designed, the information stored in the LDAP server is not modified by kred.

4.5.1 Optimization

One problem with the initial implementation was that constructing a list of all users was extremely slow, but required for the GUI administration page. The first naive implementation took roughly 14 seconds to return a list of all users. Two optimizations were implemented that drastically improved performance.

The first problem with the naive implementation was that the kuser records were populated in a very inefficient way. The result from a LDAP query is returned in the form of a property list: `[{property1, value1},...,{propertyN, valueN}]`. The naive version simply conducted a simple lookup of the property it was adding to the kuser record. This meant that for every attribute, the entire property list could potentially be traversed. This gives a complexity of $O(n * m)$, where n is the number of attributes in the record and m is the length of the list returned from the LDAP query. The algorithm was changed so that the property list is traversed

[6] `http://github.com/klacke/yaws/commit/c1cea992847be923ddce1338c115a0f820755f62`

once and the record attributes are set as they are found. Since setting an attribute in a record is a constant-time operation, populating a record now had a complexity of m, where m is the length of the attribute list. This modification cut the total retreival time down from 14 to 9 seconds, a reduction of 35%.

The second problem was that, because user entries were retrieved sequentially, there was a lot of network overhead. The retrieval was re-factored so that all entries are fetched in one query. The entries are then put in an Erlang dictionary for constant lookup. The dictionary is used for the cross matching of LDAP group membership and kuser_level membership to construct the access bitmap. This cuts the total computation time down further to 1.4 seconds, thus giving a total reduction of 90% compared to the initial implementation. In Figure 4.8 we can see a comparison of the three different revisions. One improvement that could further improve the performance would be to parallelize the population of the kuser records.

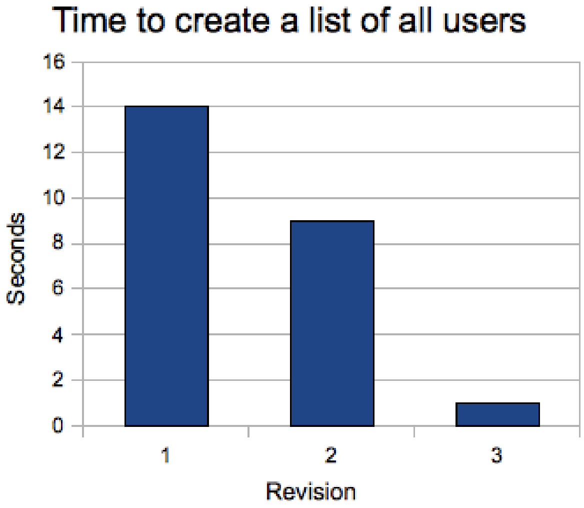


Figure 4.8: Chart of time to create list of all users.

4.5.2 Mapping of usernames

Another problem was that the usernames in the kuser table and the LDAP directory were not exact matches. While most usernames in both cases were of the form *first-name.last-name*, there were some exceptions. Unfortunately, renaming the usernames was not an option because the usernames were linked to entries in the audit logs. That link would have been lost if the usernames were renamed. Instead, every username in the directory was mapped to their Kreditor Online username.

4.5.3 Character encoding

The next problem was that the data from the OpenLDAP server is sent encoded in UTF-8 while Erlang primarily uses Latin-1. UTF-8 is an encoding format of Unicode, which is itself an attempt to standardize character representation. The primary advantage of UTF-8 is that it keeps compatibility with ASCII by using one byte to represent the most common characters. Unicode support is not one of Erlang's strong points, even though it has been improved in the latest version. The character encoding problem was solved by first converting the data into Unicode and then from Unicode to Latin-1.

4.5.4 Redundancy

The new authentication scheme introduced a single point of failure, which had to be mitigated. Kerberos is set up so that it employs two servers. If kred cannot connect to the first Kerberos server, it attempts to connect to the second server. If kred fails to connect to the second server, an alarm will be sent out. There are two LDAP servers, kred tries to connect to both before giving up and sending out an alarm.

4.5.5 Testing

Kreditor use integration testing to enhance the quality of their code. Two test suites were developed to test the authentication and authorization code. One test suite called *ldap_SUITE*, tests all exported functions in the LDAP module such as retrieving and populating kuser records. The second test suite, *gui_test_SUITE* uses the HTTP library in Erlang to test the web-interface. The outputted web-page is not parsed to check the content, but the test ensures that the server does not crash during the request.

Chapter 5

Conclusion

Adding SSO to an existing network is a very complex problem. It is also a problem that has not been comprehensively studied. Perhaps more studies that examine the costs and potential savings of adding SSO are needed. The few cost estimations that exist indicate that there are potentially huge savings for companies that implement good solutions [5] [16].

In this paper we have shown a case study at a company called Kreditor where SSO was added to an existing network. More specifically, a service written in Erlang was modified to use existing network infrastructure for authentication and authorization. The implementation was successful and the requirements from Kreditor were met. We present a design that can be used to provide SSO for web services, by integrating the web service with a Kerberos server. We also show how to provide a fallback method for when the user does not have a Kerberos ticket. The design also gives an example of how a set of usernames and passwords pairs can be integrated with a LDAP directory.

Some of the problems encountered during the design and implementation are described. Problems associated with optimization, username mapping, character encoding and redundancy, are likely to be encountered when attempting to integrate a similar service. Despite the problems, the move from design to implementation was successful.

Because the service at Kreditor was written in Erlang, the implementation details of this case study are particularly interesting. Erlang is a functional programming language that is becoming more and more popular[1]. Working with Erlang has been a very pleasant experience. Libraries exist for most needs, but they can sometimes be hard to find, unless you know exactly what you're looking for. Appendix B lists some Erlang libraries that can be used for authentication and authorization.

5.1 Improvements

While the design in this thesis was an improvement on the existing situation, there is still room for improvement. What seems to be the biggest problem is that the authorization scheme is very closely tied to the pages in the GUI. A better solution would be to tie access to the actual resources in the system. Overall the authorization decisions are too coarse-grained to fit a medium-sized company. By using the resources in the system as a base for authorization decisions, it would be possible to have different access rules for different operations. This would, for example, make it possible to give one person read access to a resource while another person is given write access. Separating out the authorization logic should also make the upcoming redesign of the GUI easier.

[1] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Bibliography

[1] Fredrik Ahlborg and Johnny Hålsjö. A study in the use of single sign-on in swedish public sector. 2008.

[2] Joe Armstrong. A history of erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[3] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, July 2007.

[4] John Barkley. Comparing simple role based access control models and access control lists. In *RBAC '97: Proceedings of the second ACM workshop on Role-based access control*, pages 127–132, New York, NY, USA, 1997. ACM.

[5] Kris Brittain and Gartner Group, 2004. What Is the Right IT Service Desk Staff Size and Structure?

[6] Matt Butcher. *Mastering OpenLDAP: Configuring, Securing and Integrating Directory Services.* Packt Publishing, 2007.

[7] Joris Claessens, Valentin Dem, Danny De Cock, Bart Preneel, and Joos Vandewalle. On the security of today's online electronic banking systems. *Computers & Security*, 21(3):253 – 265, 2002.

[8] Jason Garman. *Kerberos: The Definitive Guide.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[9] Dieter Gollmann. *Computer security.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[10] M. E. Kabay. Identification, authentication and authorization on the world wide web. `http://www.mekabay.com/infosecmgmt/iaawww.pdf`, 1997.

[11] Pauli Kaila. From end-to-end to trust-to-trust. In *PROCEEDINGS OF THE SEMINAR ON NETWORK SECURITY*, pages 18–22, 2008.

[12] NAC. Enterprise-wide security: Authentication and single sign-on. `http://www.alameda-tech-lab.com/portfolio/samples/Old_Papers/NACSEC02.pdf`, 1996.

[13] Vipin Samar. Single sign-on using cookies for web applications. In *WETICE '99: Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, pages 158–163, Washington, DC, USA, 1999. IEEE Computer Society.

[14] Gary Tagg. Implementing a kerberos single sign-on infrastructure, 2000.

[15] A. Volchkov. Revisiting single sign-on: a pragmatic approach in a new context. *IT Professional*, 3(1):39–45, Jan/Feb 2001.

[16] Roberta Witty, Kris Brittain, Ant Allan, and Gartner Group, 2004. Justify Identity Management Investment With Metric.

[17] Si Xiong. Web single sign-on system for wrl company. Master's thesis, KTH Royal institude of Technology, June 2005.

[18] J. Yan, A. Blackwell, R. Anderson, and A. Grant. Password memorability and security: empirical results. *Security & Privacy Magazine, IEEE*, 2(5):25–31, 2004.

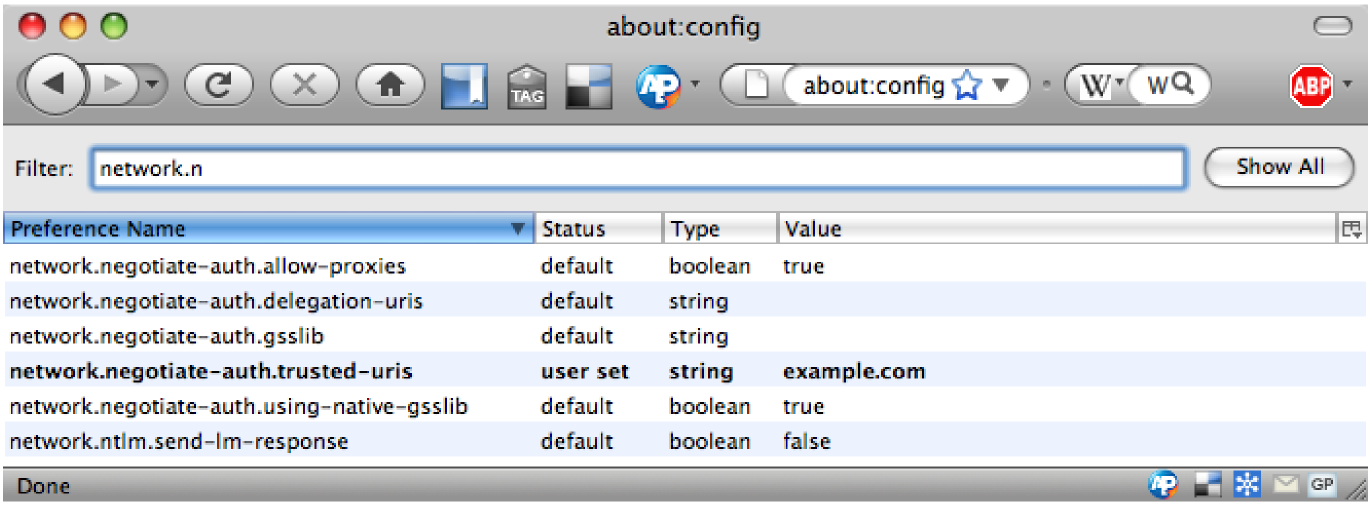
Appendix A

Client Setup

All major browsers support SPNEGO, but it's typically not enabled by default. This appendix describes how to enable SPNEGO for Mozilla Firefox and Microsoft Internet Explorer. The clients need to hold TGTs for SPNEGO to work. The web-server also needs to be configured correctly.

A.1 Mozilla Firefox

To enable SPNEGO in Firefox you have to change the advanced settings, they can be found by typing *about:config* in the address field of the browser. Ignore the warning and change the setting *network.negotiate-auth.trusted-uris* to the name of your domain, for example, *acme.com*. Figure A.1 shows the advanced configuration window.



Filter: network.n

Preference Name	Status	Type	Value
network.negotiate-auth.allow-proxies	default	boolean	true
network.negotiate-auth.delegation-uris	default	string	
network.negotiate-auth.gsslib	default	string	
network.negotiate-auth.trusted-uris	**user set**	**string**	**example.com**
network.negotiate-auth.using-native-gsslib	default	boolean	true
network.ntlm.send-lm-response	default	boolean	false

Figure A.1: Enabling SPNEGO in Firefox.

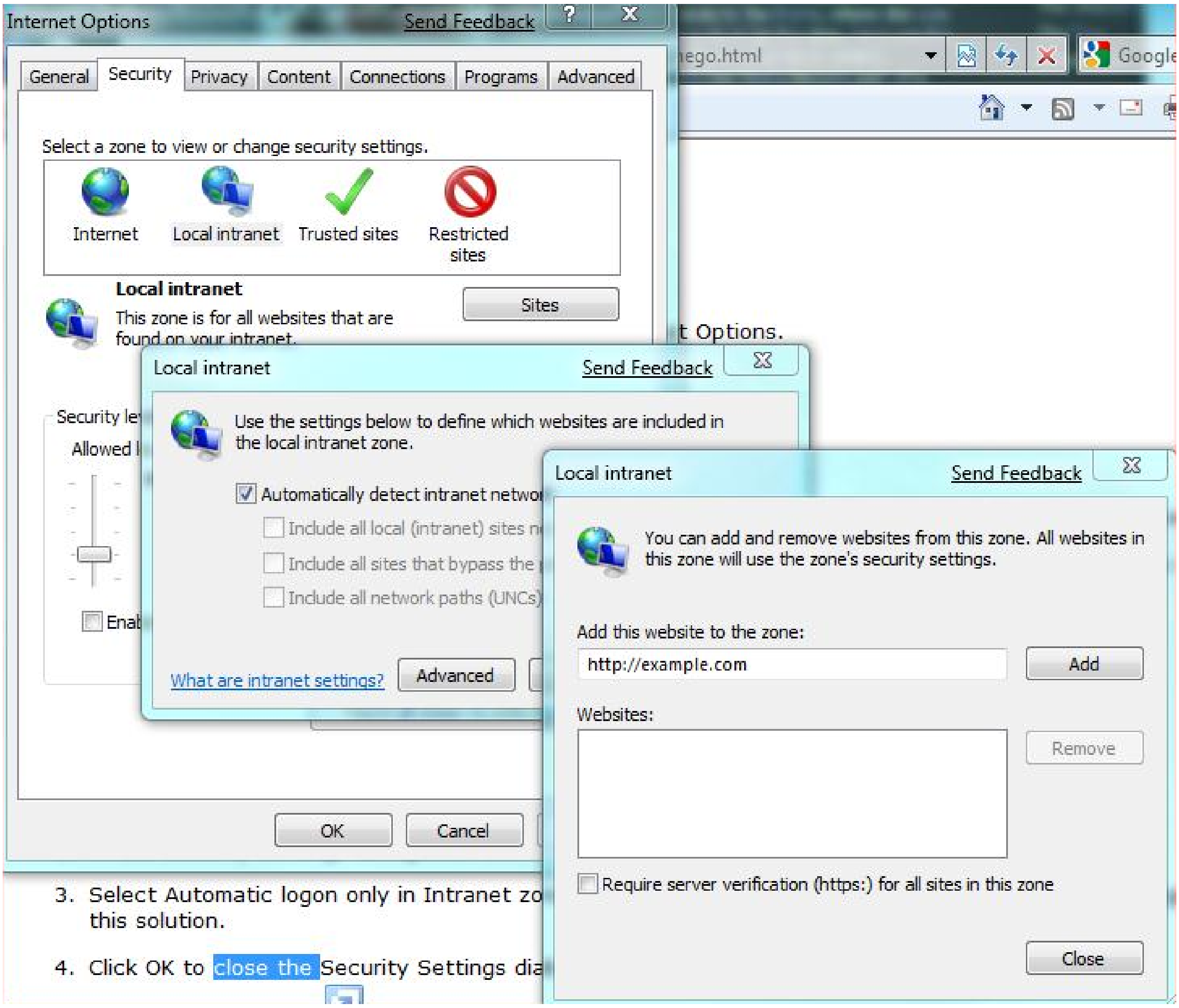
By typing *network.n* in the filter field it becomes easy to find the relevant options. These instructions were tested with Mozilla Firefox version 3.0.10 but should work with other versions as well.

A.2 Internet Explorer

To enable SPNEGO in Internet Explorer you first have to add your domain to the list of *local intranet sites*. You have to:

1. Start Internet Explorer.
2. In the menu, select Tools → Internet Options → Security.
3. Select the Local intranet icon and click Sites.
4. Make sure the *Automatically detect intranet network* checkbox is ticked.
5. Click *Advanced*.
6. Enter the URL of your site and click add, for example https://example.com.

See Figure A.2 for an illustration of these steps. Next you have to enable SPNEGO. Click on Internet Options → Advanced and scroll down to security settings.



Select a zone to view or change security settings.

Local intranet
This zone is for all websites that are found on your intranet.

Use the settings below to define which websites are included in the local intranet zone.

☑ Automatically detect intranet netwo
□ Include all local (intranet) sites n
□ Include all sites that bypass the
□ Include all network paths (UNCs)

You can add and remove websites from this zone. All websites in this zone will use the zone's security settings.

Add this website to the zone: http://example.com

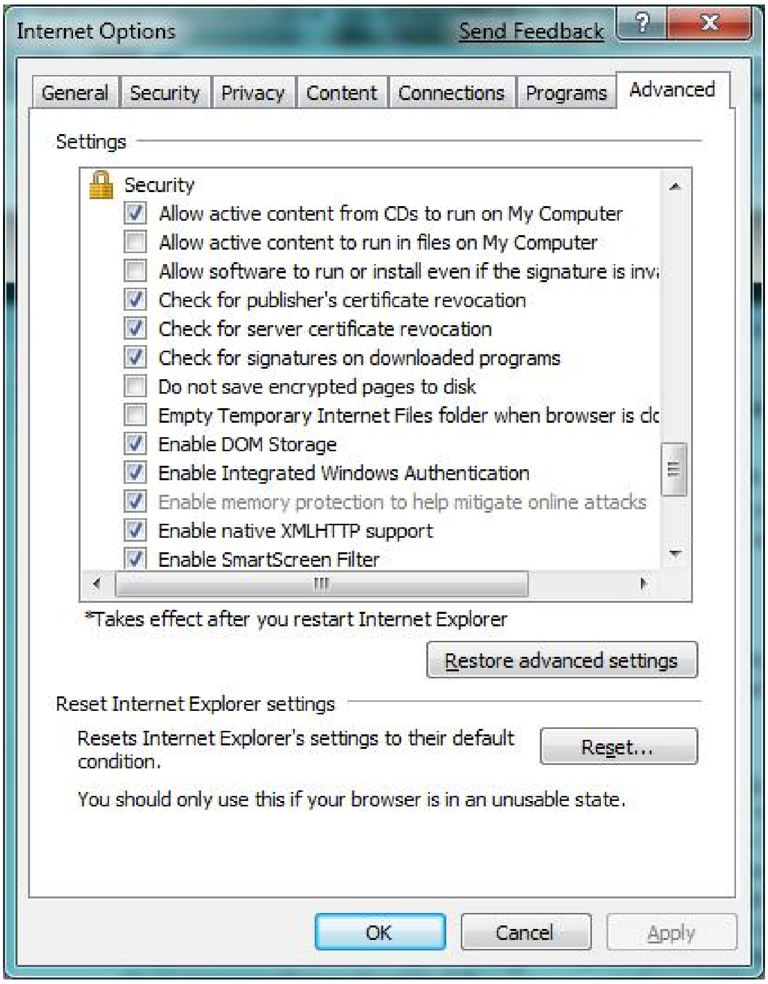
Websites:

□ Require server verification (https:) for all sites in this zone

3. Select Automatic logon only in Intranet zo this solution.
4. Click OK to close the Security Settings dia

Figure A.2: Adding your domain to local intranet sites

Make sure that the Enable Integrated Windows Authentication box is selected, see Figure A.3 for an illustration. These instructions were tested with Internet Explorer 8 but should be similar with other versions.



Security

☑ Allow active content from CDs to run on My Computer
☐ Allow active content to run in files on My Computer
☐ Allow software to run or install even if the signature is invi
☑ Check for publisher's certificate revocation
☑ Check for server certificate revocation
☑ Check for signatures on downloaded programs
☐ Do not save encrypted pages to disk
☐ Empty Temporary Internet Files folder when browser is clc
☑ Enable DOM Storage
☑ Enable Integrated Windows Authentication
☑ Enable memory protection to help mitigate online attacks
☑ Enable native XMLHTTP support
☑ Enable SmartScreen Filter

*Takes effect after you restart Internet Explorer

Reset Internet Explorer settings

Resets Internet Explorer's settings to their default condition.

You should only use this if your browser is in an unusable state.

Figure A.3: Enabling SPNEGO in Internet Explorer.

Appendix B

Libraries

Below is a collection of some of the authentication and authorization protocols that have existing Erlang implementations. There exist implementations for many protocols, but they can sometimes be hard to find. This list should help anyone who wants to add authentication or authorization in Erlang.

B.1 Egssapi

Egssapi is an implementation of the GSS-API protocol described in section 3.2.1, it can be found here: `http://github.com/danw/egssapi/tree/master`.

B.2 PAM

While there is no independent release of an Erlang PAM library, both Yaws and Ejabberd have implemented support for PAM. These implementations should provide a good reference for anyone who wants to add PAM support to their application. The Ejabberd implementation can be found at:
`http://github.com/processone/ejabberd/blob/5463478632f5c09cd694b548845f90a2549afedf/src/ejabberd_auth_pam.erl`.

The Yaws implementation can be found here:
`http://github.com/klacke/yaws/blob/3655d576ed510071b441c0f6fc621d8f888374f2/src/yaws_pam.erl`.

B.3 Eldap

Eldap is the Erlang implementation of the LDAP protocol and it is the library that was used in this case study. See section 3.3 for a description of LDAP. The API is easy to understand and includes an example. Eldap can be found here: `http://github.com/etnt/eldap/tree/master`.

B.4 EOpenID

OpenID is a federated authentication protocol. OpenID makes it possible for users to login with their credentials from any compatible provider that support the OpenID protocol. An Erlang OpenID library can be found here: `http://github.com/etnt/eopenid/tree/master`.

B.5 Erlang-oauth

Oauth is a relatively new, but very interesting protocol. The OAuth API allows secure authorization in a simple and standardized form. Oauth makes it possible for service providers to delegate resource access to third parties, called consumers. The Erlang implementation can be found here: `http://github.com/tim/erlang-oauth/tree/master`.