Daniel Carter

Kerberos-based single sign-on with delegation for web applications

Computer Science Tripos, Part II

Fitzwilliam College

2020-21

Declaration of Originality

I, Daniel Carter of Fitzwilliam College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I am content for my dissertation to be made available to the students and staff of the University.

Signed: Daniel Carter

Date: 13th May 2021

Acknowledgements

Thanks go to:

- My supervisor, **Dr Markus Kuhn**, for proposing the original idea for this project, supporting and advising me throughout the development process and giving advice on the dissertation
- My Director of Studies, **Dr Robert Harle**, for offering general advice through the year and making comments on my draft dissertation
- My overseers, **Professor Frank Stajano** and **Dr Amanda Prorok**, for advice in the project selection phase and feedback following the mid-project presentation
- **Ruth Carter** (my mother), **Jack Parkinson**, **Ted While** and **Julian Wreford** for reading the draft dissertation and making suggestions

Proforma

Candidate number:	2428G
Title:	Kerberos-based single sign-on with delegation for web applications
Examination:	Computer Science Tripos, Part II
Year:	2020-21
Word count:	11,941 [1]
Code line count:	1,106 [2]
Project Originator:	Dr Markus Kuhn [1]
Project Supervisor:	Dr Markus Kuhn

Original Aims

This project aimed to construct a Django-based web application that permits a user to present a Kerberos ticket for authentication, with the ticket being delegated to a database server such that the web server itself does not need permissions on the database. This would then be modularised so it could easily be used with any Django site, and an analysis done of the security and performance of the system. As possible extensions, I proposed to investigate using the same SQL server for web server logs (via a different Kerberos ticket), and the possibility of connecting to an NFS server.

Work Completed

The success criteria were completed (although ultimately modifications to Django itself were not necessary, so a web interface to simplify the setup of the database permissions was instead constructed). I also implemented part of the software for the extension goal of storing web logs on the same database server.

The project includes a demonstration web application as well as patches to PostgreSQL and Apache's `mod_auth_gssapi` module. There was some interest from the PostgreSQL developer community in including the patch in a future release, and I have also had discussions with the `mod_auth_gssapi` maintainer.

[1] Counted using TEXcount, including footnotes and captions

[2] Counted using `git diff --stat` against the initial state of each of the git repositories, and `wc -l` on the two Python test scripts

Special Difficulties

Completed during the COVID-19 pandemic.

Contents

Chapter 1

Introduction

1.1 Motivation

Many web applications are built around a database, which is used by the application framework to store user data and allow a user to view and edit it. Generally, the user logs in using a username and password (which is checked by the web app framework itself), and the application then makes database queries on the user's behalf, processes the results, and displays them to the user in a suitable way.

The application framework therefore has the ability to read and write arbitrary data in the database, and the only thing preventing a malicious user from accessing data for which they have no authorisation to is the application code itself. This is potentially problematic, since web app authors are often not security experts and may inadvertently cause data to be visible to users who should not be able to access it. Some examples of how this can occur include:

1.1.1 SQL Injection

SQL injection is a form of command injection attack (which are currently number 1 in the OWASP *Top 10 Web Application Security Risks* list [2] and are therefore regarded as a significant problem in web app security). It occurs when an application uses a template database query such as

```
SELECT * FROM records WHERE username='$user';
```

and replaces `$user` with a string that the user provides. However, without sufficient parameter checking, a user can supply a string such as

```
' OR 1=1; --
```

resulting in a database query of

```
SELECT * FROM records WHERE username='' OR 1=1; --';
```

which returns all users' records (since `1=1` is always true).

Figure 1.1 represents an example of a typical current setup, and the problems that can be caused if an SQL injection vulnerability is present:

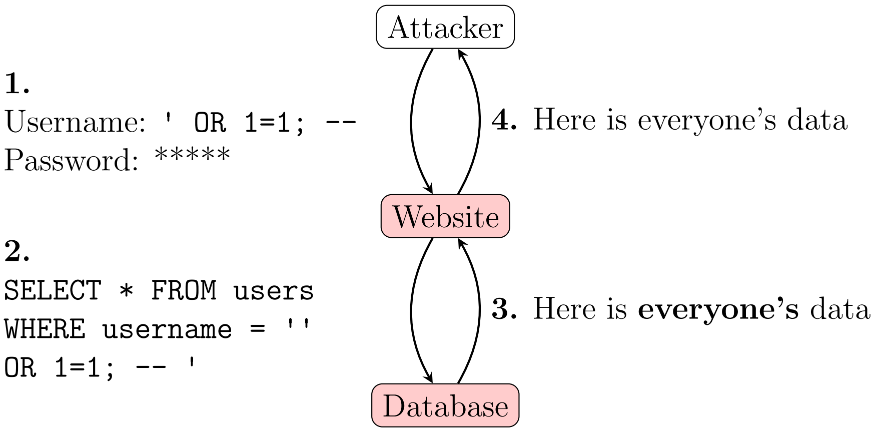


Figure 1.1: SQL Injection

Red nodes represent the **Trusted Computing Base** (TCB) of the system (the set of components which could cause the whole system to be compromised if that component contained a security vulnerability). In this setup, the website itself is a part of the TCB since a vulnerability here could allow an attacker to take over the system.

In step 2, note that the website can only steal data which is accessible to the user account used by the web app to log into the database. However, this is still likely to include the data of all users of the system since the web app needs legitimate access to user A's data when A logs into the system.

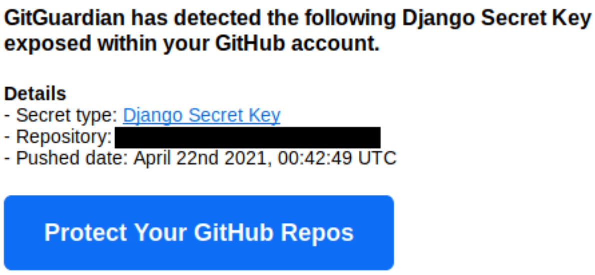
1.1.2 Master Password Leakage

Master password leakage has many variations depending on exactly what was leaked, but most such attacks enable any malicious user to arbitrarily read and write from the database. Since the web app itself has these capabilities, it usually has a single username and password for authentication with the database, and these are usually stored in a simple configuration file. If the webserver is misconfigured such that the configuration file is visible over the internet, or if the file is accidentally checked into a public source control repository, an attacker can read this file, connect to the database server and read out all the data.

In a way, SQL injection is also a form of master password leakage even though the attacker does not actually see the master password; instead, the attacker abuses an application which holds a copy of that master password.

To demonstrate the significance of this problem, a number of companies (such as GitGuardian [3]) exist simply to monitor online repositories for leaked security keys, and warn their owners. As a case in point, when setting up the test site for this project, the Django `settings.py` file initially contained a secret key that would allow an attacker to forge cryptographic signatures and so break the security of the application to an extent (but **not** gain full database access due to the application's structure – see section 4.5.2). Although the key was moved outside the repository (and changed, so knowledge of the old secret key would be of no use to an attacker) before any files were made public, the mere existence of the secret key in a prior commit caused a warning email (Figure 1.2) to be sent to me by GitGuardian within 70 minutes of changing the repository's visibility:

Even though this service only offers after-the-fact detection (a genuinely leaked secret key



GitGuardian has detected the following Django Secret Key exposed within your GitHub account.

Details
- Secret type: Django Secret Key
- Repository: [redacted]
- Pushed date: April 22nd 2021, 00:42:49 UTC

Figure 1.2: GitGuardian alert email

must quickly be changed everywhere it was used, and the site may still have been compromised in the meantime), the fact that businesses exist to perform these scans, and seemingly scan all public GitHub repositories to alert their owners, indicates the severity of this issue.

1.2 Legal Implications

As well as being commercially and reputationally damaging, unauthorised access to data can result in severe legal penalties: the Data Protection Act 2018 specifies that fines of up to 2% of a company's global turnover (or 10 million Euros, if greater) may apply for unauthorised disclosure of personal data [4]. Developing technologies that mean the web app does not have to be trusted with all data can significantly reduce the risk of these types of disclosure.

1.3 Website Sign-on Systems

Another disadvantage of separately checking passwords on each website is that users are then expected to memorise many different passwords, since otherwise a compromised password database on one website would allow the attacker to impersonate users on any other website where they had used the same password. Especially where several websites are "connected" in some way (e.g. by all being associated with the same organisation), a **single sign-on** (SSO) system can offer significant benefits.

SSO systems generally consist of an authentication server, which collects a password (or other security token) from the user to verify the user's identity. When a user wishes to log on to a site that uses a particular SSO system, the user is redirected to the authentication server to log in. Assuming the login is successful, the user is redirected back to the site with some kind of unforgeable token to indicate that the authentication server has verified the user's identity.

Note that these steps only provide **authentication** of the user (i.e. that the person logging into the site is actually user X). **Authorisation** (checking whether user X is entitled to access the site) must be done separately by the sites themselves.

The University's Raven authentication service is an example of an SSO system; numerous similar systems exist (including OAuth2, which also offers a form of "delegation" to allow one server to request resources from another, on behalf of the user [5]).

1.4 Project Summary

This project demonstrates an authentication system for web applications, such that **a user can authenticate to a web application using a Kerberos ticket, and the web application can use this ticket to obtain the user's data from a database**.

Apart from the requirement that a user has a Kerberos ticket in the first place, this is set up to be completely transparent to the user – the only action required is to visit the website and the authentication process is carried out automatically (with the minor caveat that Kerberos authentication must be specifically enabled in most web browsers [6], although in a corporate environment this could be done centrally).

As a demonstration of this in practice, this project includes building an **example file-browser application** that performs all authentication and authorisation via Kerberos tickets and database permissions, rather than implementing the security in the web app.

While most of the library functionality to achieve this already existed in some form, the actual completion of the project involved evaluating possible solutions and technologies for each aspect, and adding functionality to libraries where required.

The project also involved building the demonstration web application, to provide a simple setup interface and demonstrate the system's correct operation. However, its main purpose was to act as a feasibility study of using Kerberos-based authentication (and the associated ticket delegation) to websites. This is not a commonly used setup and many of the standards used with it (such as HTTP Negotiate) remain relatively little-known and obscure. Therefore, several areas of this project included research into possible development approaches before selection of a suitable technology, and in two cases **patches to existing open-source projects** were developed to add functionality not previously offered.

1.5 Related Work

The core Kerberos system and its extensions are described in a series of IETF RFCs, several of which are relevant to this project:

- RFC 2478 [7] describes the *Simple and Protected GSS-API Negotiation Mechanism* (SP-NEGO) which gives a (relatively generic) overview of negotiation-based authentication, where a server can provide information on what authentication methods it supports and the client can authenticate using one of these methods. This RFC illustrates examples of how the authentication process works, but is not tied to a particular protocol, and so is an important description of many of the overall concepts in this project.

- RFC 2743 [8] (along with several earlier RFCs) defines the current version of the *Generic Security Service API* (GSS-API) conceptually, using an exchange of tokens to prove identity and incorporating the concept of security tokens being delegable to another system or process. Principles stated here include that of the tokens being “opaque from the viewpoint of GSS-API callers”, where the user of a system simply requests a ticket and passes it to the appropriate service without the user attempting to extract the contents of the ticket. RFC 2744 [9] lays out a concrete API for RFC 2743 in the form of C language bindings.

- RFC 4178 [10] describes, at a conceptual level, a negotiation framework for GSS-API systems to select a mutually compatible authentication system. Under this setup, the system initiating a connection offers a set of connection mechanisms with an order of preference, and the system being connected to uses the highest-preference system which it is capable of using (or refuse the connection if none of the suggested mechanisms is suitable).
- A later RFC (RFC 4559 [11]) refers to a Windows-based system which allowed a user to log into a web application, from which a suitably equipped application could delegate the ticket to another system such as a database server. This was included by Microsoft in IIS 5.0 through adding a *Negotiate* extension to the HTTP protocol for use with Kerberos, and permitted a user to log in to the website without needing a password as described in the aims for this project.

The *HTTP Negotiate* extension has since been included in various open-source web browsers. It also has some support in the Apache web server, and this project's aim is effectively to replicate Microsoft's setup of a Kerberos-based single sign-on with delegation using an open-source web framework (Django), in conjunction with the Apache web server and any client web browser supporting the Negotiate extension.

Chapter 2

Preparation

2.1 Kerberos

The Kerberos protocol works on a system of *tickets*, which are managed by a *Key Distribution Centre* (KDC). The basic requirement is to allow users in a centralised database (e.g. the main user database in an office) to demonstrate their identity for logging in to applications and services, but **without** having to store or transmit passwords or other long-term secrets on potentially untrusted machines.

A very commonly used Kerberos implementation is found in Microsoft's Active Directory system, which is used by many organisations to centrally co-ordinate logins and resource access on Windows PCs on a corporate network. With only slight modifications, the setup illustrated in this report could be used with a pre-existing Active Directory server to provide a single sign-on system for an organisational intranet or similar.

Both the Microsoft-based and open-source implementations commonly use LDAP (Lightweight Directory Access Protocol) databases to store user information. At a basic level, these hold a set of users and corresponding *attributes* (potentially including information about how the user is allowed to log in, which resources that user is permitted to access and so on). The LDAP database also holds information about non-human *principals* (which could be systems such as web or database servers) and so, as explored in section 2.4, can be used to set delegation permissions from one system to another.

The basic workings of the Kerberos protocol are as follows:

- A user initiates a session by requesting a Kerberos ticket from the KDC, authenticating with a password.
- The KDC generates a *ticket-granting ticket* (TGT), and returns it encrypted using the user's password.
- The user decrypts the TGT, and the user's machine can then discard the stored password.
- To access a service, the user sends the TGT back to the KDC along with an identifier for the service to be logged into.
- The TGT grants the user a *service ticket*, which the user then passes on to the service. The service can then use that ticket for authentication.

The following listing shows a client (user1@LOCAL) with both a TGT and a service ticket. The ticket in the first line (krbtgt/LOCAL@LOCAL) is the *ticket-granting ticket* which the client obtained when first authenticating to the KDC, and the second line contains a *service ticket* (HTTP/krbsite.local@LOCAL) to log in to the HTTP service on krbsite.local (i.e. to access the web page hosted on that server).

```
$ klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: user1@LOCAL

Valid starting     Expires            Service principal
05/05/21 14:54:18  06/05/21 00:54:18  krbtgt/LOCAL@LOCAL
renew until 06/05/21 14:54:18
05/05/21 14:54:25  06/05/21 00:54:18  HTTP/krbsite.local@LOCAL
renew until 06/05/21 14:54:18
```

As shown in the output above, Kerberos tickets have an initial expiry time and a “renew until” time. The latter allows a user to remain connected to a Kerberos system for longer without having to re-authenticate, but reduces the potential risk of an unattended ticket being usable later since an already-expired ticket cannot be renewed.

2.2 SPNEGO

The *Simple and Protected GSSAPI Negotiation Mechanism* (SPNEGO) uses a single packet exchange to identify a suitable protocol to authenticate the user with. In practice, it is usually used with either NTLM (a different challenge-response protocol) or Kerberos. NTLM is non-ideal in practice because it uses the hash of a user’s password as a long-term key, meaning that a key that is stolen from a compromised machine can continue to be used until the user’s password is changed (whereas stolen Kerberos tickets are not practically useful since they expire within a short time).

The server can specify which protocols it is willing to accept. For example, using the mod_auth_gssapi Apache module used here, this directive ensures that only Kerberos (krb5) authentication will be accepted for GSSAPI:

```
GssapiAllowedMech krb5
```

A full copy of this project’s .htaccess file, which configures many options in Apache, is included in Appendix A.

2.3 Kerberos Backend

The KDC and related Kerberos backend services are a significant component of the system, and form part of the *trusted computing base* of the setup (see section 1.1.1). Although not actually requiring any new software development (in a real situation this backend would already exist, for instance as an Active Directory server), arranging a test setup with the right permissions was required for the project.

2.3.1 Basic MIT Kerberos Backend

The default MIT Kerberos backend is based on Berkeley DB and simply creates a set of 4 files to store information on the principals using the system [12]. While simple to set up (and what I used initially when experimenting with running a Kerberos server), the basic backend does not have sufficient functionality to support constrained delegation of tickets to another system as it lacks the required access control functionality [13]. Because this was a goal of this project, the Berkeley DB-based system was not suitable.

2.3.2 Connecting to an LDAP Server

Using the `kldap` module, a suitably configured LDAP database can be used as a data store for Kerberos principals. This offers additional flexibility above what is possible using the basic backend, and (crucially) allows attributes to be set defining what permissions a given principal has for delegation. This is explored further in section 2.4.

2.4 Kerberos Ticket Delegation

In addition to use for authentication, Kerberos supports a method of *delegating* tickets. This means that a principal A can pass a ticket to a service X as a means of authentication, and X can pass the ticket on to a third system Y to request resources on the user's behalf.

This is valuable because it means that X can access resources which "belong" to A, without X needing to have direct access to Y. Therefore, less trust in X is needed (since it does not need privileged access to Y) and so the attacks discussed in section 1.1 become much less likely.

2.4.1 Methods of Delegation

Kerberos supports several methods of delegation, and I investigated these when researching how to set up the web application. Broadly, there are three ways of controlling what data is accessible to an application X at a given time: which users' tickets the application actually has, how long those tickets last before expiring, and on which other servers X can use those tickets.

2.4.2 Unconstrained Delegation

The traditional, and simplest, method for delegation is simply for the user A to pass their ticket-granting ticket to the service X. X can now behave as though it were A, and access any resources to which A has access by simply presenting A's ticket-granting ticket to the KDC and requesting a suitable service ticket [14].

In MIT Kerberos, this is achieved by marking service X with the `ok-as-delegate` flag, which "hints the client that credentials can and should be delegated when authenticating to the service" [15].

Despite seemingly being no better than A simply giving a password to X (so that X can log in "as" A when accessing Y), this scheme offers some advantages:

- Kerberos tickets are time-limited, so if A no longer wishes to allow X to access resources, all A has to do is wait for any delegated tickets which X currently holds to expire[1]. By contrast, passwords are valid until the user changes them, may well not be straightforward to change in a complex corporate network and could have been re-used by users on other systems.
- Some limitations are placed on what can be done with the tickets. Although X can use the TGT to get a service ticket to any other system Y that A can access, Y does not automatically have the right to delegate this ticket further. If Y does **not** have the `ok-as-delegate` flag set, then X can **access** Y on A's behalf but not allow Y to perform actions for A on some other server Z. If Y is not well-trusted, this can be a significant benefit. Additionally, tickets can be bound to IP addresses to limit the risk of passing them on to insecure systems.

While the second of these advantages is significant, it still may not offer enough access control over the network. In particular, A cannot permit X to delegate to Y without also permitting X to delegate to **any** other service which A has access to. In many cases this would be too great a risk; it may well be desirable to allow X to fetch A's work documents from Y to display them in a web app, but not to allow X to retrieve A's financial records from another system Z within the same Kerberos realm.

In this case, constrained delegation offers a far more controllable method of only allowing services to delegate tickets in particular ways, at the expense of a more complex setup for managing applications.

2.4.3 Constrained Delegation (S4U2proxy)

The *Service for User to Proxy* (S4U2proxy) system is a Microsoft-designed extension to the basic Kerberos setup that allows one service X to obtain a service ticket to another service Y (on behalf of a principal A) in a controlled manner. Once service X has been marked as "permitted" to obtain tickets for service Y, it can do so simply by making a request to the KDC wit no need to know the user's Kerberos password [16].

Unlike with unconstrained delegation, the ticket-granting ticket is not passed over to X; instead, X is given a service ticket from A as normal (which proves A's identity). When X needs to access Y on behalf of A, X **submits the service ticket** to the KDC. If the permissions in the KDC are set appropriately, it gives X a ticket to Y on behalf of A.

This requires a more complex permission model in the KDC than simply a list of principals, since the KDC must now regulate when tickets can be delegated; as discussed in section 2.3, an LDAP backend allows the relevant `krbAllowedToDelegateTo` permission to be set on the user [13]. The following pattern is therefore used on the LDAP server to allow delegation from the web server on `krbsite.local` to PostgreSQL running on `krb.local`, by setting the `krbAllowedToDelegateTo` attribute on `HTTP/krbsite.local` to `postgres/krb.local`:

[1] Renewable tickets may slightly extend this timeframe, but do not cause a substantial difference.

```
dn: krbPrincipalName=HTTP/krbsite.local@LOCAL,cn=LOCAL
,cn=krbContainer,dc=local
changetype: modify
add: krbAllowedToDelegateTo
krbAllowedToDelegateTo: postgres/krb.local@LOCAL
```

Under Microsoft Active Directory, a similar setting named `msDS-AllowedToDelegateTo` is available, and this can be used in effectively the same way to enable constrained delegation from one service to another [17].

In addition to the advantages described above for unconstrained delegation, there are further advantages:

- Much richer customisation of access control is possible. A can be allowed to log into X and have a Kerberos ticket delegated to Y, or log into P and have a ticket delegated to Q, but not allow X to delegate to Q. There is no simple set of "privileged" applications which are allowed to perform (all) delegation, but a customisable set of permissions between systems, so applications which simply need to request some data from another server can be assigned restrictive access permissions and do not need to be part of the trusted computing base.
- The number of powerful ticket-granting tickets that are stored in systems on the network is reduced. If an attacker gains access to X, the attacker will, at most, get service tickets for all users who have recently logged into X (i.e. who have logged in and whose stored tickets have not yet expired), and delegated service tickets to the services which X can delegate to. The attacker does not get a ticket-granting ticket to use on arbitrary applications.

2.4.4 S4U2self

This description is included here for completeness (because of its similar name) although it is **not** a true method of delegating tickets.

The Microsoft standard document for S4U2self [16] states that:

> The S4U2self extension allows a service to obtain a service ticket to itself on behalf of a user. The user is identified to the KDC using the user's name and realm. Alternatively, the user might be identified based on the user's certificate.

In effect, Kerberos authentication is bypassed completely and the server X can get a ticket for A simply by specifying A's user name. This may be useful in some situations if not all clients can use Kerberos, but it offers a significantly weaker security model (since X can now obtain service tickets for any user) and so if X is compromised then records from all users can be gathered. As the aim of this project is to reduce the amount of trust placed in applications such as X, S4U2self is not relevant to this goal and will not be considered further.

The ability of X to get a delegable ticket using S4U2self can be controlled in the KDC, and so (for example) the following command is used to prevent the web server from doing this:

```
modprinc -ok_to_auth_as_delegate HTTP/krbsite.local@LOCAL
```

2.5 Storage of Credential Caches

To avoid considerably slowing down website accesses (see section 3.3), the web server must store local copies of client Kerberos tickets. Ignoring two Windows-specific mechanisms, MIT Kerberos offers five methods of storing credential caches [18]:

- **FILE** mode stores the credential cache into a file on the server's filesystem. This offers the best built-in support in existing modules, but its default configuration is vulnerable to having tickets stolen if the webserver is compromised (see section 3.3).
- **DIR** (directory) mode places all tickets in the supplied credential cache into a particular directory. In this case (since A only supplies one ticket to X) there is no advantage over **FILE**.
- **KEYRING** mode uses a feature of the Linux kernel to store credentials in an area of private memory. This has the advantage over many of the methods listed here that a kernel keyring can be made process- or thread-specific and so prevent other threads gaining access to private credential caches, but still has issues with removing cache access from threads which are re-used across several requests.
- **MEMORY** caches are conceptually similar to keyring-based caches except they are not Linux-specific and offer less granularity in setting access permissions, and many of the issues above also apply to this cache type.
- **KCM** mode was recently added to MIT Kerberos, and uses a specific daemon process to manage credential caches. Although potentially offering advantages over simpler systems (and being less OS-specific than **KEYRING**), it is still primarily tied to the UID of an operating system user account and so does not offer much security between different tickets held by the same server user. Since having a separate login account on the web server for each database user is normally impractical, its usefulness is limited here.

2.6 SQL Access Control

A core aspect of this project is **shifting access control from the web app itself to the SQL server**. Most SQL server applications contain permissions models which allow selective access to data, and there are often several ways of achieving this (especially since they are not very well standardised across SQL implementations). As this project specifically uses PostgreSQL, I evaluated several methods to determine how best to implement the security protocol.

2.6.1 Basic SQL Permissions

In many cases, the basic SQL syntax to allow or deny users access to a whole table may be sufficient. For example, simple SQL `GRANT` statements allow setting up permissions on a table of sensitive information (that only a certain group of employees are allowed to see) without any further complexities. Only where users need access to specific rows within a table do more complex access control methods need considering.

2.6.2 SECURITY DEFINER Functions

As with most SQL implementations, PostgreSQL implements stored procedures using the CREATE FUNCTION operation. This allows a pre-written operation, which performs a series of actions and may make changes to the database and/or return a result, to be provided to database users.

Basic stored procedures do not have any specific security implications; they are merely a way to avoid repeatedly executing the same code (potentially offering performance improvements). However, PostgreSQL defines an additional construct for security management: the SECURITY DEFINER property.

A PostgreSQL function with this property specified runs "as" the user creating the function, regardless of who executes it [19] (in a similar manner to the setuid bit in Unix). This allows the database owner to create a function which can read the whole table and when run, perform some check based on the user calling it and return appropriate results.

Effectively, these are similar to Trusted Procedures in a Clark-Wilson security model, where data can only be modified by particular procedures which enforce constraints on the data. This allows a separation of authentication (logging into the database, via Kerberos) and authorisation (where the procedures describe what access is permitted).

2.6.3 SQL Views

Views allow a dynamically-generated table to be created out of data in existing tables. Syntax such as the following constructs a view which only has the calling user's data visible; users can then be granted access to this view and do not need any access to the original table. For web application purposes, it can be queried just like any other table.

```
CREATE VIEW my_files AS
SELECT f.*
FROM files_file f, files_permission p
WHERE f.id = p.file_id
AND p.owner = user;
```

Write access is not directly available for views containing cross-table joins, but can be achieved either by using ON INSERT rules in the same way as the ON SELECT rules below, or by using SQL triggers [20].

2.6.4 Use of ON SELECT to Redefine Selection

PostgreSQL also allows creation of rules which redefine actions on tables. For example, a rule could replace the default SELECT rule on the table storing file information with one that checked user permissions.

However, this is likely to introduce unnecessary complexity (particularly with ensuring that administrators can still access data as necessary, for example). For these reasons, the PostgreSQL documentation considers it "better style to write a CREATE VIEW command than to create a real table and define an ON SELECT rule for it" [21].

2.6.5 Row-level Security

The final method considered, and the one which turned out to be best-suited to this example web application, was PostgreSQL's row-level security constructions. These allow policies (defined by the database owner) to determine who can view or modify which rows in the database, and is well-suited to tasks like this.

See section 3.4 for more information about how this was used.

2.7 Requirements Analysis

In addition to a feasibility study of the underlying technologies, this project includes a demonstration web app to illustrate their use. There are three broad components to the setup: the web server, the web application framework and the database:

- Of the available open-source **web server** frameworks, Apache and nginx are two which receive significant use. Both have some form of existing Kerberos support and so this project could probably have been completed using either. As Apache is the only one of the two with which I had previous experience, I decided to continue using it following some experimentation over the summer (see section 2.9). Apache's module system is also more flexible than nginx's, which offers advantages in this project.
- For the **web application framework** there are many more options, from a basic setup with PHP files to more full-featured systems. Although there would likely have been many good candidates, I selected Django as a widely-used framework which offers the required features (and is written in Python, which is a language that I was previously familiar with).
- Finally, there are also a number of possible options for the **SQL server** running the database. Although my previous SQL experience was primarily using mySQL, PostgreSQL offers more advanced functionality in several areas (including flexibility in table security options) and so offers advantages for this project. It also offers built-in support for authenticating to the database using a Kerberos ticket, which is clearly useful here.

2.8 Project Management

2.8.1 Source Code Management

For the Django directory itself, setting up a git repository with commits pushed to GitHub provides both version control and backup. This was also checked out into a separate folder on my PC which was backed up to Dropbox, to provide additional layers of backup.

Since the main development was completed in virtual machines which would have been time-consuming to re-create if lost, I also took backups of the whole VM setup to an external hard drive during development. In the event of a failure, I could have restored these backups to a replacement PC to continue working.

2.8.2 Testing Methodology

Since the core aims of the project are to provide a secure system, this can be most appropriately tested using a security analysis of the types of vulnerabilities (such as SQL injection) which it is vulnerable to, and how this compares with other systems in current use. A naïve implementation of the project also has the potential to result in slower performance than existing systems (due to extra authentication overheads), so evaluation of performance and comparing it with a non-Kerberos based system is also considered.

2.9 Starting Point

(The text below is a slightly modified version of the starting point in my project proposal)

Kerberos support for Apache already exists (via the `mod_auth_gssapi` module), and this includes some ability to delegate tickets although with drawbacks (principally that the tickets to be delegated are placed in a directory accessible to any process running as the webserver user). Django similarly includes Kerberos extensions (such as `django-gssapi`), but these are ultimately not really relevant to this project since Django does not need to participate in Kerberos authentication other than passing on login information. As a starting point, I investigated these and built on the functionality already offered.

PostgreSQL already has support for Kerberos and is used as a back-end database system.

I previously had some knowledge of authentication systems and general security topics from the 1B Security course, and of C programming from the 1B Programming in C and C++ course (although Apache includes several specific libraries that differ from those used in "normal" C programming [22]). I have also had general experience with web application setup (although not the specific technologies involved here, which required learning how Django works and the syntax of its templating language) through the 1B Group Project and work done outside the Tripos.

Over the summer vacation I did some experimentation with Kerberos authentication to set up a virtual machine using existing Kerberos functionality in Apache and PostgreSQL (logging in separately to the two applications without any delegation of tickets).

The Django documentation [23] was used as a reference and guide throughout development, including using (some of) the provided tutorial to set up the basic web app before integration with Kerberos authentication.

Chapter 3

Implementation

3.1 Project Structure

3.1.1 Overall Structure

Figure 3.1 shows the overall structure of the authentication process.

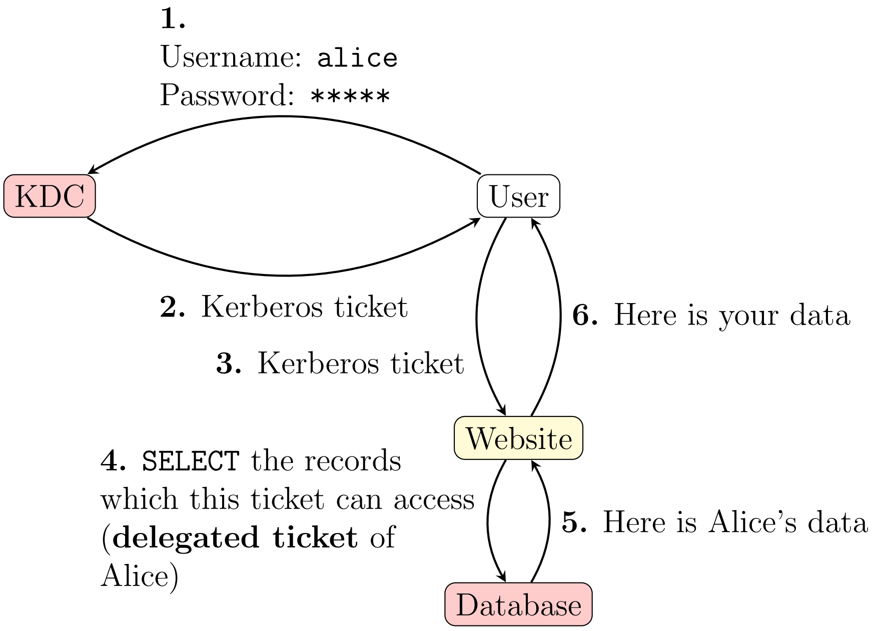


Figure 3.1: The authentication process

Since the same KDC can be used for many websites where all users have a login to the same Kerberos realm, the system also provides single sign-on functionality (using the Kerberos ticket as proof of identity). As described in section 2.4, none of the individual websites need access to the user's password, they cannot arbitrarily impersonate the user on other systems, and they cannot get any data from other systems which they have not been authorised to access. A website also cannot get data on behalf of users for whom it does not hold a live Kerberos ticket (so there is no way to access the records of users who have not used the system for a time interval at least as long as the expiry time of their tickets).

Note that, unlike in Figure 1.1, the web application no longer has to be part of the trusted computing base; it merely passes on the Kerberos ticket which a user has provided and does not have any other method of accessing the data in the database. This means that an attack such as the one depicted in Figure 1.1 cannot take place.

3.1.2 Demonstration Web Application

The project demonstrates a web application that uses Kerberos authentication and delegation to fetch data from a database and display it to the user, in the form of a basic file browser-type application. Django uses of a structure based on *models* which correspond roughly to classes in object-oriented programming (and are represented by Python classes in the Django configuration), where each model has attributes that are set up as member variables of the class. When stored in a database, each model is a table, each object of a model is a row in that table and each attribute is a column.

In the example web app, the models are files and directories, plus associated tables to hold access permissions for both files and directories. Figure 3.2 shows the database model as an entity-relationship diagram.

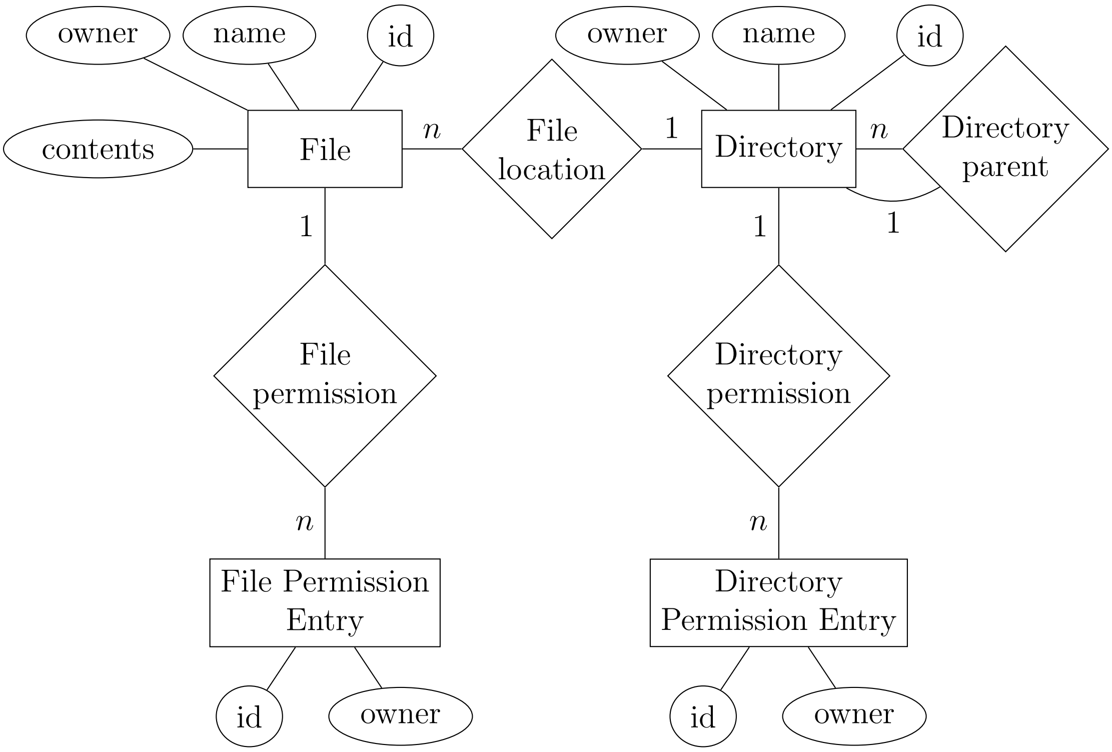


Figure 3.2: Entity-relationship diagram for the example app

3.1.3 Database Setup Application

The project also includes a component which performs the necessary setup tasks (setting up SQL permissions), so that all that is required for the user of the application is to:

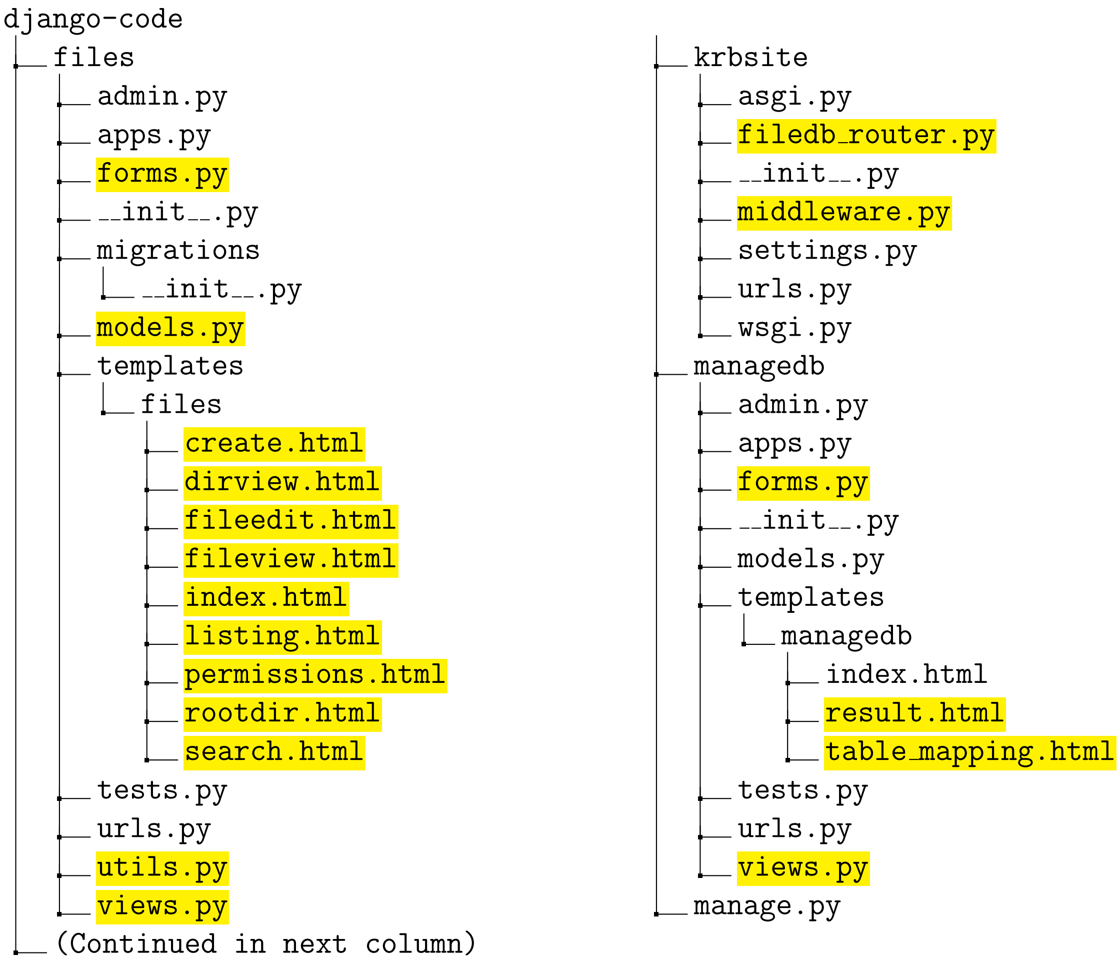
- Set up a Django app with appropriate models
- Run the normal Django `migrate` command to set up the database

- Use the web interface (created as part of this project) to set up access controls

3.2 Repository Overview

The core of the repository is a Django web app, which can both perform the necessary setup steps to get the system working and (as a separate component) demonstrate the working of the system.

Figure 3.3 shows the structure of the directories involved. The basic directory layout is as produced by the Django project setup procedure, with some files having "stubs" automatically generated by this process (it also provides a default `settings.py` and the short `manage.py` script which runs various application management steps).



```
django-code
├── files                              ├── krbsite
│   ├── admin.py                       │   ├── asgi.py
│   ├── apps.py                        │   ├── filedb_router.py
│   ├── forms.py                       │   ├── __init__.py
│   ├── __init__.py                    │   ├── middleware.py
│   ├── migrations                     │   ├── settings.py
│   │   └── __init__.py                │   ├── urls.py
│   ├── models.py                      │   └── wsgi.py
│   ├── templates                      ├── managedb
│   │   └── files                      │   ├── admin.py
│   │       ├── create.html            │   ├── apps.py
│   │       ├── dirview.html           │   ├── forms.py
│   │       ├── fileedit.html          │   ├── __init__.py
│   │       ├── fileview.html          │   ├── models.py
│   │       ├── index.html             │   ├── templates
│   │       ├── listing.html           │   │   └── managedb
│   │       ├── permissions.html       │   │       ├── index.html
│   │       ├── rootdir.html           │   │       ├── result.html
│   │       └── search.html            │   │       └── table_mapping.html
│   ├── tests.py                       │   ├── tests.py
│   ├── urls.py                        │   ├── urls.py
│   ├── utils.py                       │   └── views.py
│   └── views.py                       └── manage.py
└── (Continued in next column)
```

Figure 3.3: Repository overview

The top-level directories represent the separation of "components" in the Django setup, with `krbsite` containing whole-system configuration, `files` containing the setup of the demonstration file manager application, and `managedb` having the code for the database setup application.

The primary code files are those highlighted above, with the most substantial code additions being in the `views.py` files for both `files` and `managedb` (since these contain the main logic for constructing web pages in the file browser and database management apps respectively).

The forms.py files in each of the two directories have definitions of web forms for gathering user input (some of these are very simple text-gathering fields but others, such as the TableMappingForm in managedb/forms.py, require complex validation of user input.

files/utils.py contains two helper functions called by files/views.py.

The object model for the file browser application is defined in files/models.py, in the same way as for any other Django app. This includes both the actual data storage tables Directory and File and two permission tables (as shown in Figure 3.2) that are defined here so the web app can manipulate them using standard Django model operations which are fed back to the database. The models.py file under managedb does not contain any models since the database management app does not need to store any state itself – it simply "reaches in" to the file management app's database structures to add security policies.

The tests.py files are automatically created by Django for writing unit tests. However, this test setup is generally designed for testing application-specific features (e.g. whether the object model has been correctly defined) so is not especially useful for this project and was not used.

The templates directories contain Django templates for HTML pages, which contain special commands (filled in by the rendering engine) such as the following:

```
{% for dir, subfiles, subdirs in directories %}
        {% include "files/listing.html"
           with root=dir files=subfiles dirs=subdirs %}
{% endfor %}
```

Two of these also contain JavaScript functions, including one which populates one form field based on the value of another in managedb/templates/managedb/table_mapping.html.

In keeping with the principle that the web app should simply be a "view" onto the main database, and the limited privileges that the app itself has for writing to the database, Django uses a separate database for recording session information and associated "housekeeping" data. This does not include any of the actual records which have access control applied to them.

For ease of setup (since this is not a significant aspect of the project), this was done using a simple SQLite database, although it did require a Django AuthRouter to make sure that queries are always directed to the appropriate database: see Appendix B for more details and code.

Also included in the repository, but outside the django-code directory, are the two **patches** (provided as diffs against the existing projects) and a testing directory containing two Python scripts for **performance evaluation and graph plotting** respectively.

3.3 Credential Caching in the Web Server

When user A provides a Kerberos ticket for service X as part of the GSSAPI process, the ticket can either be discarded immediately following the completion of the request (including after any necessary delegation has occurred) or be cached by the server for future use.

The first approach offers a "purer" Kerberos setup, since the web server does not need to store any state and can simply use a ticket provided by the user on each request. However, it significantly limits the performance of the system:

- Since the basic HTTP Negotiate setup requires a multi-stage "handshake" before communication can begin, each request to the web server becomes at least two requests. This potentially doubles the website access time, as well as consuming extra network bandwidth and placing additional load on the web server. Given that individual web pages usually consist of many elements (e.g. each image on the page would require a new request and another round of HTTP negotiation) this is certainly non-ideal.
- Kerberos ticket files are potentially large (especially those issued by Windows-based systems[1]) so including the full ticket in each request may use significant additional bandwidth. This further increases demand on the server, and could cause problems for users on slow or quota-controlled network connections.

See section 4.6 for more discussion of the problems with this approach.

One possible solution would be for the web server to place the ticket itself into a cookie that is sent to the client. This would mean that subsequent requests could take place as normal (with the client providing the cookie and the server extracting the ticket from it) and so solve the first of the issues above, but still leave (potentially) large cookie files being sent around as an overhead on every request.

The solution generally adopted here is to cache the tickets on server X and send a (short) session cookie to A, so X can identify A when A connects again and provides the same cookie. However, since this cookie does not carry any information beyond acting as an identifier, X must store a copy of A's Kerberos ticket to use when A reconnects to the server.

As described in section 2.5, MIT Kerberos offers several ways to store credential caches for future use. Although keyring-based credential caches are initially attractive, the process model used by the Apache web server introduces some difficulties: since the same Apache worker process may be used to process different (unrelated) requests at different times, it is important that these requests do not have access to the credential caches saved for previous requests using that thread. This becomes difficult to achieve where the thread and process IDs remain the same across requests.

Further, `mod_auth_gssapi` is currently only designed to use file-based caches (any input to the `mag_store_deleg_creds` function is assumed to be a filename and has `FILE:` prepended, and the `gss_store_cred_into` function which is then called is only set up to write output data into a file). It would be possible to modify this to work with kernel keyrings (and this was something I investigated as part of the project), but eventually I used a variation on the file-based approach.

Although the best-supported by current systems, file caches are non-ideal in their default configuration because they are normally placed in a directory whose contents can be enumerated and read by any other process running as the same user. In this case, this means that a web app which was compromised to allow code execution in the context of a user's web app session could access tickets belonging to other website users who had logged in recently and whose tickets were still cached.

However, clever use of the Unix directory permission model allows this limitation to be overcome. Since read (`r`) permissions on a directory allow the user to list files contained in it,

[1] As a (somewhat simplistic) illustration, a Kerberos ticket from my test application (running MIT Kerberos) produced a 650-byte credential cache, compared with 1536 bytes for a ticket from the University's Active Directory server `BLUE.CAM.AC.UK`.

and execute (x) permissions allow a user to *traverse* the directory and read a file whose name is known, granting execute but not read permissions allows the filename to (in effect) become a password that can be used to access the file. The permission to list files in the directory is not provided, so it is not possible to obtain the filename unless it is separately provided. Therefore, **combining this with a "secret" filename that is not easily guessable (and is provided separately to the specific webserver process that needs it) can keep saved tickets secure from other processes running as the same user**.

3.3.1 mod_auth_gssapi Patch

To make use of this, I added an option to the `GssapiDelegCcacheUnique` directive in `mod_auth_gssapi` which adds a random 16-character string to the credential cache name. When combined with `0300` directory permissions (i.e. `d-wx------` – the owning user can write to the directory and traverse it to find files with known filenames, but not read the list of files in it), tickets can only be accessed by knowing the credential cache filename, which is passed to the web app and on to the Kerberos library via an environment variable. Therefore, one instance user's of the web app is not able to obtain tickets belonging to another.

The most significant addition in this patch was a new function to generate the random string (using Apache's own random byte generator, but mapping it onto a string that can be included in a filename). I initially investigated using a base64-encoded series of random bytes as a way of representing the random data as a text string, but the "standard" base64 encoding used in the `apr_base64` Apache library does not produce a string suitable for inclusion in filenames because it includes the `/` character as part of the character set [24]. A new library to include the `BASE64URL` encoding method was recently added to Apache [25], but was not yet available in the packaged Apache version I was using and would have reduced the utility of the patch by causing `mod_auth_gssapi` to require a more recent set of libraries to work. Since an actual standardised encoding of information is not required (just a random character sequence which can be part of a valid filename), I simply used 6 bits per byte of the random byte string and mapped these onto printable characters:

```
static char *get_random_string(apr_size_t length, apr_pool_t *pool)
{
    const char *chars =
     "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_.";
    /* fill a buffer with random data */
    unsigned char *data = apr_palloc(pool, length);
    apr_generate_random_bytes(data, length);

    /* convert into filename-safe characters */
    char *output = apr_palloc(pool, length+1);
    apr_size_t i;
    for (i = 0; i < length; i++) {
        output[i] = chars[data[i] % 64];
    }
    output[length] = '\0';

    return output;
}
```

Further details on the patch are shown in Appendix C.

3.4 Implementing Database Access Control

Having investigated possible methods to set up access control on database tables in section 2.6, I evaluated these methods for use in the project.

3.4.1 Stored Procedures

While very flexible, `SECURITY DEFINER` functions cannot easily be integrated into frameworks such as Django, which are designed to execute queries on tables rather than (effectively) making function calls to an API. To integrate this approach properly into Django, I would have needed to replace the current object-relationship mapper with one that supported using procedures rather than simple `SELECT` statements, or otherwise the web app designer would have to manually make SQL queries rather than using the built-in database methods. Since the goal is to allow the user to set up a Django web app using existing models, this setup is certainly non-ideal.

However, it **can** be used to perform certain "privileged" operations as part of the access control checking procedure. I therefore implemented the following (template) function, which allows a user's privileges to be checked without the user having read access over the whole privilege table:

```
cursor.execute(f"""CREATE OR REPLACE FUNCTION {source_table}_permitted
(object {source_table}.{source_column}%TYPE)
RETURNS boolean
AS $$
BEGIN
RETURN session_user IN (SELECT {owner_column} FROM {perm_table}
WHERE {perm_column} = object);
END;
$$ LANGUAGE plpgsql
SECURITY DEFINER;""")
```

The bracketed strings such as `{source_table}` are filled in by the setup application to reflect the specific database tables in use.

3.4.2 SQL Views

While less significant than in the case of `SECURITY DEFINER` functions, views also introduce difficulties for interfacing with Django since each database table now needs two entries in the Django model setup (one for the table itself and one for the view). Using abstract classes and inheritance allows some reduction of duplication, but this is still non-ideal. For example, the following code (which I developed for an initial implementation of the web application but subsequently replaced) defines `File` (the actual model represented in a database **table**, but not visible to ordinary users) and `UserFile` (the SQL **view** that users can select from, which is marked as unmanaged to stop Django creating a full table for it). The `AbstractFile` class avoids repeating all the fields in both of the subclasses, but still makes for somewhat messy code:

```
 class AbstractFile(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=127)
    directory = models.ForeignKey(Directory,
                        on_delete=models.CASCADE, null=False)
    contents = models.TextField(null=True)
    class Meta:
        abstract = True

class File(AbstractFile):
    pass

class UserFile(AbstractFile):
    class Meta:
        managed = False
        db_table = 'my_files'
```

3.4.3 Row-level Security

The main database access control is done using row-level security policies on the database tables. PostgreSQL offers two forms of policy, which are applied in different circumstances [26]:

- USING policies work on existing rows to determine whether users are allowed to view and/or update the contents of a row. This is used for the main data tables (since the main concern is the confidentiality of file data), where the following lines of code allow users to view and update files for which they have permission records:

  ```
  # Enable row-level security on the relevant table
  cursor.execute(f"""ALTER TABLE {source_table}
                     ENABLE ROW LEVEL SECURITY""")
  # Only allow users to view objects which they have permission to see
  cursor.execute(f"""CREATE POLICY {source_table}_view
                     ON {source_table} FOR SELECT
                     USING ((SELECT {source_table}_permitted
                     ({source_column})))""")
  # Or where they are the owner (in which case allow editing as well)
  cursor.execute(f"""CREATE POLICY {source_table}_view_owner
                     ON {source_table}
                     USING ({source_owner_column} = session_user)""")
  ```

  With a USING policy, a row which fails the policy is not visible to the user and so cannot be accessed.

- WITH CHECK policies are designed to protect insertion of new rows. These are more suited to protecting the integrity of data (by stopping unauthorised additions) and so are used in this application for the permission tables. Since an attacker could otherwise maliciously insert permissions to allow access to arbitrary files, this is another critical part of the security model:

```
# Only allow users to set permissions where they are the owner
cursor.execute(f"""CREATE POLICY {perm_table}_insert ON {perm_table}
                   FOR INSERT WITH CHECK ({perm_column} IN
                   (SELECT {source_column} FROM {source_table}
                   WHERE {source_owner_column} = session_user))""")
```

Note that the table names above are separately verified by the application before being placed into these database queries to avoid SQL injection being directly possible here, but also that logging in as the database administrator to perform these types of actions inherently involves having full access to the database via the administrator's Kerberos ticket. Therefore, it is impossible to remove all risk of SQL injection if the application is compromised **while** the setup phase is going on.

The row-level security model runs in parallel with the standard SQL permissions model and an action must "pass" both systems to be permitted. Thus, to only allow read-only access to selected rows, a `USING` policy can be set up to determine row accesses, and the user concerned only granted `SELECT` (and not `UPDATE`) access to the table. This is employed in the application to restrict malicious modifications to permission entries.

3.5 The Database Setup Application

As an illustration of a setup interface for a Kerberos-based web app, the database setup application is designed to **allow a website administrator to set up SQL permissions easily, such that other Django applications (including the file browser example app) work with minimal changes**.

This application first gathers information from the user to arrange a "matching" of data and permission tables (determining which table holds permission records for a given table of data). This clearly depends on where there are appropriate table relationships already set up (a permission table must contain a foreign key to the data table), and PostgreSQL exposes this information via an internal `information_schema` table [28]:

- `information_schema.table_constraints` contains details of all constraints on database records; in this case, only `FOREIGN KEY` constraints are of interest.
- `information_schema.constraint_column_usage` describes the columns **referenced** by a foreign key constraint.
- `information_schema.key_column_usage` lists columns which are **restricted** by key constraints.

Therefore, executing the following query on a given model (using the `model_data` parameters which are incorporated into the query) provides a set of candidate permission tables where such a foreign key exists:

```
cursor.execute("""SELECT dst.table_name, dst.column_name
    FROM information_schema.constraint_column_usage src,
         information_schema.key_column_usage dst,
         information_schema.table_constraints constraints
    WHERE src.constraint_name = constraints.constraint_name
    AND constraints.constraint_name = dst.constraint_name
    AND constraints.constraint_type = 'FOREIGN KEY'
    AND src.table_name = %s
    AND src.column_name = %s""",
    [model_data['table'], model_data['primary_key']])
```

(The `FROM information_schema.constraint_column_usage src` syntax would be written as `FROM information_schema.constraint_column_usage AS src` in some other SQL implementations.)

The application then performs a further query to the `information_schema` table to identify columns in these candidate tables which could be used to store the user's identity in a permission element. Because Django itself has no knowledge of the database's users, these are simply stored as string values (`character` or `character varying` in PostgreSQL nomenclature) in the database:

```
cursor.execute("""SELECT column_name
    FROM information_schema.columns
    WHERE table_name = %s
    AND data_type LIKE 'character%%'""",
    [table])
```

This therefore provides **an easy-to-use interface to set up Kerberos-based access control to a site, without having to set up SQL permissions manually**.

3.6 PostgreSQL Patch to Specify Credential Cache

Although PostgreSQL includes Kerberos-based authentication via the GSSAPI framework [27], the client which connects it to Django (and to most other applications) currently does not offer a choice of which ticket to use and simply takes a stored ticket from the system's default `ccache` location. This is obviously unsatisfactory for an application where many users' tickets will be stored and where the application must use the appropriate ticket each time, and so I have added a patch to the PostgreSQL client library to perform this.

Django's interactions with PostgreSQL are achieved using the `psycopg` library, which forms the basis for the PostgreSQL backend which comes with Django. However, much of the actual library functionality is simply a wrapper around the C-based `libpq` library, which is provided as part of PostgreSQL and manages the actual interaction with the database server.

To specify the ticket which will be used for a Kerberos-based (GSSAPI) connection, the MIT Kerberos API provides a `gss_krb5_ccache_name` function which allows the user to specify a credential cache to use. The "core" addition to the `libpq` library is simply a call to this function at the appropriate point, specifying the location of the credential cache which is to be used:

```
if (ccache_name != NULL) {
        gss_krb5_ccache_name(&minor, ccache_name, NULL);
}
```

As well as this, some further changes are needed to store the ccache name provided by the user with the rest of the connection information, until it is actually needed at the point of initiating the connection. Therefore, I added an additional field to the PGconn structure, with an associated new option in the PQconninfoOptions array:

```
#ifdef ENABLE_GSS
       {"ccache_name", NULL, NULL, NULL,
               "Credential-cache-name", "", 64,
       offsetof(struct pg_conn, ccache_name)},
#endif
```

The #ifdef macro allows the compiler of PostgreSQL to choose (at compile time) whether GSSAPI support will be included in the build or not.

Note that the public API does not change at all: since the user passes in a set of key-value pairs specifying connection options, the only change made here is to recognise an additional option in this set (via the ccache_name shown above). If the user does not specify this option, its value defaults to NULL and (because of the null check around the gss_krb5_ccache_name call) it is simply ignored and PostgreSQL works as normal.

Having created the patch, I submitted it to the pgsql-hackers mailing list for consideration by the PostgreSQL maintainers [29]. Following some discussion of use cases for this patch, another list member (Stephen Frost) suggested that for an application such as this one, it would be possible to pass the ticket directly from Apache to the webserver's Kerberos library via an environment variable. That is true in this particular scenario (and I then implemented that in this project as a simpler solution), but there are still certainly use cases for the patch. In the case of this project, it could be used if the webserver needed to access the database using credentials other than the user's (for example, in a system where the web server can write a log to a database table using its own Kerberos ticket as discussed in section 4.1).

Several other users on the mailing list expressed interest in having the patch submitted for inclusion in a future PostgreSQL version.

3.7 Summary

This section demonstrates how **a web app operating entirely by means of delegated Kerberos tickets** can be set up, with the project including an example application. The security and performance of the system (along with its usability) are key considerations in its effectiveness, and these will now be considered in the evaluation.

Chapter 4

Evaluation

4.1 Success Criteria

The two success criteria for the functionality of the application (as described in Appendix E: that a user can log in by presenting a suitable Kerberos ticket, and that the application has no access to the database **except** using those tickets) were **both fully met**, and the application demonstrates this.

The final criterion (that the project is constructed as a Django add-on) was **ultimately not necessary**, since no Django modifications were needed to achieve the changes made. Instead, I constructed a setup application (implemented as a Django-built app) that makes the necessary setup permission changes on the database and provides an easy interface for the database administrator to set up the system.

As an extension, I proposed investigating using the same SQL server to record web server logs (with the web server having its own Kerberos ticket that it could use for this purpose in place of the user's ticket). Although I did not actually implement this, the PostgreSQL patch detailed in section 3.6 would form part of the setup required by allowing the web server to switch between tickets to connect to the database.

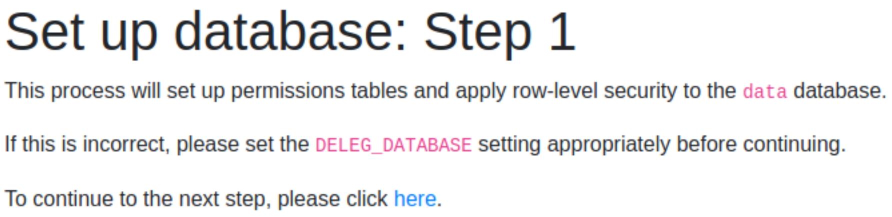
4.2 The Django Application

Since this project was intended to provide a configuration which is easy to set up and use on a Django-based website (as well as providing a usable proof-of-concept web application), I now present a walkthrough of the system to demonstrate its usability.

After authenticating to the web server via Kerberos, administrative users with sufficient PostgreSQL privileges can set up access control policies via the database setup application, and all users with appropriate permissions can view and edit "files" on the system. The actual authentication process is done by the browser and is transparent to the user (see section 4.3).

4.2.1 Database Setup

The database setup application first displays a welcome screen (Figure 4.1), showing the database name to be operated on (fetched from the Django configuration – `DELEG_DATABASE` is a new option added to the `settings.py` file which specifies to the web app which database should be accessed via delegated ticket).



Set up database: Step 1

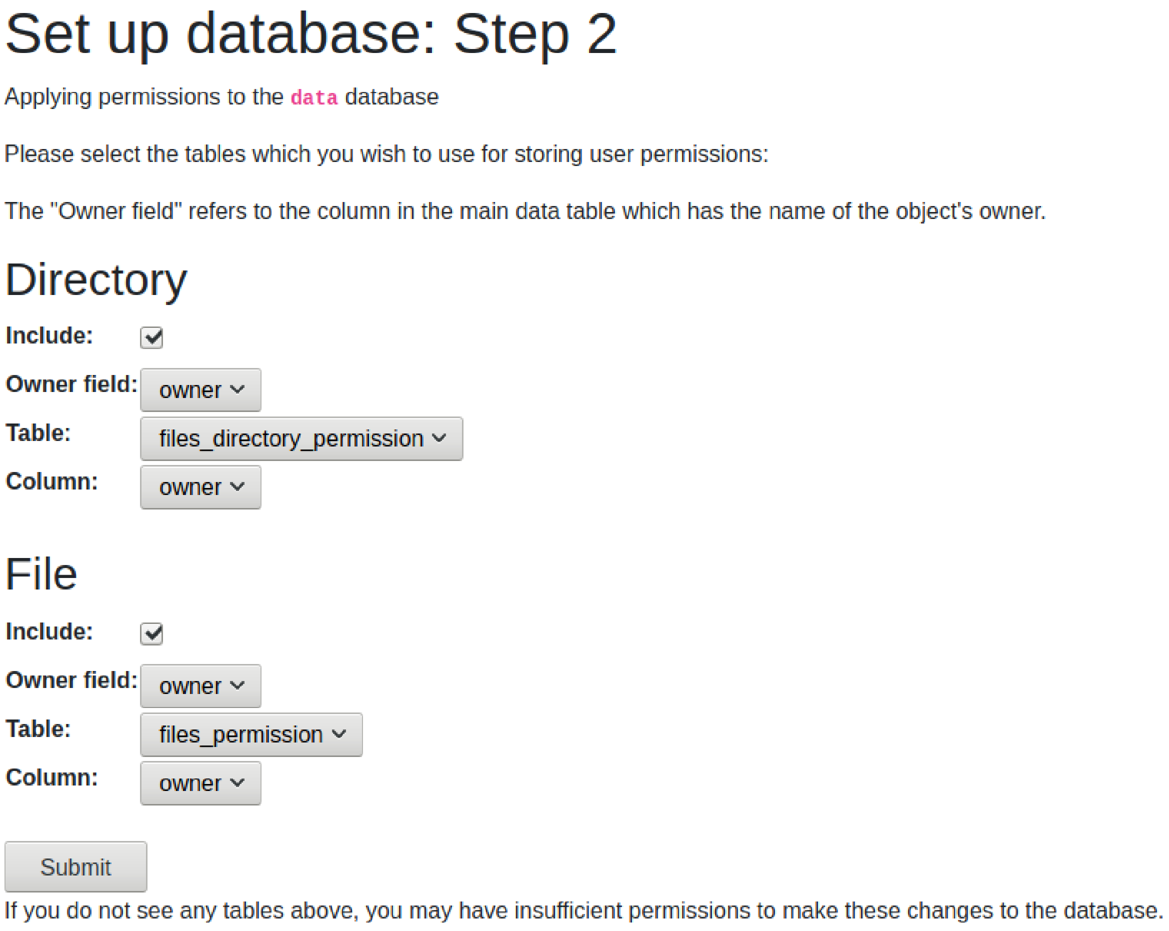
This process will set up permissions tables and apply row-level security to the `data` database.

If this is incorrect, please set the `DELEG_DATABASE` setting appropriately before continuing.

To continue to the next step, please click here.

Figure 4.1: Setup welcome screen

The next stage in the process (Figure 4.2) fetches data on the database tables from the Django configuration. It then queries the `information_schema` tables (see section 3.5) and presents a setup page for choosing which tables and fields should be used when implementing the row-level security policies.



Set up database: Step 2

Applying permissions to the **`data`** database

Please select the tables which you wish to use for storing user permissions:

The "Owner field" refers to the column in the main data table which has the name of the object's owner.

Directory

Include: ☑

Owner field: owner

Table: files_directory_permission

Column: owner

File

Include: ☑

Owner field: owner

Table: files_permission

Column: owner

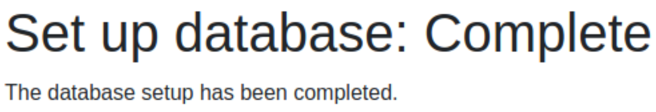
If you do not see any tables above, you may have insufficient permissions to make these changes to the database.

Figure 4.2: Table selection

Note the access control to this page is not critical (although probably desirable to avoid confusing users). If an unauthorised user attempts to access this page, there should not be any tables listed here (as the warning at the bottom of the previous image describes) due to insufficient privileges on the `information_schema` tables. Even if a user does have this access but is not authorised to create database policies, the following step will simply fail with a database permissions error. This is another illustration of the advantages of this approach over a traditional web application setup – **as long as the database access rights are set appropriately, the web app does not need extra security protection mechanisms**.

The user performing these steps does not need knowledge of Kerberos authentication, and since the application only displays tables which have appropriate foreign-key relationships in the database (see section 3.5), the potential for choosing an incorrect table is limited. Compared with manually creating the access control rules using SQL statements, this system is far more straightforward to set up.

Finally, a "completion" screen is shown (Figure 4.3, which would include any error messages from the database engine).



Set up database: Complete

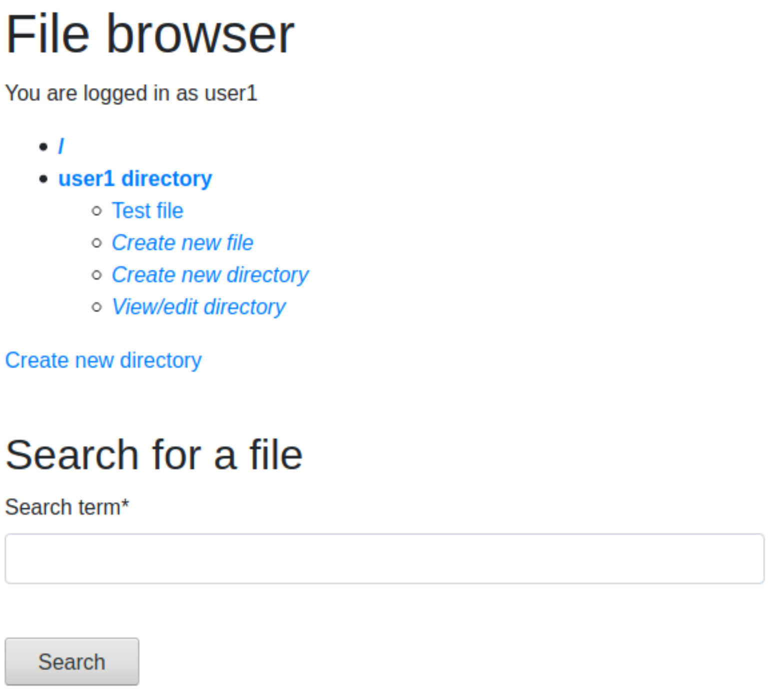
The database setup has been completed.

Figure 4.3: Setup complete

4.2.2 Demonstration Application (File Browser)

As a practical illustration of how the technology works, I developed a file browser application allowing a user to create, view and edit "files" (which, for demonstration purposes, are simple text entries stored in the database) and assign permissions to other users.

The basic interface (Figure 4.4) shows a hierarchical directory structure with files and directories, and options to create new records:



File browser

You are logged in as user1

- */*
- **user1 directory**
  - Test file
  - *Create new file*
  - *Create new directory*
  - *View/edit directory*

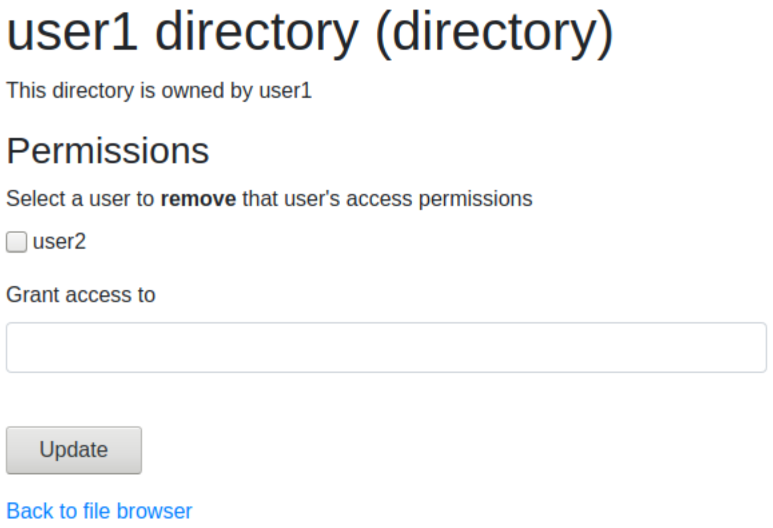
Create new directory

Search for a file

Search term*

Figure 4.4: File browser homepage

Each file and directory has a permissions interface (Figure 4.5) visible to the registered "owner" of that record, allowing that user to add or remove permission entries.



user1 directory (directory)

This directory is owned by user1

Permissions

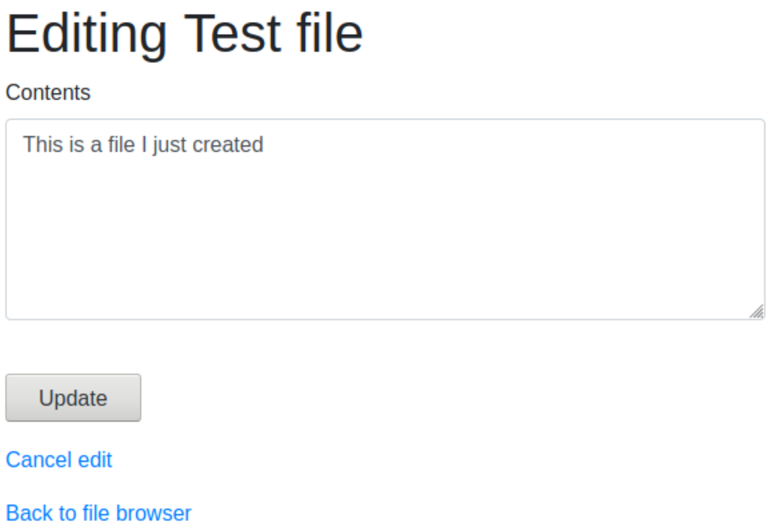
Select a user to **remove** that user's access permissions

□ user2

Grant access to

Figure 4.5: Permissions editing interface

In addition, the file owner can easily update file contents, as in Figure 4.6 (other users with permission to access the file have read-only access – this could easily be augmented to allow a more complex access control mechanism for assigning separate read/write permissions to users).



Editing Test file

Contents

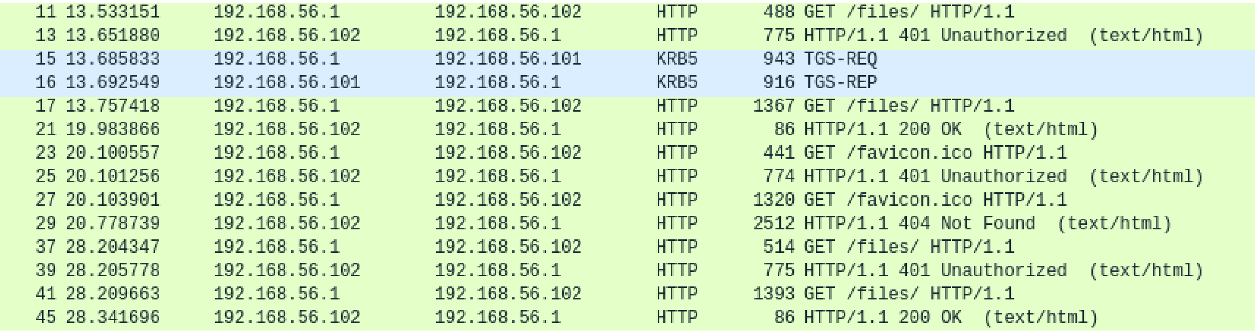
This is a file I just created

Figure 4.6: File editing interface

This demonstrates **a usable application (with a simple interface for users) that works entirely by using and delegating Kerberos tickets, as described in the success criteria**.

4.3 Protocol Execution

Using an approach without session cookies, a simple request to the webserver produces a sequence of packets such as in Figure 4.7.



```
   11 13.533151     192.168.56.1         192.168.56.102       HTTP      488 GET /files/ HTTP/1.1
   13 13.651880     192.168.56.102       192.168.56.1         HTTP      775 HTTP/1.1 401 Unauthorized  (text/html)
   15 13.685833     192.168.56.1         192.168.56.101       KRB5      943 TGS-REQ
   16 13.692549     192.168.56.101       192.168.56.1         KRB5      916 TGS-REP
   17 13.757418     192.168.56.1         192.168.56.102       HTTP     1367 GET /files/ HTTP/1.1
   21 19.983866     192.168.56.102       192.168.56.1         HTTP       86 HTTP/1.1 200 OK  (text/html)
   23 20.100557     192.168.56.1         192.168.56.102       HTTP      441 GET /favicon.ico HTTP/1.1
   25 20.101256     192.168.56.102       192.168.56.1         HTTP      774 HTTP/1.1 401 Unauthorized  (text/html)
   27 20.103901     192.168.56.1         192.168.56.102       HTTP     1320 GET /favicon.ico HTTP/1.1
   29 20.778739     192.168.56.102       192.168.56.1         HTTP     2512 HTTP/1.1 404 Not Found  (text/html)
   37 28.204347     192.168.56.1         192.168.56.102       HTTP      514 GET /files/ HTTP/1.1
   39 28.205778     192.168.56.102       192.168.56.1         HTTP      775 HTTP/1.1 401 Unauthorized  (text/html)
   41 28.209663     192.168.56.1         192.168.56.102       HTTP     1393 GET /files/ HTTP/1.1
   45 28.341696     192.168.56.102       192.168.56.1         HTTP       86 HTTP/1.1 200 OK  (text/html)
```

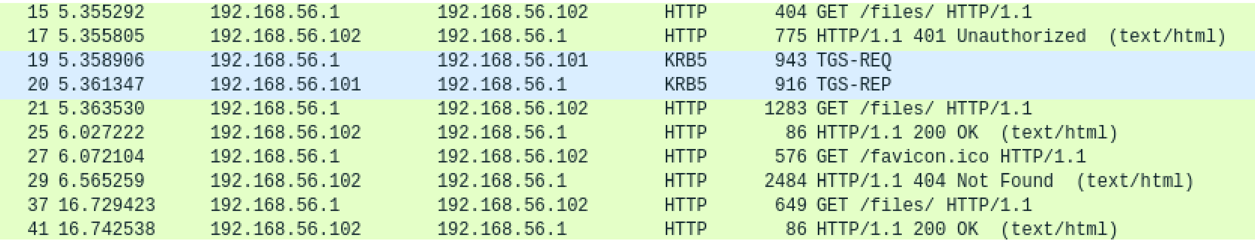
Figure 4.7: Packet sequence **without** session cookies

The sequence of packets in Figure 4.7 shows:

- Packet 11: an initial HTTP request from the client (192.168.56.1) to the web server (192.168.56.102)
- Packet 13: an HTTP 401 response asking the user to authenticate
- Packets 15 and 16: the client requesting and obtaining a Kerberos service ticket from the Kerberos server (192.168.56.101)
- Packet 17: another HTTP request to the web server **with** this ticket
- Packet 21: the client receives the page contents as an HTTP response

Each of the following requests in the sequence of packets repeats this cycle: although the client already has the service ticket, it must still make an initial HTTP request, receive a response requesting authentication and send another HTTP request with the service ticket. This is clearly inefficient, and so session cookies (as described in section 3.3) are used in this application.

Once session cookies are enabled, the sequence of packets appears in Figure 4.8.



```
   15 5.355292      192.168.56.1         192.168.56.102       HTTP      404 GET /files/ HTTP/1.1
   17 5.355805      192.168.56.102       192.168.56.1         HTTP      775 HTTP/1.1 401 Unauthorized  (text/html)
   19 5.358906      192.168.56.1         192.168.56.101       KRB5      943 TGS-REQ
   20 5.361347      192.168.56.101       192.168.56.1         KRB5      916 TGS-REP
   21 5.363530      192.168.56.1         192.168.56.102       HTTP     1283 GET /files/ HTTP/1.1
   25 6.027222      192.168.56.102       192.168.56.1         HTTP       86 HTTP/1.1 200 OK  (text/html)
   27 6.072104      192.168.56.1         192.168.56.102       HTTP      576 GET /favicon.ico HTTP/1.1
   29 6.565259      192.168.56.102       192.168.56.1         HTTP     2484 HTTP/1.1 404 Not Found  (text/html)
   37 16.729423     192.168.56.1         192.168.56.102       HTTP      649 GET /files/ HTTP/1.1
   41 16.742538     192.168.56.102       192.168.56.1         HTTP       86 HTTP/1.1 200 OK  (text/html)
```

Figure 4.8: Packet sequence **with** session cookies

The initial exchange is effectively identical to the previous setup. However, the HTTP response in packet 25 now includes a session cookie, which is included by the browser in the subsequent requests. The client can then request resources as “standard” HTTP requests (with no need to repeat the negotiation process), and so the efficiency of the system is increased (see section 4.6).

See Appendix D for a protocol trace from the client’s perspective. This includes setting a session cookie, but the negotiation process illustrated is the same whether or not session cookies are used.

From these packet traces, **authentication using Kerberos tickets can be seen to be working**.

4.4 Creation of Database Rules

The row-level security rules created by the database setup process can be inspected by directly logging in to the database server and using the PostgreSQL `\dp` command to list all privilege configurations on the current database. After running the setup application, the output is as follows:

(For space reasons, the “Schema” and “Column privileges” columns, which are included in the PostgreSQL output but do not show any useful information, have been omitted from this document. Similarly, some additional line breaks have been added to the raw output.)

```
filedb2=# \dp
                                                     Access privileges
               Name                 |   Type   |  Access privileges    |                         Policies
------------------------------------+----------+-----------------------+-----------------------------------------------------------
 django_migrations                  | table    |                       |
 django_migrations_id_seq           | sequence | django=rwU/django    +|
                                    |          | =rU/django            |
 files_directory                    | table    | django=arwdDxt/django+| files_directory_view (r):                                +
                                    |          | =arw/django           |   (u): ( SELECT files_directory_permitted(
                                    |          |                       |          files_directory.id)
                                    |          |                       |          AS files_directory_permitted)                   +
                                    |          |                       | files_directory_view_owner:                              +
                                    |          |                       |   (u): ((owner)::text = SESSION_USER)                    +
                                    |          |                       | files_directory_insert (a):                              +
                                    |          |                       |   (c): true
 files_directory_id_seq             | sequence | django=rwU/django    +|
                                    |          | =rU/django            |
 files_directory_permission         | table    | django=arwdDxt/django+| files_directory_permission_owner:                        +
                                    |          | =ard/django           |   (u): (directory_id IN ( SELECT files_directory.id      +
                                    |          |                       |    FROM files_directory                                  +
                                    |          |                       |   WHERE ((files_directory.owner)::text = SESSION_USER)))+
                                    |          |                       | files_directory_permission_view (r):                     +
                                    |          |                       |   (u): ( SELECT files_directory_permitted(
                                    |          |                       |          files_directory_permission.directory_id)
                                    |          |                       |          AS files_directory_permitted)                   +
                                    |          |                       | files_directory_permission_insert (a):                   +
                                    |          |                       |   (c): (directory_id IN ( SELECT files_directory.id      +
                                    |          |                       |    FROM files_directory                                  +
                                    |          |                       |   WHERE ((files_directory.owner)::text = SESSION_USER)))
 files_directory_permission_id_seq | sequence | django=rwU/django    +|
                                   |          | =rU/django            |
 files_file                        | table    | django=arwdDxt/django+| files_file_view (r):                                    +
                                   |          | =arw/django           |   (u): ( SELECT files_file_permitted(files_file.id)
                                   |          |                       |          AS files_file_permitted)                       +
                                   |          |                       | files_file_view_owner:                                  +
                                   |          |                       |   (u): ((owner)::text = SESSION_USER)                   +
                                   |          |                       | files_file_insert (a):                                  +
                                   |          |                       |   (c): true
 files_file_id_seq                 | sequence | django=rwU/django    +|
                                   |          | =rU/django            |
 files_permission                  | table    | django=arwdDxt/django+| files_permission_owner:                                 +
                                   |          | =ard/django           |   (u): (file_id IN ( SELECT files_file.id               +
                                   |          |                       |    FROM files_file                                      +
                                   |          |                       |   WHERE ((files_file.owner)::text = SESSION_USER)))     +
                                   |          |                       | files_permission_view (r):                              +
                                   |          |                       |   (u): ( SELECT files_file_permitted(
                                   |          |                       |          files_permission.file_id)
                                   |          |                       |          AS files_file_permitted)                       +
                                   |          |                       | files_permission_insert (a):                            +
                                   |          |                       |   (c): (file_id IN ( SELECT files_file.id               +
                                   |          |                       |    FROM files_file                                      +
                                   |          |                       |   WHERE ((files_file.owner)::text = SESSION_USER)))
 files_permssion_id_seq            | sequence | django=rwU/django    +|
                                   |          | =rU/django            |
(10 rows)
```

This demonstrates the **correct operation of the setup procedure**, with suitable constraints on viewing records based on ownership of information that match the SQL queries made in the application code.

4.5 Potential Attacks and Vulnerabilities

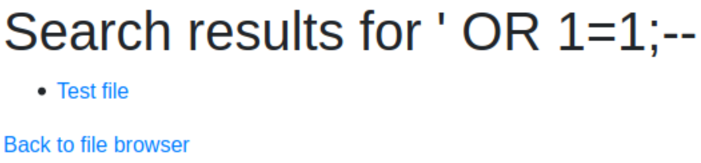
As the most significant aspect of this technology is its security implications, I completed an evaluation of possible security vulnerabilities and how they would affect a system such as the one demonstrated.

4.5.1 SQL Injection

The search feature of the demonstration app helps demonstrate the security of the Kerberos delegation approach in that it contains a (deliberate) SQL injection vulnerability. The search function performs the following query (where `term` is user-provided):

```
query = "SELECT * FROM " + table + "
         WHERE CONTENTS LIKE '%%" + term + "%%'"
```

This can trivially be exploited by passing a search term such as `' OR 1=1;--` (as detailed in section 1.1.1), and the web app's response (Figure 4.9) shows that it is indeed vulnerable to SQL injection:



Search results for ' OR 1=1;--

- Test file

Back to file browser

Figure 4.9: SQL injection in the file browser app

However, **this does not disclose any information which `user1` cannot access anyway**. Counting all files in the database (running the count as the database super-user) shows that there are actually many more files than were extracted using the SQL injection:

```
filedb2=# SELECT owner, COUNT(*) FROM files_file GROUP BY owner;
 owner  | count
--------+-------
 user2  |     3
 django |     6
 user1  |     1
(3 rows)
```

This therefore demonstrates how **the presence of an SQL injection vulnerability in the constructed application does not allow unauthorised access to data or weaken the threat model of the system**.

The web app simply acts as a "window" onto the database to execute queries with the privileges of the user logged into the web app, and a user could just as easily use the same Kerberos ticket to log into the database server directly (which could be considered a feature to provide expert uses with a safe SQL-based API). This contrasts favourably with traditional systems where the web app itself is trusted to access the data of any user and SQL injection vulnerabilities must be avoided.

4.5.2 Theft of Long-Term Secrets

As discussed in section 1.1, the possible presence of passwords or other access tokens in source code can cause a risk of improper access if the secrets are not adequately protected. In many cases, database passwords such as these can be used from anywhere on the internet to gain access to all records in the database.

In this application, all authentication to the database is done using Kerberos tickets, and so there is no possibility of directly stealing a database key. However, it is important to note that the approach is still not entirely free of long-term secrets since they are still used in two significant contexts:

- Django's `SECRET_KEY` is stored in the Django filesystem and used for signing session cookies. Although this is certainly a potential security risk, in this case the Django session is only used for storage of database metadata between requests in the database setup application. Therefore, being able to break or forge these cookies would not provide a substantial advantage to an attacker since the PostgreSQL permissions would stop any genuine malicious activity on the database.
- Apache's `mod_session` module is used to provide a session cookie that holds details of a user's authentication status. This represents a compromise to provide more usability at the expense of a slight loss in security, since if the signing key for these cookies were stolen an attacker would be able to impersonate any user who still had a cached ticket on the server. While still a clear improvement on a single master password (which if stolen could lead to **any** user being impersonated), it is nevertheless still a potential issue in some environments. It would easily be possible to dispense with this session cookie (if required) for security, but at the expense of a substantial performance reduction (see section 4.6).

  This key is also considerably harder for an attacker to obtain since it is not stored anywhere in the directory structure of the web app itself and incorporates a randomly generated element which means that the key changes (at least) when the server is restarted. Therefore, in all likelihood, stealing this key is not a practical attack against the system.

4.5.3 Data Obtainable from the Database

One consequence of using delegated Kerberos tickets for accessing data is that users may well be given direct access to the database (which they potentially would not have had otherwise). This means that access control must be set up in the database itself, and this may offer a less granular permission model than could be specified if it were controlled by the web app (although the use of SQL stored procedures would allow a relatively broad set of options).

An example of a (minor) data leak caused by this setup is that it is possible to determine whether an object ID has already been allocated based on whether an `INSERT` operation with that ID succeeds or fails. On the assumption that these IDs are opaque identifiers such as sequential numbers (as they are in the example application), this is not significant, but if the ID contained useful information about the object then this would be a possible covert channel to leak information.

4.5.4 Theft of Credential Caches

If a malicious user could execute a process on the web server (under the same system user account as the server application), that process would gain the same access to the credential cache directory that the web server application has.

Because of the patch added to `mod_auth_gssapi` and the directory permissions being set as described in section 3.3, a malicious user would not have any other credential cache filenames and could only enumerate files in the credential cache directory by doing a brute-force search. Given that each ticket is only useful for a small length of time, adding a suitably long random string to the filename should ensure that it is not viable to obtain tickets. As further mitigations of the risks of credential caches being stolen:

- The tickets stored in these caches are service tickets, which can only be used on this web server and any services which it is allowed to delegate to (see section 2.4). Unlike a password or a Kerberos ticket-granting ticket, they cannot be used on arbitrary other systems.
- The only valid tickets that will be on the web server at all are those of users who have recently logged into it. Other users in the organisation will not have any "live" ticket data cached on the webserver, and so **their data is not vulnerable even if the web server is compromised**.

Following some discussion with the maintainer of `mod_auth_gssapi`, I was alerted to the possibility of other possible side-channel attacks to access credential caches [31]. However, there are possible mitigations against these (such as ensuring `/proc` was not mounted on the web server), and there is the potential for future work into securing stored credential caches (see section 5.2).

4.5.5 Compromise of the Django Application

A malicious user who can compromise the Django application itself would be able to see all data flowing through the compromised session, since the web app must have access to the data to display it.

The credential caches of users actively using the compromised application would also be available the attacker, subject to the same mitigation that these are server-restricted service tickets. As in section 4.5.4, a brute-force attack to obtain all credential caches on the system would be technically possible, but should be infeasible given the time taken. There is still no method of obtaining data on behalf of users who have not recently logged into the system.

4.5.6 Compromise of the Whole Webserver

Given a root-level compromise of the webserver, all cached credentials would be available to be stolen by an attacker since the root user can override file system access control policies. Any of these tickets could then be used to fetch data from the database, but note that two of the mitigating factors stated in section 4.5.4 still apply: **these are still service tickets which can only be used on a limited number of systems, and there is no effect on the security of users who have not recently logged into the web server**.

4.6 Performance Evaluation

A legitimate concern in building this project is the risk of degraded performance. Since the HTTP Negotiate protocol requires a multi-way "handshake" before starting data transfer, a naïve implementation of this project could have much slower access times than a traditional web app.

Therefore, I measured the access times to reach the file browser main page (which involves requesting data from the database) in each of the following configurations:

- No database security at all (unlikely in reality, but serves as a baseline)
- Database authentication with a master password (typical real-world model)
- Access via HTTP-Negotiate and delegated Kerberos tickets, using ticket caching on the web server and session cookies (as described in section 3.3, and implemented in this project)
- Access via HTTP-Negotiate and delegated Kerberos tickets, but **without** use of session cookies

Each of these access times was recorded 200 times in succession to allow summary statistics (median and quartiles) to be calculated. I measured the connection times (in both cases to a VM on my laptop running the web server) from my laptop and from a Microsoft Azure VM via an SSH tunnel (as in Figure 4.10) to give some simulation of the effect of network latency.

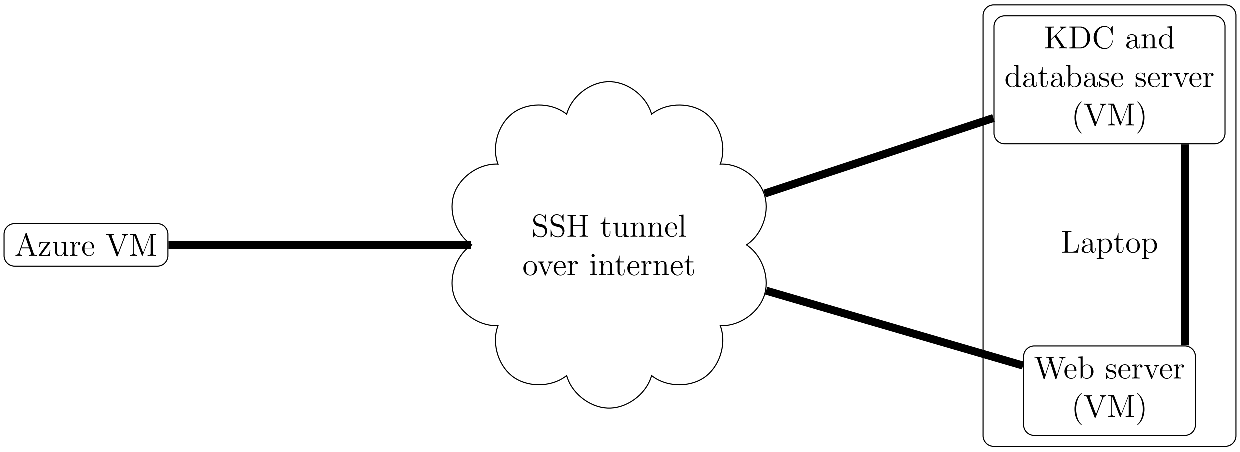


Figure 4.10: Performance testing setup for Azure VM

Response Times (milliseconds)

		Min.	Quartile 1	Median	Quartile 3	Max.
Laptop	No security	21.1	23.4	24.8	26.4	57.0
	Master password	20.7	25.6	27.8	29.8	377
	Kerberos with session cookies	19.7	24.1	26.2	29.2	74.3
	Kerberos without session cookies	104	169	210	824	2070
Azure VM	No security	38.2	41.9	43.3	46.3	531
	Master password	38.7	41.7	42.7	44.3	459
	Kerberos with session cookies	38.2	40.5	41.5	43.2	111
	Kerberos without session cookies	125	183	209	838	2160

Note that the "Kerberos with session cookies" data **excludes** the first connection of the session, where the cookie is obtained.

From the data above, there is a clear performance penalty from using HTTP negotiation without any form of session cookie, with the median access time rising from 20–45ms to over 200ms. There is also less significant, but noticeable, increase in access times when connecting from a remote location (particularly in the case of the maximum times, which would be expected given the possibility of packet loss or other connection issues).

To more easily visualise the data and analyse the performance of the system **with** session cookies enabled, I created box-plots (after removing outliers beyond 1.5×IQR of the quartiles) of first three rows of data for each of the two connection sources in Figure 4.11 and Figure 4.12.

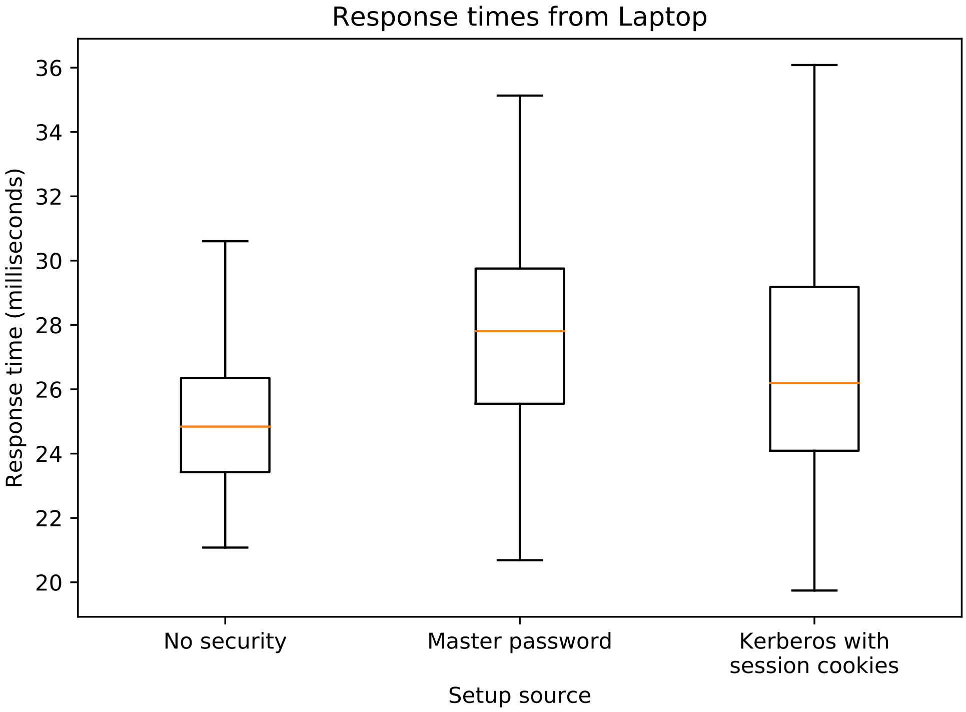


Figure 4.11: Response times from laptop

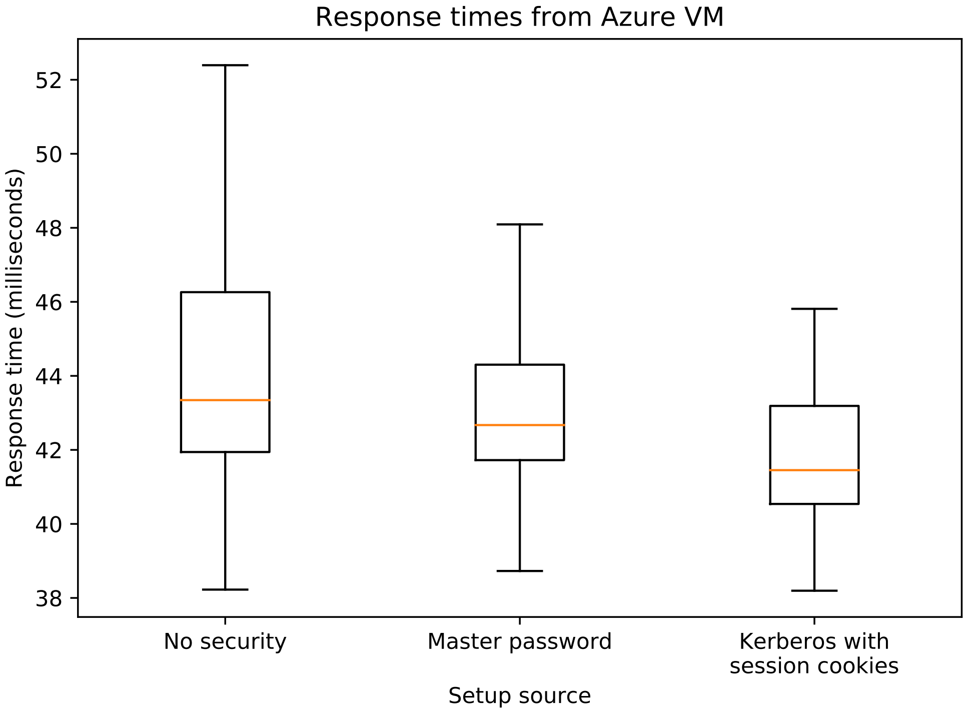


Figure 4.12: Response times from Azure VM

From these plots, there is no indication of a reduction in performance compared with the use of a master database password (in fact the median values suggest a slight speed improvement, although the natural variation in the data means that the two approaches probably offer roughly equal performance in practice).

In both sample sets, the **mean response time** for the "Kerberos with session cookies" approach was also below that of the "master password" approach. Full timing data is included in the `testing` directory of the code repository.

Although there is the overhead of the initial request to obtain the session cookie before the response times above are achievable, the same would apply to any single sign-on system (where a user is normally redirected to an external server for authentication, as in the OAuth2 protocol). Therefore, **the performance overheads associated with this system should not be noticeably greater than in existing single sign-on systems**.

The project demonstrates an application using delegated Kerberos tickets for secure access to a database, with no noticeable performance penalty.

Chapter 5

Conclusion

5.1 Summary of Work Completed

Having gained an understanding of Kerberos and related issues, I produced a Django web application which both provides an interface for setting up appropriate SQL permissions on a database (for use with delegated Kerberos tickets) and demonstrates a use of the setup with a file browser application. I also wrote two patches to existing software codebases (PostgreSQL and `mod_auth_gssapi`) to add functionality where this is useful for the project's application.

5.2 Future Work

Further research on methods of storing delegated tickets is an obvious potential extension to this project. Using kernel keyrings rather than a filesystem-based credential cache would be one possible strategy, or alternatively a method of encrypting the ticket at rest (and storing the decryption key in an HTTP cookie) may be effective. This could offer a number of security advantages if the process-privacy issue can be resolved and the libraries (particularly `mod_auth_gssapi`) suitably extended to work with it.

There is also clearly potential to use delegated Kerberos tickets for services beyond database servers. For example, extending the system to work with a network-based filestore using NFS would give a more flexible system than one which is specifically designed to access a database server, and introduce some nuances which are not covered by the scope of the current project (such as the fact that the OS kernel of the web server would likely need to be involved in the process of connecting to the NFS server).

5.3 Lessons Learned

As well as an exploration into the setup and use of Kerberos-based applications, this project also served as my first major experience of working on an existing open-source codebase. The time required to do this was fairly significant compared with the eventual volume of code written (in the form of the patches), and required preparatory work to understand the code structure and method calls in the hope of avoiding introducing any new bugs.

In some cases, I also began work on project components without a full enough understanding of the options available. For example, my initial implementation of the database security setup (using views and abstract classes), while functional, was inelegant and I would have used row-level security constructs from the start if I had developed a proper understanding of the options available at an earlier stage.

The process of developing the software, and discussions with my supervisor and the open-source project maintainers, was also a very useful learning experience, especially with regard to identifying potential security vulnerabilities in the system.

Bibliography

[1] Markus Kuhn (2020): Ideas for Student Projects – Kerberos-based single sign-on with delegation for web applications: https://www.cl.cam.ac.uk/~mgk25/project-ideas/#http-gssapi (accessed 12/05/2021) [N.B. link contains a personal identifier]

[2] OWASP *Top 10 Web Application Security Risks* list: https://owasp.org/www-project-top-ten/ (accessed 21/10/2020)

[3] GitGuardian: Git Security Scanning & Secrets Detection: https://www.gitguardian.com/ (accessed 22/04/2021)

[4] Data Protection Act 2018: Section 157: https://www.legislation.gov.uk/ukpga/2018/12/section/157/enacted (accessed 10/02/2021)

[5] Oracle OAuth Guide: API Gateway OAuth 2.0 Authentication Flows: https://docs.oracle.com/cd/E50612_01/doc.11122/oauth_guide/content/oauth_flows.html (accessed 08/04/2021)

[6] IBM Open Liberty Docs: Configuring SPNEGO authentication: https://openliberty.io/docs/21.0.0.2/configuring-spnego-authentication.html#configure-client-app (accessed 13/05/2021)

[7] Eric Baize and Denis Pinkas (December 1998), *The Simple and Protected GSS-API Negotiation Mechanism* (RFC 2478): https://tools.ietf.org/html/rfc2478 (accessed 22/04/2021)

[8] John Linn (January 2000), *Generic Security Service Application Program Interface Version 2, Update 1* (RFC 2743): https://tools.ietf.org/html/rfc2743 (accessed 26/04/2021)

[9] John Wray (January 2000), *Generic Security Service API Version 2 : C-bindings* (RFC 2744): https://tools.ietf.org/html/rfc2744.html (accessed 26/04/2021)

[10] Larry Zhu et. al. (October 2005), *The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism* (RFC 4178): https://tools.ietf.org/html/rfc4178 (accessed 26/04/2021)

[11] Karthik Jaganathan, Larry Zhu and John Brezak (June 2006), *SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows* (RFC 4559): https://tools.ietf.org/html/rfc4559 (accessed 04/04/2021)

[12] MIT Kerberos documentation: Database Types: https://web.mit.edu/kerberos/krb5-devel/doc/admin/conf_ldap.html (accessed 05/05/2021)

[13] Simo Sorce (February 2012), Kerberos: delegation and s4u2proxy: https://ssimo.org/blog/id_011.html (accessed 02/02/2021)

[14] Ioannis Kollitidis (March 2020), Unconstrained Delegation: https://johnkol.com/unconstrained-delegation/ (accessed 27/04/2021)

[15] MIT Kerberos documentation: kdc.conf: https://web.mit.edu/kerberos/www/krb5-devel/doc/admin/conf_files/kdc_conf.html (accessed 05/04/2021)

[16] Microsoft Corporation (May 2014), *Kerberos Protocol Extensions: Service for User and Constrained Delegation Protocol*: https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-SFU/[MS-SFU]-140515.pdf (accessed 05/04/2021)

[17] Microsoft Corporation (February 2019), Attribute msDS-AllowedToDelegateTo: https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-ada2/86261ca1-154c-41fb-8e5f-c6446e77daaa (accessed 05/05/2021)

[18] MIT Kerberos documentation: Credential cache: https://web.mit.edu/kerberos/krb5-devel/doc/basic/ccache_def.html (accessed 25/04/2021)

[19] PostgreSQL documentation: CREATE FUNCTION: https://www.postgresql.org/docs/current/sql-createfunction.html#SQL-CREATEFUNCTION-SECURITY (accessed 22/02/2021)

[20] PostgreSQL documentation: CREATE VIEW: https://www.postgresql.org/docs/13/sql-createview.html (accessed 05/05/2021)

[21] PostgreSQL documentation: CREATE RULE: https://www.postgresql.org/docs/current/sql-createrule.html (accessed 22/02/2021)

[22] Nick Kew (2007), The Apache modules book : Application development with Apache (1st ed., Prentice Hall open source software development series)

[23] Django documentation (including the sub-pages linked from here): https://docs.djangoproject.com/en/3.1/ (accessed 27/04/2021)

[24] Simon Josefsson (October 2006), *The Base16, Base32, and Base64 Data Encodings* (RFC 4648): https://tools.ietf.org/html/rfc4648 (accessed 30/04/2021)

[25] minifrin (June 2018), Revision 1834371 to Apache web server codebase: https://svn.apache.org/viewvc?view=revision&revision=1834371 (accessed 30/04/2021)

[26] PostgreSQL documentation: CREATE POLICY: https://www.postgresql.org/docs/13/sql-createpolicy.html (accessed 23/04/2021)

[27] PostgreSQL documentation: GSSAPI Authentication: https://www.postgresql.org/docs/13/gssapi-auth.html (accessed 19/04/2021)

[28] PostgreSQL documentation: The Information Schema: https://www.postgresql.org/docs/13/information-schema.html (accessed 11/05/2021)

[29] pgsql-hackers mailing list, *PATCH: Add GSSAPI ccache_name option to libpq*: https://www.postgresql.org/message-id/flat/183cb0c3-30a9-149e-5403-fd36800e8c2b%40gmail.com (accessed 06/05/2021) [N.B. link contains a personal identifier]

[30] Apache web server source code: AcceptPathInfo: https://github.com/apache/httpd/blob/587d17015167f442c98efaf497504fe3825a3fe7/server/core.c#L3394 (accessed 02/05/2021)

[31] Pull request to `mod_auth_gssapi` on GitHub: `https://github.com/gssapi/mod_auth_gssapi/pull/250` (accessed 12/05/2021) [N.B. link contains a personal identifier]

Appendix A

Apache Configuration

The following is the complete .htaccess file used to configure options in the Apache web server. This primarily demonstrates the setup of the mod_auth_gssapi module, including the new Secure mode for the GssapiDelegCcacheUnique parameter that was added as part of this project.

```
# Enable GSSAPI authentication
AuthType GSSAPI
AuthName "Kerberos site authentication"

# Only allow Kerberos authentication (not NTLM etc.)
GssapiAllowedMech krb5

# Enable delegation support
GssapiUseS4U2Proxy On

# Specify where keytab file for server principal is located
GssapiCredStore keytab:/var/www/apache.keytab
GssapiCredStore client_keytab:/var/www/apache.keytab

# Save delegated tickets to directory, and add random suffix to make
# them harder to brute-force
GssapiDelegCcacheDir /var/run/apache2/clientcaches
GssapiDelegCcacheUnique Secure

# Strip realm off user name (after Kerberos library has verified it)
GssapiLocalName On

# Use a session cookie to avoid having to negotiate on each request
GssapiUseSessions On
Session On
SessionCookieName session path=/;httponly;samesite=strict

# Only allow authenticated users to access the application
Require valid-user
```

Appendix B

Django Database Router

The code below shows the routing process for Django to determine which database to use for a given type of request:

```
class FileDBRouter:
    APP_LABEL = 'files'

    DATA_DB = settings.DELEG_DATABASE
    DEFAULT_DB = 'default'

    def db_for_read(self, model, **hints):
        """
        File access goes to 'data' database, otherwise default
        """
        if model._meta.app_label == self.APP_LABEL:
            return self.DATA_DB
        return self.DEFAULT_DB

    def db_for_write(self, model, **hints):
        """
        File access goes to 'data' database, otherwise default
        """
        if model._meta.app_label == self.APP_LABEL:
            return self.DATA_DB
        return self.DEFAULT_DB

    def allow_relation(self, obj1, obj2, **hints):
        """
        Only allow relations within a database, not across multiple
        databases
        """
        return ((obj1._meta.app_label == self.APP_LABEL) ==
                (obj2._meta.app_label == self.APP_LABEL))

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        """
        Only put files into the 'data' database, and everything else
        into the 'default' database
        """
        return (app_label == self.APP_LABEL) == (db == self.DATA_DB)
```

The main database (`DATA_DB`, whose name is stored in the general Django settings file as it is also required in the database setup procedure) contains the data to be protected by PostgreSQL access control. The “default” database (`DEFAULT_DB`) is used for all other records (primarily web app session information), and so this router directs queries from the file browser app to the PostgreSQL database, and other data which is internal to the web app to the other database.

Once this router is in place, the routing is transparent to the web app, which simply needs to use the Django models classes as normal. The router can be overridden in an application if necessary, and this is done on the `managedb` pages to allow setup of the database permissions.

(The code above is loosely based on several examples given in the Django documentation at `https://docs.djangoproject.com/en/3.1/topics/db/multi-db/`.)

Appendix C

mod_auth_gssapi Patch

The following `diff` extracts (slightly altered for line length in this report) show more detail on the changes made to the `mod_auth_gssapi` module.

Calling of the subroutine to create a random string:

```
-    ccname = apr_psprintf(pool, "%s/%s-XXXXXX", dir, escaped);
+    if (use_random) {
+        const char *rand_string = get_random_string(16, req->pool);
+        ccname = apr_psprintf(pool, "%s/%s-%s-XXXXXX", dir, escaped,
+                              rand_string);
+    } else {
+        ccname = apr_psprintf(pool, "%s/%s-XXXXXX", dir, escaped);
+    }
```

Implementing a multiple-choice option (rather than a simple on/off) flag:

(Based on the style used elsewhere in Apache code, for example in the `AcceptPathInfo` option [30])

```
 static const char *mag_deleg_ccache_unique(cmd_parms *parms,
                                            void *mconfig,
-                                           int on)
+                                           const char *arg)
 {
     struct mag_config *cfg = (struct mag_config *)mconfig;
-    cfg->deleg_ccache_unique = on ? true : false;
+
+    if (ap_cstr_casecmp(arg, "on") == 0) {
+        cfg->deleg_ccache_unique = true;
+        cfg->deleg_ccache_random = false;
+    }
+    else if (ap_cstr_casecmp(arg, "off") == 0) {
+        cfg->deleg_ccache_unique = false;
+        cfg->deleg_ccache_random = false;
+    }
+    else if (ap_cstr_casecmp(arg, "secure") == 0) {
+        cfg->deleg_ccache_unique = true;
+        cfg->deleg_ccache_random = true;
+    }
+    else {
+        return
+        "GssapiDelegCcacheUnique must be set to on, off or secure";
+    }
     return NULL;
 }
```

Appendix D

HTTP Negotiate Protocol Trace

```
$ curl -v --negotiate -u : krbsite.local/files/
*   Trying 192.168.56.102...
* TCP_NODELAY set
* Connected to krbsite.local (192.168.56.102) port 80 (#0)
> GET /files/ HTTP/1.1
> Host: krbsite.local
> User-Agent: curl/7.58.0
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< Date: Thu, 29 Apr 2021 16:21:40 GMT
< Server: Apache/2.4.41 (Ubuntu)
< WWW-Authenticate: Negotiate
< Content-Length: 460
< Content-Type: text/html; charset=iso-8859-1
<
* Ignoring the response-body
* Connection #0 to host krbsite.local left intact
* Issue another request to this URL: 'http://krbsite.local/files/'
* Found bundle for host krbsite.local: 0x56331d9c8ad0 [can pipeline]
* Re-using existing connection! (#0) with host krbsite.local
* Connected to krbsite.local (192.168.56.102) port 80 (#0)
* Server auth using Negotiate with user ''
> GET /files/ HTTP/1.1
> Host: krbsite.local
> Authorization: Negotiate YIICewYGKwYBBQUCoIICbzCCAmugDTALBgkqhkiG9...
> User-Agent: curl/7.58.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Thu, 29 Apr 2021 16:21:40 GMT
< Server: Apache/2.4.41 (Ubuntu)
< WWW-Authenticate: Negotiate oYG3MIG0oAMKAQChCwYJKoZIhvcSAQICooGfBIG...
< Set-Cookie: session=MagBearerToken=fGZU6v8uKhalBbmvCLOYfeHsN5JLLTg3...;
  path=/;httponly;samesite=strict
```

```
< Cache-Control: no-cache, private
< Content-Length: 2031
< X-Frame-Options: DENY
< X-Content-Type-Options: nosniff
< Referrer-Policy: same-origin
< Vary: Accept-Encoding
< Content-Type: text/html; charset=utf-8
```

[The page content now follows]

Appendix E

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Kerberos-based single sign-on with delegation for web applications

Candidate 2428G

Originator: Dr Markus Kuhn

October 22, 2020

Project Supervisor: Dr Markus Kuhn

Director of Studies: Dr Robert Harle

Overseers: Professor Frank Stajano and Dr Amanda Prorok

Project Description

Many web applications are built around databases, and traditionally the application itself has full access to any database records and controls access to them by users. This means that security vulnerabilities in the application or framework can result in incorrect access or modification to data, with potentially severe consequences – a simple vulnerability in a custom-written web application could allow an attacker to access other site users personal data.

Kerberos authentication works on the basis of each user getting a "ticket" which can be used to gain access to other systems, and so this project aims to set up a system where users can authenticate to a web application via Kerberos, and where there is no "global secret" that allows the web application to access all database records. Therefore, as long as the database access rights are set correctly, there is no need for the front-end to perform access control.

The project will involve adding to the Django framework (in the form of a plugin) so that it can receive authenticate a user via a Kerberos ticket, pass that ticket on to a database, and then access that users data from the database. Basic Kerberos authentication functionality already exists in Django modules, but this is only designed for authenticating to the web app itself and not passing a Kerberos ticket onto the database.

This will mean that a user can obtain a Kerberos ticket, then use that to authenticate to the website (using the HTTP Negotiate authentication scheme). The same ticket could be used for multiple different sites that are set up in this way, and so it effectively forms a single sign-on system. The web server itself should not have access to database records except via the Kerberos tickets.

To the user, this will mean that they can provide a Kerberos ticket to any site which is part of this system (and is in the same Kerberos "realm"). Quite apart from the standard benefits of a single sign-on system (only one password required for a whole set of sites, which means that users are more likely to be able to remember a strong password and less likely to record it using insecure methods), this would mean that SQL injection (a type of command injection, which is currently number 1 in the OWASP *Top 10 Web Application Security Risks* list) would not result in any data being disclosed beyond what a user had access to anyway. This is because the database itself would be implementing the security restrictions, and eliminates a significant vulnerability faced by many web applications.

An important aspect of the project is how Kerberos tickets are stored on the web server. Existing technologies often simply store all tickets in a directory on the server, and this makes them vulnerable to any other site running on the same system being able to read and use the tickets. By investigating and building a better means for storing tickets, a compromised site should not mean that all other web sites on the server are automatically also compromised.

Starting Point

Kerberos support for Apache already exists (via the `mod_auth_gssapi` module and the older `mod_auth_kerb` module), and this includes some ability to delegate tickets although with drawbacks (particularly that tickets to be delegated are placed in a directory which is accessible to any process running as the webserver user). Django similarly includes Kerberos extensions (such as `django-gssapi`), but these do not directly offer any delegation facilities. As a starting point, I will investigate these and build on the functionality which they already offer.

PostgreSQL already has support for Kerberos and this can be used as a back-end database system.

I have some knowledge of authentication systems and general security topics from the 1B Security course, and of C programming from the 1B Programming in C and C++ course. I have also had general experience with web application setup (although not the specific technologies involved here) through the 1B Group Project and work done outside the Tripos.

In addition, over the summer vacation I have done some experimentation with Kerberos authentication to set up a virtual machine using the existing Kerberos functionality of Apache and PostgreSQL (logging in separately to the two applications without any delegation of tickets).

Work to be done

1. Familiarity with the structure of Kerberos authentication (and MIT Kerberos specifically)
2. Familiarity with Apache web server development (as modifications to the Apache modules may be necessary)
3. Setting up a environment with a suitable web application and Kerberos server
4. Implementation and testing of the authentication functionality
5. Modularising the setup to work as a Django module
6. Evaluation of the performance and security of the system

Success Criteria

- The project will consist of an add-on to the Django framework which offers Kerberos authentication.
- A user must be able to log into the web application by presenting a suitable Kerberos ticket, which the application then uses to fetch data from a database.
- The application must not have access to the database other than by Kerberos tickets (so a given piece of data cannot be fetched without a Kerberos ticket corresponding to a principal who has access to that data). This means that a given user cannot access data which they are not authorised to have access to, even in the presence of web app vulnerabilities.

Evaluation

In order to evaluate the functionality, I will develop a basic web application which allows a user to manage an access control list associated with a resource they have uploaded, so as to permit or deny other users access to it. The aim of the project is to enforce this access policy on the SQL server, so that even in the presence of a vulnerability in the web app front-end where arbitrary SQL code could be executed by a user, the access control policy would not be broken.

As a result of this, it is important that the SQL server always “agrees” with the web app on who is accessing a page. If this can be demonstrated to be the case, then security vulnerabilities in the web app would not allow and inappropriate access to data, because the authorisation (and authentication) steps would all be performed by the database system.

Uses for this system could include web-based file systems (where a user can upload files and grant certain other users permission to see it), with each user registered as a separate “user” in the SQL database. The web app itself would not perform any security and would merely act as a front-end to pass the user’s Kerberos ticket onto the database, where authentication and authorisation would occur.

Therefore, the evaluation will take the form of a security analysis of the system and how vulnerable it is to attacks, in particular via SQL injection (as well as also potentially looking at the effect if another site running on the same web server is compromised). In addition, I will measure the performance of the system to consider impacts on response times compared with a basic (non-Kerberos enabled) web app and database.

Timetable

Michaelmas Weeks 4 to 5 (29/10 to 11/11)

Set up a basic test environment for this project, with a web server and SQL database which can be queried by the site. At this stage, neither the web site itself nor its connection to the database would have any security.

Milestone: a test web application connecting to a database (without security)

Michaelmas Weeks 6 to 7 (12/11 to 25/11)

Investigate existing Kerberos authentication libraries, and use them to set up authentication to the site (so that the site can access the user's Kerberos principal name, for example, and pull this user's data from a database). This would be done with a connection to a "test" Kerberos KDC (running locally in a VM). If possible, this will include using existing ticket delegation functionality in Apache.

Milestone: authentication to the web application itself via Kerberos (using existing technologies)

Michaelmas Week 8 to Vacation Week 3 (26/11 to 23/12)

Work on adding and improving the ticket delegation functions of the system, so that a user to this (particular) application can supply a Kerberos ticket which is used to access database resources. This includes evaluating approaches to storing tickets, ensuring that the delegation process works properly, and fixing any issues that arise.

Milestone: able to log into a single web application and delegate the Kerberos ticket to a database

Vacation Weeks 4 to 7 (24/12 to 20/01)

Modularise the previously built delegation code, so that it can be used as a plug-in with any suitable site based on the Django framework. Test use of this plug-in and fix any further issues.

Milestone: Kerberos ticket delegation system as a web framework plug-in

Lent Weeks 1 to 2 (21/01 to 03/02)

Contingency time to fix any remaining issues and further test the system. Write mid-project report, and (if possible) begin writing up the project.

Milestone: project report

Lent Weeks 3 to 4 (04/02 to 17/02)

Continue to write up the project.

Look into possible improvements to further reduce the vulnerability of the system.

Lent Weeks 5 to 6 (18/02 to 03/03)

Finish draft write-up (continue working on extensions if ahead of schedule). Submit to supervisor for review.

Milestone: draft dissertation

Lent Weeks 7 to 8 (04/03 to 17/03)

Time for supervisor to read and comment on report.

Work on extensions if time permits (N.B. reduced time available due to Unit of Assessment).

Vacation Weeks 1 to 6 (18/03 to 28/04)

Modifications and changes to the report based on feedback.

Complete full write-up, aiming to be ready to submit at the start of Easter term. If further time is available, then continue to work on extensions.

Milestone: dissertation complete

Easter Weeks 1 to 2 (29/04 to 12/05)

Contingency time to deal with any last-minute issues and problems.

Deadline: Friday 14th May 2021

Resources required

I plan to use my own computer (quad-core 1.6GHz CPU, 12GB RAM, 1TB hard drive, Linux Mint OS).

Source code will be held in a GitHub repository, with a copy uploaded to Dropbox each day and weekly backups made to an external hard drive.

This means that, in the event of hardware or software failure, I will be able to transition work to a replacement machine or to the MCS. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

Possible extensions

- Investigate using the same SQL server to record web server logs – this would require the web server application to switch Kerberos tickets between the client's ticket (to access client resources) and its own ticket (which would have access to server log tables), and ensure that this process of switching did not impact on the security of the system.
- Investigate whether this could be extended to access files over NFS via the same process of ticket delegation from a web application.

Bibliography

- `https://www.cl.cam.ac.uk/~mgk25/project-ideas/#http-gssapi` (accessed 19/10/2020)
  The project suggestion which this proposal is based on

- `https://tools.ietf.org/html/rfc4178` (accessed 19/10/2020)
  RFC 4178 (The Simple and Protected GSS-API Negotiation Mechanism), which describes the overall standard for Kerberos authentication to web applications

- `https://tools.ietf.org/html/rfc4559` (accessed 19/10/2020)
  RFC 4559 (SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows), describing a Microsoft-based system which offers similar functionality

- `https://developer.mozilla.org/en-US/docs/Mozilla/Integrated_authentication` (accessed 19/10/2020)
  Mozilla Integrated authentication documentation (a useful overall description of web application Kerberos authentication)

- `https://github.com/gssapi/mod_auth_gssapi` (accessed 19/10/2020)
  `mod_auth_gssapi` Apache module documentation

- `https://pypi.org/project/django-gssapi/` (accessed 19/10/2020)
  `django-gssapi` documentation

- `https://www.postgresql.org/docs/12/gssapi-auth.html` (accessed 19/10/2020)
  Using PostgreSQL with GSSAPI

- `https://web.mit.edu/kerberos/krb5-1.12/doc/` (accessed 19/10/2020)
  MIT Kerberos documentation

- `https://owasp.org/www-project-top-ten/` (accessed 21/10/2020)
  OWASP *Top 10 Web Application Security Risks* list