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

Daniel Carter

May 5, 2021

Chapter 1

Introduction

1.1 Web application security

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. In general, 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.

In this scenario, the application framework has the ability to read and write arbitrary data in the database, and the only thing which prevents 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 accidentally cause data to be visible to the wrong users. 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 [1] 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, with insufficient checking of parameters, a user can supply a string such as
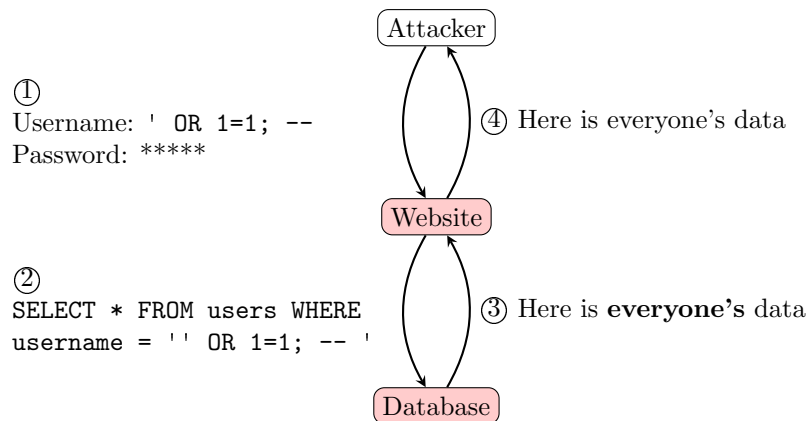
```
' OR 1=1; --
```

resulting in a total query of

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

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

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

In the diagram above, **red nodes** represent the Trusted Computing Base (TCB) of the system (i.e. 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 in it could allow an attacker to take over the system.

In step 2, it is worth noting 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

There are many variations on master password leakage depending on exactly what was leaked, but these usually 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 which it uses to authenticate 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, they are able to abuse an application which holds a copy of that master password to perform malicious acts.

To demonstrate how significant a problem this is, a number of companies (such as GitGuardian [2]) have been set up 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 access to the database due to the application's structure, as detailed below). Although the key was moved to a file outside the repository (and changed, so that 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 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 service which has genuinely leaked a secret key must quickly change this key everywhere it is 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.

1.1.3 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 a fine of up to 2% of a company's global turnover (or 10 million Euros, if that is greater) may apply in cases of unauthorised disclosure of personal data [3]. 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.

1.2 Website Sign-on Systems

A further disadvantage of having a "local" system of checking passwords on each website is that users are then expected to memorise a large number of different passwords, since otherwise a compromise of the password database of 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* system can offer significant benefits.

A single sign-on (SSO) system generally consists of an authentication server, which collects a password (or some other security token) from the user and verifies 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, along 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** (i.e. 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 has the ability to do a form of "delegation" to allow one server to request resources from another, on behalf of the user [4]).

1.3 Project Summary

This project aims to produce 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 – all they need to do is to visit the website and the authentication process will be carried out automatically (with the minor caveat that Kerberos authentication must be specifically enabled in most web browsers, 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 acheive this was already existent in some form, the actual completion of the project involved evaluating possible solutions and technologies for each part of the project, and adding functionality to libraries where this was required. The project also involved building the demonstration web application itself in order to provide a simple setup interface and demonstrate the correct working of the system.

Although involving the development of a web application and associated software patches, the main purpose of this project 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 a number of different possible approaches for development before selection of a suitable technology, and in two cases patches to existing open-source projects were developed to add functionality that they did not previously offer.

1.4   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 [5] describes the *Simple and Protected GSS-API Negotiation Mechanism* (SPNEGO) 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 complete the authentication process 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 referred to in this project.

- RFC 2743 [6] (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 delegatable 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 then passes it to the appropriate service without the user attempting to extract the contents of the ticket. RFC 2744 [7] lays out a concrete API for RFC 2743 in the form of C language bindings.

- RFC 4178 [8] describes, at a conceptual level, a negotiation framework for GSS-API systems to select an authentication system which they are both compatible with. Under this setup, the system initiating a connection offers a set of mechanisms which it can use to connect, with an order of preference. The system being connected to can then use the highest-preference system which it is capable of using, or refuse the connection if none of the suggested mechanisms are suitable.

- A later RFC (RFC 4559 [9]) 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 by virtue of 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 above in the aims for this project.

The "HTTP Negotiate" extension was subsequently included in various open-source web browsers. It also has some support in the Apache web server, and the aim of this project 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 which supports the Negotiate extension.

Chapter 2

Preparation

2.1 Kerberos

The Kerberos protocol works based on a system of *tickets*, which are managed by a *Key Distribution Centre* (KDC). The basic requirement of the system is to allow a centralised database of users (for example, the main user database in an office) who can demonstrate their identity in order to log 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 implementation of the Kerberos protocol is found in Microsoft's Active Directory system, which is used by many organisations to centrally co-ordinate logins and access to resources 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.

In both Microsoft-based and open-source implementations, LDAP (Lightweight Directory Access Protocol) databases are commonly used to hold user information. At the most basic level, these hold a set of users and corresponding *attributes* (which could include information about how the user is allowed to log in, which resources that user is allowed 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 TODO, 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 reqesting a Kerberos ticket from the KDC, authenticating using their password.

- The KDC returns 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.

- When the user wants to access a service, they send the TGT back to the KDC along with an identifier for the service which they want to use.

- The TGT grants the user a *service ticket*, which the user then passes on to the service. The service is then able to use that ticket for authentication.

The following listing shows a client which has obtained 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 (`HTTP/krbsite.local@LOCAL`) is a *service ticket* 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 command 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 a ticket cannot be renewed after it has expired.

2.2 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.
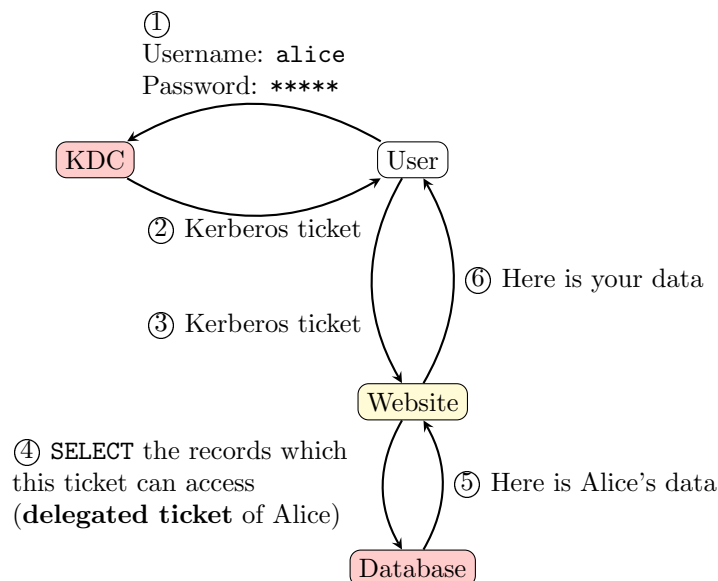
This is valuable because it means that X can access resources which "belong" to A, without X needing to have direct access to Y. This means that less trust in X is needed (since it does not need privileged access to Y), and so the types of attacks discussed above become much less likely.

2.3 Project Structure

2.3.1 Overall Structure

The overall structure of the authentication process is as shown below.

Since the same KDC can be used for many websites where the users all have a login to the same Kerberos realm, the system also functions as a single sign-on system (using the Kerberos ticket as proof of identity). As detailed in sections 2 and 3, this means that none of the individual websites need to be given access to the user's password, and that that they are not able to arbitrarily impersonate the user on other systems. The possibility of delegating tickets in a controlled manner means that the website shown can fetch data from the database on behalf of the user, but it cannot get any data from other systems which it has not been authorised to access. It 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).

① Username: alice
Password: *****

KDC

User

② Kerberos ticket

③ Kerberos ticket

⑥ Here is your data

Website

④ SELECT the records which this ticket can access (**delegated ticket** of Alice)

⑤ Here is Alice's data

Database

Note that, unlike in the diagram shown in Section 1.1.1, the web application no longer has to be part of the trusted computing base; it does 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 the previous diagram cannot take place.

2.3.2 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 makes use of a structure based on *models* which correspond roughly to classes in the object-oriented programming model (and are represented by Python classes in the Django configuration), with each model being able to have different attributes (which are set up as member variables of the class). When stored in a database, each model is a table, with each object of a model being a row in that table and each attribute being 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. The database model can be represented in the following entity-relationship diagram:
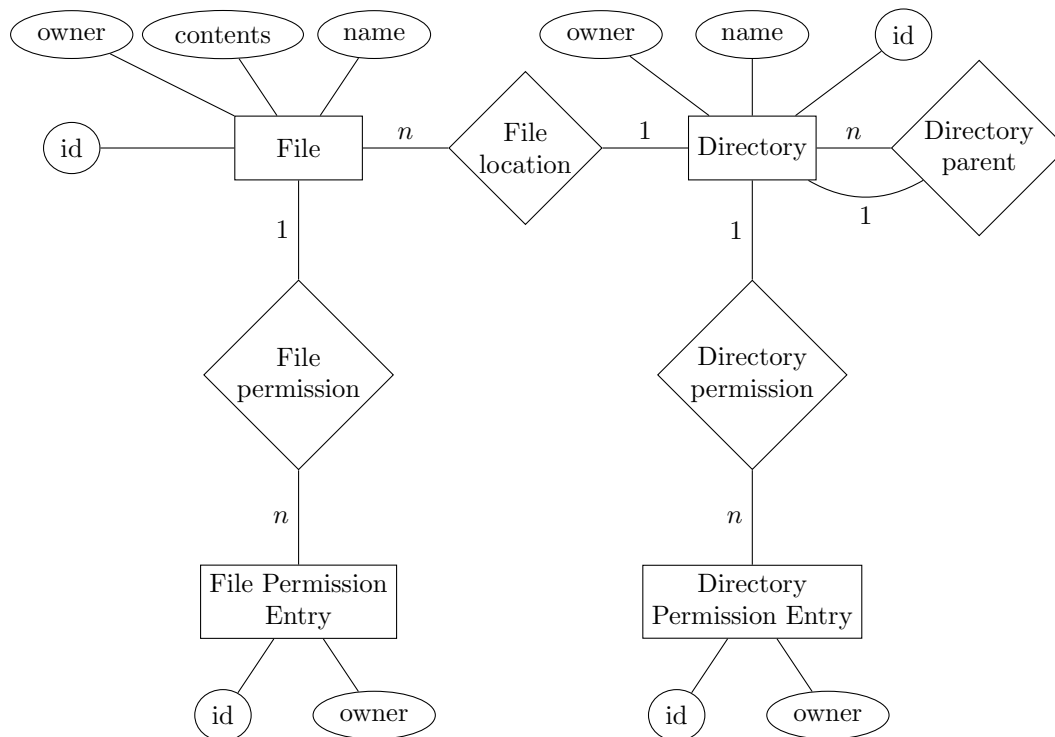
Database Setup Application

It 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 the 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 appropriate access controls

2.4 Requirements Analysis

There are three broad components to the setup of this project: 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 any previous experience, I decided to continue using it following some experimentation over the summer (see "starting point" below). 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 a number of 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 of use here.

2.4.1 Source Code Management

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

Since the main development was completed in virtual machines which would have been time-consuming to set up again 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 on them.

2.4.2 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. As a naïve implementation of the project also has the potential to result in slower performance than existing systems (due to extra authentication overheads), evaluation of performance and comparing it with a non-Kerberos based system is also a component of evaluating the system.

2.5 Starting Point

(N.B. the text below is a slightly modified version of the starting point submitted 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 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 and 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 have investigated these and built 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 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. 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.

In addition, over the summer vacation I did 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).

The Django documentation [10] was used as a reference and guide throughout development, including using (some of) the provided tutorial to set up the basic web app before being integrated with the Kerberos authentication setup.
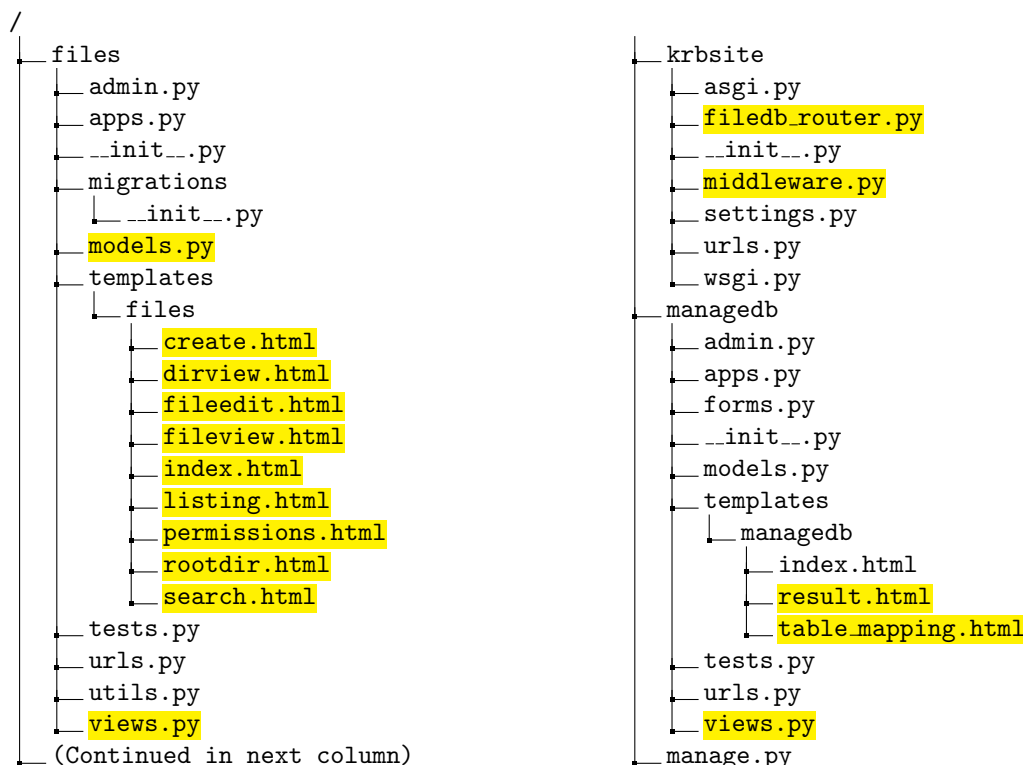
Chapter 3

Implementation

3.1 System Structure and Repository Overview

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

The structure of the directories involved is as shown below. The basic directory layout is as produced by the Django project setup procedure, with some files having "stubs" automatically generated by this process (as well as a default `settings.py` and the short `manage.py` script which is provided to run various application management steps).

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

The top-level directories represent the separation of "compontents" 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 are used to define web forms to gather user input (some of which are very simple text-gathering fields, but others of which require complex validation of user input, such as the `TableMappingForm` in `managedb/forms.py`.

`files/utils.py` contains two small helper functions which are called by `files/views.py`.

The `models.py` in the `files` app defines the object model used by the file browser application, in the same way as would be done for any other Django app. This includes both the actual data storage tables `Directory` and `File`, as well as two permission tables (for directories and files respectively) that are defined here so that 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 anything (beyond the two-line template file provided when a Django app is created) since the database mangement 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 is generally designed for testing application-specific features (e.g. whether the object model has been correctly defined) and 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 such as the following (that are filled in by the rendering engine):

```
{% 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 is used to populate one form field based on the value of another in `table_mapping.html` (under `managedb/templates`).

In keeping with the principle that the web app should simply be a "view" onto the main database (and also 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, but not 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 the setup did require a Django `AuthRouter` to make sure that queries are always directed to the appropriate database: see Appendix 1 for more details and code.

3.2   SPNEGO

SPNEGO (Simple and Protected GSSAPI Negotiation Mechanism) is a mechanism which uses a single packet exchange to itentify 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 involves using the hash of a user's password as a long-term key, which means 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 relatively short period of time).

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

```
GssapiAllowedMech krb5
```

A full copy of the `.htaccess` file which configures many options in Apache for this project, is included in Appendix 2.

3.3   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 (i.e. the parts of the system which, if compromised,

could allow the security of the whole system to be compromised). Although not actually requiring any new software development (since in a real situation this backend would already exist, for instance as an Active Directory server), arranging a suitable test setup with the right permissions was required for the project.

3.3.1 Basic MIT Kerberos Backend

The default backend used by MIT Kerberos is based on Berkeley DB and simply creates a set of 4 files in which 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 because it is missing the required access control functionality [13]. Because this was a goal of this project, the Berkeley DB-based system was not suitable.

3.3.2 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 to define what permissions a given principal has for delegation. This is explored further in section TODO.

3.4 Kerberos Ticket Delegation

Another major aspect of this project is how a ticket can be *delegated* from one server to another, such that the ticket used to log into the web app can be used to authenticate to another system. Kerberos supports several methods of delegation, and I investigated these when researching the best method 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 is able to use those tickets.

3.4.1 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" [11].

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 (and not send any more, although if A sent a ticket that was renewable it may be necessary to wait for the "renew until" time has passed to be sure that it cannot be used). This contrasts with passwords which 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 sent in this way can be bound to IP addresses, to limit the risk that they are passed on to insecure other systems.

While the second of these advantages is significant, it may still not offer enough access control over the network. In particular, A cannot give X the ability to delegate to Y without also giving X the ability 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 retreive 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.

3.4.2 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, without having any need to know the user's Kerberos password [15].

Unlike with unconstrained delegation, here the ticket-granting ticket is not passed over to X; instead, X is given a service ticket from A as normal (to prove A's identity and that A wishes to connect to X). 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 is able to give X a ticket to Y on behalf of A.

This requires a more complex setup of permission models in the KDC than simply a list of principals, since the KDC must now decide which services can delegate tickets to which others. The default back-end for MIT Kerberos does not support this, but moving the user data into an LDAP server and using this as the data source allows the relevant `krbAllowedToDelegateTo` permission to be set on the user [13].

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

- Much richer customization of access control is possible. The system can now be set up so that A can 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 customizable set of permissions between systems, and 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 is allowed to. The attacker does not get a ticket-granting ticket to use on arbitrary applications.

In this project, the permitted delegation patterns are described using records in the LDAP database. The following pattern is therefore used to allow the web server on `krbsite.local` to delegate to PostgreSQL on `krb.local` (i.e. set the `krbAllowedToDelegateTo` attribte on `HTTP/krbsite.local` to `postgres/krb.local`):

```
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 very similar setting named `msDS-AllowedToDelegateTo` is available, and this can be used in effectively the same way as above to enable constrained delegation from one service to another [16].

3.4.3   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 [15] 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 is able to 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 delegatable ticket using S4U2self can be controlled in the KDC, and so (for example) the following command is used to stop the web server from being able to to so:

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

3.5   Credential Cacheing 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 of these 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, this approach 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 now becomes at least two requests. This potentially doubles the access time to the website, as well as consuming extra network bandwidth and placing additional load on the web server. Given that individual web pages often consist of a number of 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]) and having to include the full ticket in each request may take up significant additional bandwidth. This may cause problems if users are on slow or quota-controlled network connections, and further increases the demand on the server.

See section TODO in the evaluation for more discussion of the problems caused by using 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 each request could take place as normal (with the client providing the cookie and the server extracting the ticket from the cookie to use for authentication) and so solve the first of the issues above, but still result in (potentially) large cookie files being sent around and adding a lot of overhead to every request.

The solution generally adopted here is to cache the tickets on server X and send a (short) session cookie to A so that 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 connects to the server on subsequent occasions.

Ignoring two Windows-specific mechanisms, MIT Kerberos offers five methods of storing credential caches [17]:

---

[1] As a (somehwat simplistic) illustration of this, obtaining a Kerberos ticket from my test application (running MIT Kerberos) produced a credential cache of size 650 bytes. Obtaining a ticket from the University's Active Directory server `BLUE.CAM.AC.UK` resulted in a file size of 1536 bytes, or over double the size of the MIT Kerberos ticket.

- **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 below).

- **DIR** (directory) mode places all tickets in the supplied credential cache into a particular directory. In this case (since A only needs to supply one ticket to X) this has 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.

  However, the process model used by the Apache web server introduces some difficulties in this situation. 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 the previous requests using that thread. Since the thread and process IDs will remain the same for future requests, this is difficult to acheive.

  Further, the `mod_auth_gssapi` module is currently only designed to work with file-based caches (the `mag_store_deleg_creds` function assumes that any input it is given is a filename and prepends `FILE:` to the input, 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 file-based approach (with some additional protections to keep tickets private, as detailed below).

- **MEMORY** caches, which are conceptually similar to keyring-based caches except they are not Linux-specific and offer less granularity in setting access permissions. Many of the same issues described above also apply to this cache type, particularly in relation to removing access from threads.

- **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 would normally be impractical, this limits the usefulness of this option.

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 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, and execute (`x`) permissions allow a user to "traverse" the directory and read a given file whose name is known, granting a user execute but not read permissions allows the filename to (in effect) become a password that can be used to access the file. Since the permission to list files in the directory is not provided, 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.

3.5.1 `mod_auth_gssapi` Patch

In order to make use of this, I added an option to `mod_auth_gssapi` (`GssapiDelegCcacheRandom`) 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 if the filename is known, but not read the list of files in it), this means that 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 [18]. A new library to include the `BASE64URL` encoding method was recently added to Apache [19], but was not yet available in the packaged Apache version I was using and would have reduced the utility of the patch if it resulted in `mod_auth_gssapi` now requiring a more recent set of libraries to work. Since an actual standardised encoding of information is not required (just a random sequence of characters 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 3.

3.6   SQL Access Control

A core aspect of this project is the ability to shift 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 a number of ways of achieving this (especially since they are not very well standardised across SQL implementations). As this project specifically uses PostgreSQL, I evaluated a number of supported methods to determine how best to implement the security protocol.

3.6.1   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, if there is a table of "sensitive" information that only a certain group of employees are allowed to see, using simple SQL `GRANT` statements allows this to be set up without any further complexities. It is only in the case that users need access to specific rows within a table that more complex access control methods need considering.

3.6.2   SECURITY DEFINER functions

As with most SQL implementations, PostgreSQL allows the definition of 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 users of the database.

Basic stored procedures do not have any specific security implications; they are merely a way of avoiding repeatedly executing the same code (and potentially offering performance improvements). However,

PostgreSQL defines an additional construct that permits 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 [20] (in a similar way to how the `setuid` bit works 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.

This approach requires some caution since `SECURITY DEFINER` also redefines `current_user` (a variable which normally contains the username of the user executing a query) to the user who defined the function, although this can be worked around using `session_user` instead. Once this is taken into account, such a procedure can be used to return files where the current user has an appropriate entry in a permission table.

These are effectively 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 therefore allows something of a separation of authentication (logging into the database, via Kerberos) and authorisation (where the procedures describe what access is permitted).

While very flexible, this approach is is not easily integratable into frameworks such as Django, which are designed to execute queries on tables rather than (effectively) making function calls to an API. In order to integrate this approach properly into Django, I would have needed to replace for the current object-relationship mapper with one that supported using procedures rather than simple `SELECT` statements, or otherwise resort to the web app designer having 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 which works using existing models, this setup is non-ideal.

3.6.3   SQL views

These effectively allow a new table to be created out of a view over an existing table. Using syntax such as the following, it is therefore possible to construct a new table which only has the calling user's data visible; the user can then be granted access to this view (which automatically selects data based on the user's own username) and does not then need to have 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 cannot be directly granted in this way (except for simple views, which this is not considered to be because it selects data across multiple tables. This can be achieved, either by using `ON INSERT` rules in the same way as the `ON SELECT` rules below, or by using SQL triggers [21].

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 to have 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 in the amount of duplication, but this is still non-ideal. For example, the following code 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, rather than simply a view, 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'
```

3.6.4   Use of `ON SELECT` to redefine selection

PostgreSQL also allows rules to be created which effectively redefine actions on tables. For example, a rule could be used to 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" [22].

3.6.5   Row-level security

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

There are two forms of policy, which are applied in different cicumstances [23]:

- `USING` policies, which 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 which they have permission records for:

  ```
  # Enable row-level security on the relevant table
  cursor.execute("""ALTER TABLE %s ENABLE ROW LEVEL SECURITY"""
          % (source_table))
  # Only allow users to view and edit files which they have permission to see
  cursor.execute("""CREATE POLICY %s_view ON %s
  USING (%s IN (SELECT %s FROM %s WHERE %s = session_user))"""
          % (source_table, source_table, source_column, perm_column, perm_table,
              owner_column))
  ```

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

- `WITH CHECK` policies, which 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 insert permissions to allow them to read arbitrary files, this is another critical part of the security model:

  ```
  # Also enable row-level security on the permissions table
  cursor.execute("""ALTER TABLE %s ENABLE ROW LEVEL SECURITY"""
          % (perm_table))
  # Only allow users to set permissions where they already have access to that file
  cursor.execute("""CREATE POLICY %s_view ON %s FOR INSERT
  WITH CHECK (%s IN (SELECT %s FROM %s WHERE %s = session_user))"""
          % (perm_table, perm_table, perm_column, perm_column, perm_table,
              owner_column))
  ```

A further special-case policy allows newly created files (with no current owner) to have permissions assigned. With this type of policy, an error is generated if a user tries to create entries without the authority to do so.

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 and so it is impossible to remove all SQL injection risks if the application is compromised **while** the set-up phase is going on.

The row-level security model runs in parallel with the standard SQL permissions model and both systems must allow access for an action to be allowed. 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.

3.7   The database setup application

As the most significant component of software produced for this project, the database setup application is designed to allow a website administrator to easily set up SQL permissions such that other Django applications (including the file browser example app) work with minimal changes.

This application first needs to gather information from the user to allow a "matching" of data and permission tables (i.e. to determine which table holds permission records for a given table of data). This is clearly dependent 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.

Therefore, executing the following query on a given model (using the `model_data` paramaters 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 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])
```

3.8   Kerberos access to PostgreSQL

Although PostgreSQL includes Kerberos-based authentication via the GSSAPI framework [24], 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 in the case of an application where many user's tickets will be stored and 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 therefore 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, an additional field has also been added 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 Kerberos support will be included in the build or not.

Note that the public API does not need to 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 in this case it is simply ignored and PostgreSQL works as normal (because of the null check around the `gss_krb5_ccache_name` call shown above).

Chapter 4

Evaluation

4.1 Success Criteria

The two success criteria for the functionality of the application (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 (which was 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.

4.2 The Django Application

4.2.1 Database Setup

The database setup application first presents the user with a welcome screen, displaying the database name which it will operate 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.

Following this, the next stage in the process fetches data on the database tables from the Django configuration. It then queries the database itself for information on relationship models (see implementation section for more details) and presents the user with a setup page to choose 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:

Directory

Include: ☑
Owner field: owner ∨
Table: files_directory_permission ∨
Column: owner ∨

File

Include: ☑
Owner field: owner ∨
Table: files_permission ∨
Column: owner ∨

Submit

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

Note the access control to this page is not critical (although it may well be done in practice to avoid confusing users). If an unauthorised user attempts to access this page, they will likely see no tables listed here (as the warning at the bottom of the previous image describes) due to not having access privileges to PostgreSQL's `information_schema` tables that hold metadata about the databases. 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.

Finally, a "completion" screen is shown. Following a successful operation this is blank, and error messages are reproduced here if the database engine returned an error:

Set up database: Complete

The database setup has been completed.

4.2.2 Demonstration Application (File Browser)

As a practical illustration of how the technology works, I developed a file browser application which allows 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 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

Term: [   ]

[Search]

Each file and directory has a permissions interface visible to the registered "owner" of that record, to allow that user to add or remove permission entries:

user1 directory (directory)

This directory is owned by user1

Permissions

Select a user to **remove** their access permissions

Remove:

- ☐ user2

Grant access to: [   ]

[Update]

Back to file browser

In addition, file contents can easily be updated by the file owner (other users who have permission to access the file have read-only access – this could easily be augmented to allow a more complex access control mechanism with separate read/write permissions that could be assigned to users):

Editing Test file

```
This is a file I just created
```

Contents:

[Update]

Cancel edit

Back to file browser

The search feature of the app helps to 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 a string provided by the user):

```
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 the introduction), and the web app's response shows that it is indeed vulnerable to SQL injection:

Search results for ' OR 1=1;--

['Test file']

Although this is clearly non-ideal, it does not disclose any information which `user1` cannot access anyway. Note that 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(*) AS TOTAL FROM files_file GROUP BY owner;
 owner  | total
--------+-------
 django |     6
 ***    |     2
 user1  |     1
(3 rows)
```

(Note that the entry marked `***` was changed from the real command output to remove a personal identifier, but otherwise the actual output is identical to what is shown here. Note also that, for simplicity, `user1` has not been given access to any non-owned files.)

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

4.3   Security Analysis

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. Doing this for the database after running the setup application produces the following result:

(N.B. the "Schema" and "Column privileges" columns do not show any useful information here and have been omitted from this document for space reaspons)

```
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): (id IN ( SELECT files_directory_permission.directory_id    +
                                   |          |                      |          FROM files_directory_permission                          +
                                   |          |                      |         WHERE ((files_directory_permission.owner)::text = SESSION_USER)))+
                                   |          |                      | 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_view (r):                              +
                                   |          | =ard/django          |   (u): true                                                       +
                                   |          |                      | 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_delete (d):                            +
                                   |          |                      |   (u): (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): (id IN ( SELECT files_permission.file_id                   +
                                   |          |                      |          FROM files_permission                                    +
                                   |          |                      |         WHERE ((files_permission.owner)::text = SESSION_USER)))   +
```

```
                                  |          |                        | 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_view (r):                                +
                                  |          | =ard/django            |   (u): true                                               +
                                  |          |                        | files_permission_insert (a):                              +
                                  |          |                        |   (c): (file_id IN ( SELECT files_file.id                 +
                                  |          |                        |    FROM files_file                                        +
                                  |          |                        |   WHERE ((files_file.owner)::text = SESSION_USER)))       +
                                  |          |                        | files_permission_delete (d):                              +
                                  |          |                        |   (u): (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 (and corresponding to the SQL queries made in the application code).

Chapter 5

Conclusion

Bibliography

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

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

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

[4] 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)

[5] 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)

[6] 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)

[7] 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)

[8] 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)

[9] 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)

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

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

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

[13] Kerberos: delegation and s4u2proxy (Simo Sorce, 12/02/2012) `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] 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)

[16] 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)

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

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

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

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

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

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

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

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

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

Appendix 1: Django database router

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

```python
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
        """
        if (app_label == self.APP_LABEL) and (db == self.DATA_DB):
            return True
        if  (app_label != self.APP_LABEL) and (db != self.DATA_DB):
            return True
        return False
```

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 2: 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 3: `mod_auth_gssapi` Patch

The following `diff` extracts show some 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 [25])

```
 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;
 }
```