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

Daniel Carter

April 25, 2021

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 load it for display to the user. 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 which they are not authorised to read 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

This is a form of command injection attack (which are currently number 1 in the OWASP *Top 10 Web Application Security Risks* list [1]). This 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

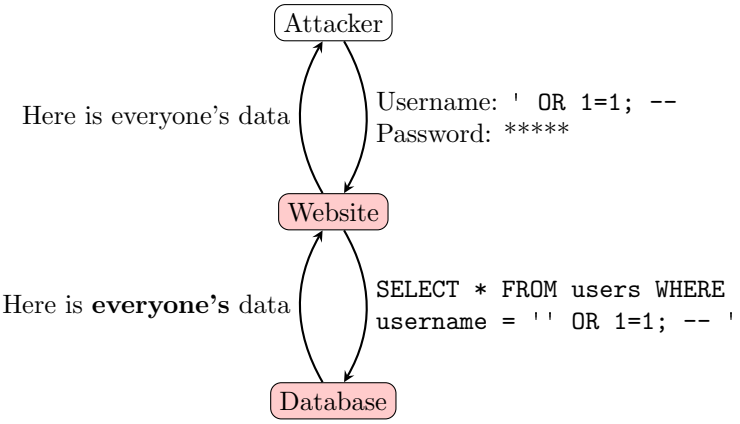
```
' OR 1=1; --
```

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:



1.1.2 Master password leakage

There are many variations on this 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.

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:

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

Protect Your GitHub Repos

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 which uses that 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 having 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, the project includes 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.

1.4 Related Work

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.

A later RFC (RFC 4559 [6]) 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 using an open-source web framework (Django) in conjunction with the Apache web server and any client web browser which supports the Negotiate extension.

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 login directory 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.

The basic workings of the 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).

```
daniel@Daniel-Laptop:~$ klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: dcc@LOCAL

Valid starting     Expires            Service principal
02/04/21 15:36:01  03/04/21 01:36:01  krbtgt/LOCAL@LOCAL
renew until 03/04/21 15:36:01
02/04/21 15:36:07  03/04/21 01:36:01  HTTP/krbsite.local@LOCAL
renew until 03/04/21 15:36:01
```

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

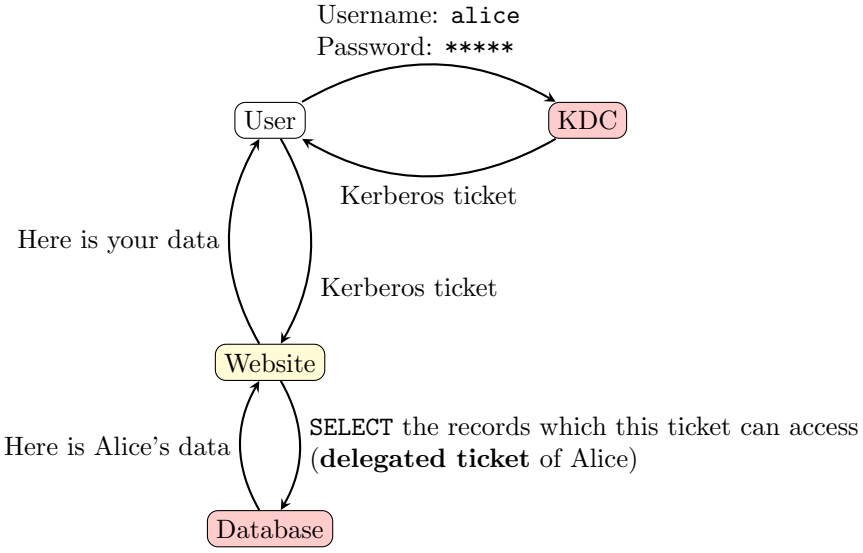
This project demonstrates a web application that uses Kerberos authentication and delegation to fetch data from a database and display it to the user, in the form of a basic file browser-type application. It also includes a component which performs the necessary setup tasks (setting up SQL permissions), so that all that is required for the user of the application is to:

- Set up a Django app with the appropriate models (in the example web app, the models are files and directories, plus associated tables to hold permissions)

- 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

The overall structure of the authentication process is as shown below, noting that the application itself does not have any other access means of accessing the database, so an attack such as the one depicted above cannot take place.

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 does **not** mean that any of the individual websites need to be given access to the user's password, or that they are 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 not from other systems which it has not been authorised to access.



3 Implementation

3.1 System Structure

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.

TODO: Overview of repository structure

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 challenge-response protocol, whose cryptography can be easily broken using current technology and so which is vulnerable to attack) or Kerberos, and 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
```

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

3.3.2 Connecting to an LDAP Server

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.

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.

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" [7].

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). This contrasts with passwords which are valid indefinitely and may well not be straightforward to change in a complex corporate network.
- 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.

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 an extension to the basic Kerberos setup that allows one service X to obtain a 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 [8].

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 [9].

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

- Access control can be almost completely customized. 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.
- 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.

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 [8] 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.

3.5 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 achieveing 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.

3.5.1 `SECURITY DEFINER` functions

A PostgreSQL function with `SECURITY DEFINER` in the signature runs "as" the user creating the function, regardless of who executes it [10] (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 the `SECURITY DEFINER` property also redefines the `current_user` 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.

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. 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.

3.5.2 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 files visible; the user can then be granted access to this view (which automatically selects files based on their 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;
```

While less significant than for `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.

3.5.3 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" [11].

3.5.4 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 [12]:

- `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.

3.6 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])
```

3.7 Kerberos access to PostgreSQL

Although PostgreSQL includes Kerberos-based authentication via the GSSAPI framework [13], 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.

4 Evaluation

5 Conclusion

Bibliography

References

[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] 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)

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

[8] 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)

[9] Kerberos: delegation and s4u2proxy (Simo Sorce, 12/02/2012) https://ssimo.org/blog/id_011.html (accessed 02/02/2021)

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

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

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

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

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:

```
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/`.)