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

Daniel Carter

February 22, 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:

- **SQL injection**, 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

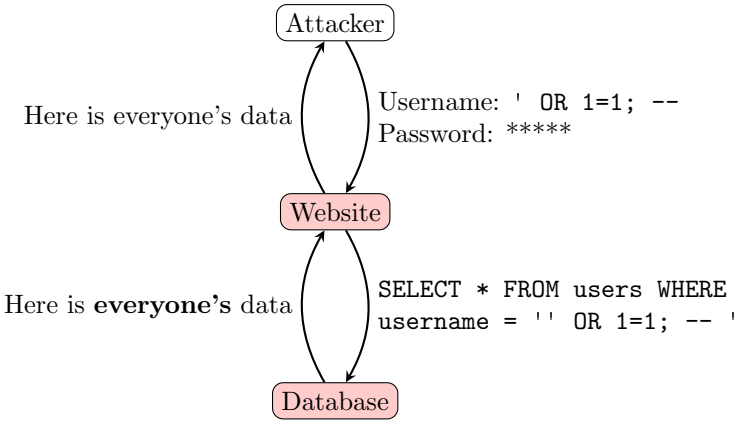
  ```
  SELECT * FROM records WHERE username='' OR 1=1; --';
  ```

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

- **Master password leakage**, which enables any 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.

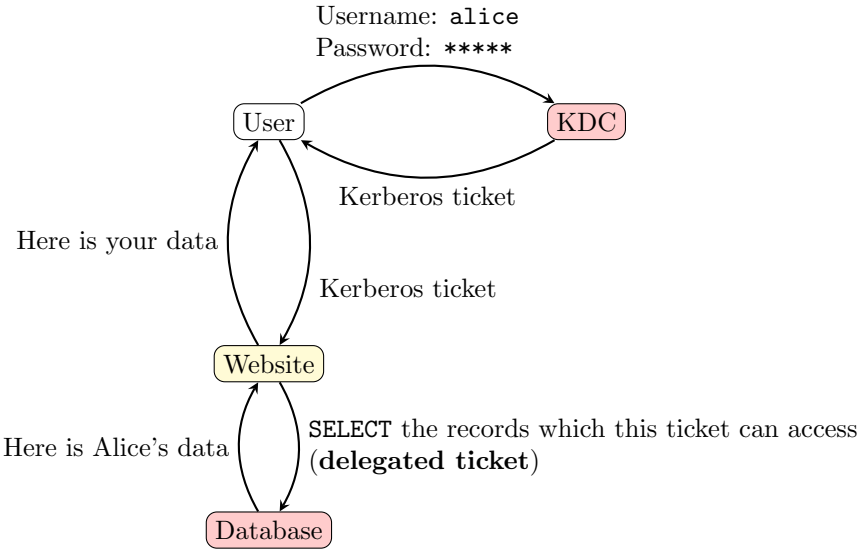
As well as being commercially and reputationally damaging, unauthorised access to data can result in severe legal penalties: the Data Protection Act 2018 specifies a fine of up to 2% of a company's global turnover (or 10 million Euros, if that is greater) may apply in cases of unauthorised disclosure of personal data [2]. 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.

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:



1.2 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. 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.



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.
- (Extend to cover ticket delegation here)

3 Implementation

3.1 Implementing SQL Procedures

As one aspect of the project, the database server must be able to provide the correct set of records to each user without exposing the rest of the table. Since PostgreSQL does not offer the ability to selectively allow a user to query certain rows in a table, this can instead be acheived using stored procedures. However, these are not a standardised construct in SQL, and even within PostgreSQL there are multiple ways to achieve this.

3.1.1 `SECURITY DEFINER` functions

A PostgreSQL function with `SECURITY DEFINER` in the signature runs "as" the user creating the function, regardless of who executes it [3] (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. It is also not as easily integratable into frameworks such as Django, which are designed to execute queries on tables rather than (effectively) making function calls to an API.

3.1.2 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" [4].

3.1.3 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 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;
```

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

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

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