Sharing Databases on the Web with Porter Proxy

Xin Wang
School of Electronics and Computer Science
University of Southampton
xwang@soton.ac.uk

Aastha Madaan
School of Electronics and Computer Science
University of Southampton
A.Madaan@soton.ac.uk

Eugene Siow
School of Electronics and Computer Science
University of Southampton
eugene.Siow@soton.ac.uk

Thanassis Tiropanis
School of Electronics and Computer Science
University of Southampton
tt2@soton.ac.uk

ABSTRACT

With large number of datasets now available through the Web, data-sharing ecosystems such as the Web Observatory have emerged. The Web Observatory provides an active decentralised ecosystem for datasets and applications based on a number Web Observatory sites, each of which can run in a different administrative domain. On a Web Observatory site users can publish and securely access datasets across domains via a harmonised API and reverse proxies for access control. However, that API provides a different interface to that of the databases on which datasets are stored and, consequently, existing applications that consume data from specific databases require major modification to be added to the Web Observatory ecosystem. In this paper we propose a lightweight architecture called Porter Proxy to address this concern. *Porter Proxy* exposes the same interfaces as databases as requested by the users while enforcing access control. Characteristics of the proposed Porter Proxy architecture are evaluated based on adversarial scenario-handling in Web Observatory eco-system.

CCS Concepts

•**Security and privacy** → **Distributed systems security;** •**Networks** → *Cloud computing;* •**Computer systems organization** → *Cloud computing;*

Keywords

security, access control, data sharing, proxy, web observatory

1. INTRODUCTION

In the past a few years an increasing number of datasets have been shared on Web Observatory. Some datasets are distributed as files and more as databases such as MySQL and MongoDB. While enables users to access (e.g. query) these databases, Web Observatory also protect them from unauthorised access. Similar to a reverse proxy, Web Observatory 1) hides the addresses and credentials of the underlying databases; 2) provides an authentication and access control layer (based on OAuth 2.0[1]) that is independent from the underlying databases; and 3) enables authorised users to query the underlying databases via an API. However, the Web Observatory API authenticates and communicates with user clients via the HTTP protocol, while many databases adopts their own protocols based on the TCP protocol. Since existing programming libraries and applications are built around those proprietary protocols, their code cannot be easily reused with the Web Observatory API.

To encourage user engagement and enable the reuse of existing application code, we need to expose the same interfaces as those of the databases shared on Web Observatory in addition to the Web Observatory API, while not to weaken the protection of the databases. A TCP reverse proxy can effectively exposes databases while hiding their addresses and credentials, but it relies on the underlying databases to manage who are authorised to access the databases. We require a reverse proxy that can mimic databases' protocols (for both authentication and query) and separate access control from the underlying databases. We propose Porter Proxy to achieve this goal.

2. RELATED WORK

The community of users engaging with the Web Observatory may share their artifacts, datasets (or datastreams) and applications in a variety of ways. For example, some of them may allow other users to download a copy of their datasets (in form of files). While another set of users may share end-points to their databases through which the users can query the given datasets. The former method though simple to share the datasets but may be expensive in terms of storage required and also not secure. On the contrary, the latter method enables users to access the latest available datasets, avoids complicated data exchanges but also raises systemic concerns for the Web Observatory infrastructure. These include, providing a uniform access interface for authentication on the Web Observatory which is compatible with the underlying databases.

Addressing this issue is important as, in contrast to limited number of databases available, now a data is available through an increasing number of databases. These

[1]https://tools.ietf.org/html/rfc6749

©2017 International World Wide Web Conference Committee (IW3C2), published under Creative Commons CC BY 4.0 License.

ACM 978-1-4503-4914-7/17/04.
http://dx.doi.org/10.1145/***Add your assigned DOI**

databases have their own authentication and access protocols and can make the re-use of database complex. Moreover, it may not be necessary that all of these datasets support public access. There are very limited, almost no studies in the existing literature that directly address these concerns. However, draw our related work on web observatories, secure and trusted data sharing, lightweight access control for mobile and cloud environments and we consider the studies done in the area of heterogeneous authentication methods.

Web Observatory is a distributed infrastructure capable of harvesting, querying, and analysing multiple real-time and historic heterogeneous data, whilst providing data owners access control to their resources [1] [2], [3]. A building pillar of the Web Observatory is that access to some datasets can be restricted for licensing, privacy or other reasons. The Web Observatory allows its users to list or host datasets that are public or private. Access to private datasets is managed by the user who hosts them on the Web Observatory [2].

Datasets on the Web Observatory are in various types of stores including, SQL, NoSQL, and triple (RDF) stores, non-structured formats such as CSV [2]. Resources can be queried using the **Web Observatory API**, which uses a JSON structured query language, and the mappings to the various types of data-stores is handled by the Web Observatory API, and processed server-side. Unlike a distributed database architecture which can be considered as a single logical database, we wish to provide a middle layer which provide a NoSQL like query syntax that uses the Web Observatory API to query multiple data-stores solutions [2]. This API acts as a secure middle layer between the dataset locations, and the end-user connections, whether this be via direct access on the Web Observatory portal, or via programmatic use via the development of third party applications and visualisations [2], [4].

3. ARCHITECTURE OF PORTER PROXY

Porter Proxy adopts a lightweight architecture to separate credential management from databases and at the same time exposes the same interface as the databases it represents, which allows the reuse of existing application code. It manipulates connections between sockets of clients and databases at different stages to achieve its goal. As shown in Figure 1, for a protected database Porter Proxy runs an authentication database of the same type, and an authentication client that can communicate with the protected database. Each client is registered as a user of the authentication database and managed by Porter Proxy. Credentials to access the protected database are securely kept by Porter Proxy and only used by the authentication client. Modifications made to credentials at the authentication database (i.e. client credentials) do not affect credentials of the protected database, and vice versa. When a client initiates a request to access the protected database, it authenticates against the authentication database. At the same time the authentication client authenticates against the protected database to establish a trusted connection. If both pass the authentication respectively, the client socket is detached from the authentication database and connected to the socket of the protected database (as shown in the right half of Figure 1). After the sockets switch, TCP packets representing requests from the client are redirected to the protected database. This redirection is transparent to both the client and the protected database, and can be easily integrated with intermediate functionalities that work with the TCP protocol, such as IP address filtering and load balancing.

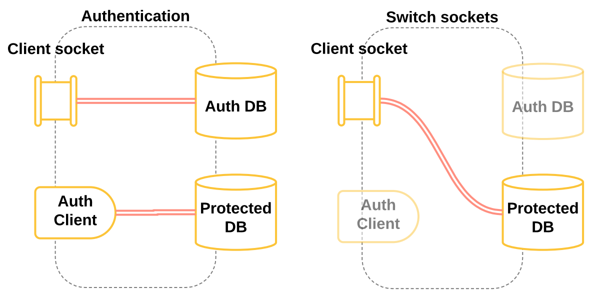

Figure 1: Porter Proxy architecture.

4. IMPLEMENTATION

A prototype of Porter Proxy is implemented in Node.js, which has native support for the TCP protocol and the potential to build resilient applications. In the prototype we take MongoDB as an example, but all other databases can be supported alike.

This implementation has the following components:

- A database management module that registers databases and randomly assigns a unique id to each database. Users browse and request access to databases using these ids to prevent leaking identities of the underlying databases.

- A user management module that provides an interface for user to register, authenticate and manage their access to databases. In addition, users can manage all the databases to which they have access. When a user (e.g. *Bob*) is granted access to a database, this module creates a new user (e.g. *Bob:passBob*) in the Auth DB and generates a databases address based on the database id, user profile and a random string. The generated address is unique for each user-database combination, and it is independent from the true address of the underlying database. Both the newly created user credentials and the generated address are shown to the user, as shown in Figure 2.

- The Auth DB, that stores user credentials and authenticates user connections. It is worth noticing that the Auth DB only cares about whether the credentials used to establish a connection are valid, but does not know which database the user connects to. It is not necessary to use a complete MongoDB as the Auth DB since only the authentication function is required. In this prototype we implement the MongoDB authentication protocol as a standalone component and use it as the Auth DB to reduce resource usage.

- A MongoDB client as the Auth Clien, that holds the real addresses and credentials of underlying databases and connects them. This client is implemented using the native MongoDB Node.js driver[2].

[2]https://mongodb.github.io/node-mongodb-native

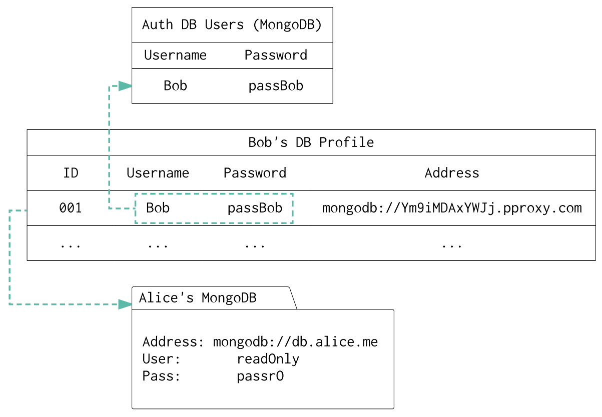

Figure 2: An example user profile demonstrating the unique database address and credentials generated for each user-database combination. In the profile the *ID* column refers to Alice's MongoDB; Entries under *Username* and *Password* are registered in the Auth DB.

- A connection coordination module that manages 1) the connection and authentication between a user client and the Auth DB; 2) the connection and authentication between the Auth Client and the underlying database, and 3) reconnects the user client to the underlying database after authentication. The coordination module resolves an alias database address to retrieve the id of the underlying database, and then resolves the database id to its real address and credentials. It is the only module that know which user connects to which database.

One challenge we encounter is at the implementation of the MongoDB authentication protocol. MongoDB supports several authentication mechanisms, and SCRAM-SHA-1[3] becomes the default one after MongoDB 3.0. In a SCRAM-SHA-1 conversation the client and the server exchange four messages (two round trips) that involve several hashing and verification. MongoDB serialises the SCRAM messages into binary data and transports them along with some metadata (such as a conversation id) over the TCP protocol[4]. Since every message is transported as several TCP data chunks, we need to trace the start and end of each message and reconstruct it from data chunks. The first 4 bytes of a message gives the total length of the message in bytes. We implement a buffer to store all received data chunks. In the beginning we wait until there are 4 bytes data to calculate the total length of the message. Once there are enough chunks (the size of all chunks is equal or greater than the total length of the message) we recover the message and remove consumed data from the buffer (it could be that a data chunk is partially consumed and only the consumed part is removed). By repeating the above procedure we manage to recover SCRAM conversations from TCP data stream.

[3]https://docs.mongodb.com/v3.0/core/security-scram-sha-1

[4]https://github.com/mongodb/specifications/blob/master/source/auth/auth.rst#scram-sha-1

5. USE CASES OF PORTER PROXY

5.1 Sharing a MongoDB with Porter Proxy

Alice wants to share a large volume of social network data in a MongoDB so colleagues can query her data. Alice follows two steps: 1) she creates an read-only account (*readOnly:passrO*) in her MongoDB; 2) she provides the credential of the read-only account and her MongoDB's address to Porter Proxy. Porter Proxy sets up the Auth DB and the Auth Client internally.

Via Porter Proxy Bob requests permission to query Alice's database. Porter Proxy asks Alice for approval. Once approved, Porter Proxy creates an account (*Bob:passBob*) in the Auth DB, and gives Bob the credentials and an alias address to Alice's database (in his profile). Bob can connect to Alice's database with a MongoDB client with the information in his profile.

5.2 Updating Existing Applications

Before registered with Porter Proxy, Alice's MongoDB are used by a few internal applications (whose credentials are manually managed by Alice). Similar to Bob, owners of these applications can request access to Alice's MongoDB from Porter Proxy and find the generated credentials and alias addresses from their profile pages. By only updating the database credentials and addresses used in their application code, these applications would function as normal.

5.3 Building Applications with Porter Proxy

Alice's data (in the MongoDB) turns out to be useful not only internally but also among users outside her institute. As the MongoDB is on Porter Proxy, external users can follow the same procedure to request access to the MongoDB and have their own unique database credentials and addresses. Since Porter Proxy exposes the MongoDB with its own interface and protocol, users can build their applications using existing database drivers, libraries in different programming languages. They do not need to learn new APIs and write data retrieval functionalities from scratch.

6. CONCLUSION AND DISCUSISON

In this paper we describe a lightweight architecture called Porter Proxy to enable securely sharing databases on the Web. Porter Proxy exposes the same interfaces as the underlying databases to enable the reuse of existing application code, and separates access control from the underlying databases to reduce credentials management cost. We implement a prototype of Porter Proxy in Node.js and we demonstrate and evaluate Porter Proxy using several use cases.

Porter Proxy can encourage users to share databases in an easy and secure way, and improves the ability to build analytics in a decentralised manner. Based on the same sockets manipulation technique we work towards generalising Porter Proxy to incorporate various access control and authentication mechanisms and integrating it with common proxy functionalists. On the Web Observatory platform it allows for the reuse of applications developed using native database APIs.

Nevertheless, there are challenges and trade-offs that need to be considered. On the one hand, programmers can develop applications using existing libraries based on native database APIs, on the other hand it does not encourage the exposure of data resources using harmonised HTTP APIs that can be used by pure client-side web applications. It is possible that both approaches will be accommodated on the Web Observatory ecosystem: data resources will be exposed via native database APIs (using architectures such as Porter Proxy) and via harmonised HTTP APIs at the same time.

7. REFERENCES

[1] Aastha Madaan, Thanassis Tiropanis, Srinath Srinivasa, and Wendy Hall. Observlets: Empowering analytical observations on web observatory. In *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, pages 775–780, 2016.

[2] R. Tinati, X. Wang, T. Tiropanis, and W. Hall. Building a real-time web observatory. *IEEE Internet Computing*, 19(6):36–45, Nov 2015.

[3] Thanassis Tiropanis, Wendy Hall, Jim Hendler, and Christian de Larrinaga. The web observatory: a middle layer for broad data. September 2014.

[4] Thanassis Tiropanis, Xin Wang, Ramine Tinati, and Wendy Hall. Building a connected web observatory: architecture and challenges. June 2014.