Addressing Privacy Leakage from Search Engine Logs

Mike Stunes

August 28, 2008

1 Introduction

Search engines have the ability to collect massive amounts of data about their users. The ability to use this data is invaluable for research to improve search engines, and to provide useful, personalized services to users. The type of data collected by search engines carries an enormous privacy risk for users, however. Users often reveal very personal information through their search engine usage, and the collection of a number of search queries can allow detailed profiling of persons.

2 Motivation

In order to address the privacy problems that arise with collecting users' data from search engines, most solutions rely on traditional access control methods, which while practical and useful, do have their limitations. MIT CSAIL's Decentralized Information Group (DIG), with which I have been working this summer, has been working on alternative methods of information access control methods, in the form of policy-based data use control, where access to information is logged in such a way that it may be used only for allowed purposes, and violations of applicable policies can be tracked and violators held accountable [6].

In order to implement this policy-based access control, there is a need for a "complete accountability architecture," which is a collection of software that completely manages all of the information necessary for a policy-based system.

This complete architecture would have a server holding sensitive data and applicable policies that logs the usage of this data, and a client that queries the server for data and also logs how the data that is returned is being used. The system would also contain reasoners that can check that the logged usage is compliant with the applicable policies.

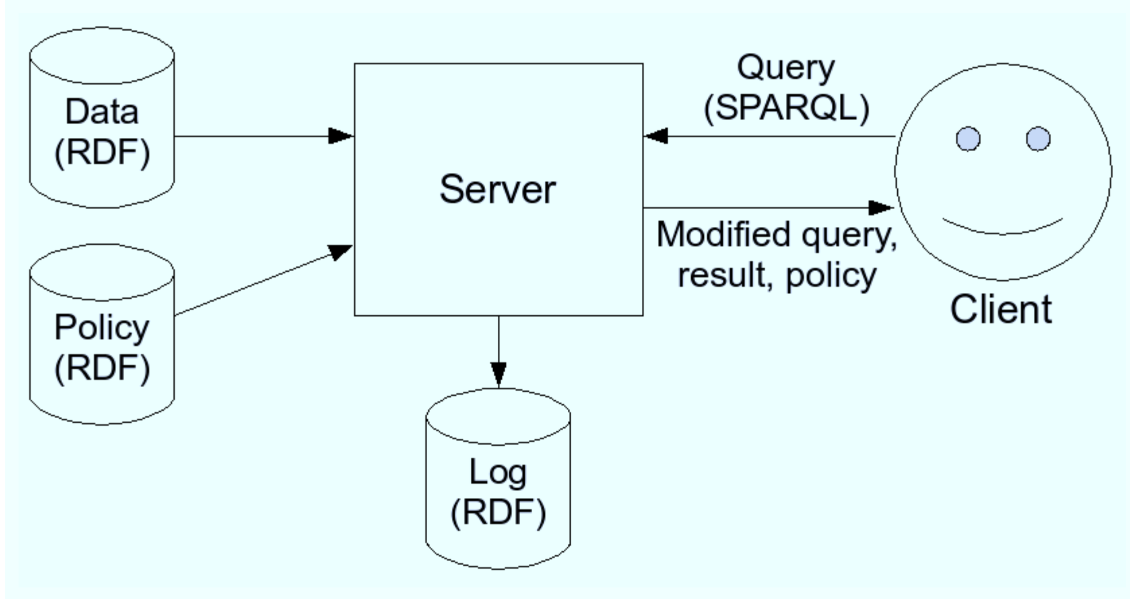

Figure 1: Accountability framework architecture.

3 Architecture

The accountability framework system exists as a set of interoperating components, including a custom SPARQL [5] endpoint server, matching Python client library, DIG's `AIR` [3] and `cwm` [1] reasoners, and MySQL databases for RDF triple storage.

3.1 Features

The server is a custom HTTP server, written in Python, that implements most of the features that are necessary for it to integrate into our complete accountability framework. It handles authentication of requesters using OpenID [2], logging queries in RDF, and secure client/server communication using SSL.

The server uses Python's `BaseHTTPServer` for its HTTP functionality, and implements SSL using the `pyOpenSSL` library. It also uses `rdflib` [4] with a persistent MySQL store for RDF storage, querying, and logging, and uses `python-openid` for OpenID authentication.

3.2 Access Policy

Access policy on the server is implemented by allowing the server administrator to define a custom function to enforce access control. This function receives as input the OpenID identifier of the client, and outputs a boolean value corresponding to whether access is granted, based on whatever reasoning system it chooses.

This allows an extremely high amount of flexibility in creating access control policy. The example code included with the server includes a function that uses the `cwm` reasoner with a set of rules written in N3.

3.3 Usage Policy

The server also has the ability to attach a usage policy to data that is returned to clients, accessible by navigating to the path `/policy` from the server's base URI. Similar to its handling of access control policy, the document to be returned to the client upon request is handled dynamically by a custom function. A link to the policy is attached to the XML-formatted query results with the `<sparql:link>` tag, included in the header.

3.4 Client

In order to enable the client to be used interactively, a matching Python client library was created. The library consists of a class, `SparqlWrapper`, which abstracts the public interface of the server. Details on the use of the client are provided in the Non-Interactive Example Request section, below.

4 Example Requests

This section provides complete walkthroughs of example requests for data on the server. For the purpose of these demonstrations, assume that there is a server running on `https://example.org:8080`.

4.1 Interactive

This scenario describes the sequence of actions that take place when a user accesses the server with an interactive client, such as a Web browser.

First, the client navigates his/her browser to the base URI of the server, which in this example is `https://example.org:8080`. Upon receiving the request, `SparqlServer` (a subclass of `BaseHTTPServer.HTTPServer`) calls the `do_GET` method of `SparqlRequestHandler`. `do_GET` extracts certain information from the request (such as the URL's CGI parameters), attaches a timestamp to the request instance, and calls the appropriate handler function inside of `SparqlRequestHandler`. In this case, it will call the `showEntryPage` method, which checks if the client already has a valid session (more on this later), and if not (as is the case now), presents the client with a login page. Had the client already had a valid session, he/she would be redirected to the interactive webform instead, which we will see shortly.

When the user submits his/her identifier, the form data is submitted to `https://example.org:8080/-login`, which after invoking `do_GET`, calls `showLogin`. `showLogin` creates a session ID for the user; creates a new dictionary of session information (stored in the server's `sessions` dictionary); checks if the user is allowed to access the server (see Access Policy, above); inserts the session ID, IP address, hostname, and OpenID identifier of the client into the session dictionary; and redirects the client to his/her OpenID Provider (OP).

The client's OP will redirect him/her to the path `/loginComplete`, which invokes `showLoginComplete`. This function loads the session dictionary which was created earlier, verifies the client's OpenID authentication, sets a session cookie on the client's browser that holds a session ID, and redirects the client to the interactive webform.

When the client reaches the interactive webform, he/she has the option of entering a query directly, or choosing from a list of prewritten sample queries. (Note that this particular webform was designed primarily as a demonstration of the server's capabilities, and can easily be replaced with a different webform designed for the purpose at hand.) Also, in this demonstration webform, the user has the option of viewing a list of queries logged with his/her OpenID identifier.

The webform submits its data to the `sparql` path, which is the actual SPARQL-endpoint-like interface to the server. It invokes `showSparql`, which loads the existing session dictionary, performs a set of security checks (defined in the `doSecurityChecks` method), logs the query, calls out to the query modification interface (which currently does nothing), runs the query, attaches the usage policy to the results, and returns the results to the client.

4.2 Non-interactive

This scenario describes the sequence of actions that take place when a user accesses the server with a non-interactive client, using the matching Python client library, and also describes how OpenID authentication is used in this situation. The OpenID process used is summarized in Figure 2.

First, the user creates a new `SparqlWrapper` instance:

```
>>> import sparql_client
>>> from sparql_client import SparqlWrapper
>>> foo = SparqlWrapper('https://example.org:8080')
```

At some point, before running queries, the client authenticates to the server:

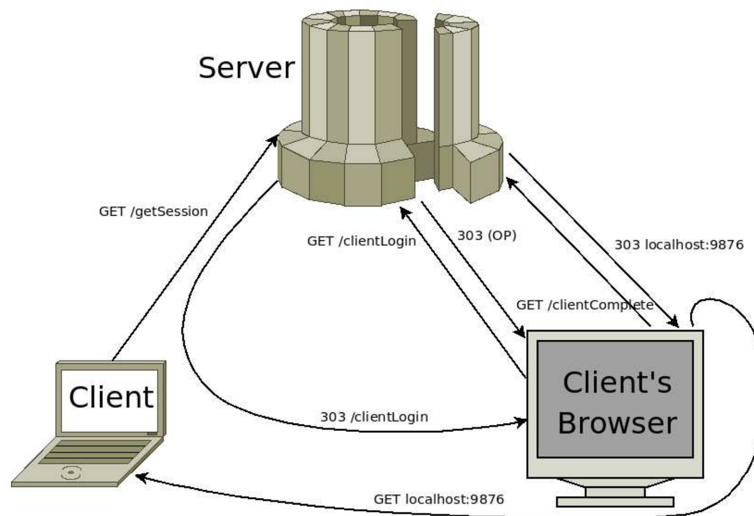

Figure 2: Non-interactive OpenID authentication process.

```
>>> foo.authenticate('http://some_guy.sample_openid_provider.com')
```

This causes the client to request the page at the path `/getSession`, which invokes the method `showGetSession`. This function creates a new session ID and session dictionary; stores the session ID, IP address, hostname, and OpenID identifier in the dictionary; checks if the user is allowed to access the server (see Access Policy, above); and redirects the client to the path `/clientLogin` on the server. Back on the client side, the `authenticate` function opens an HTTP server on `localhost:9876`, which will be explained shortly.

Upon receiving this redirect, the client library captures the URL it is pointing to, extracts the session ID (stored in the URL) and saves it, and opens the user's browser, pointed to the redirect's URL. This causes the user's browser to request the page at `/clientLogin`, which loads the existing session dictionary and redirects the browser to the user's OP.

The user's OP redirects the browser to the page at `/clientComplete`, which loads the existing session and verifies the user's OpenID. It then sends a redirect to `localhost:9876`, causing the user's browser to make a request on the HTTP server that the client library started earlier. This serves as a signal to the client (which has been unaware of events taking place in the browser) that the authentication process is complete.

The client can issue simple queries as follows:

```
>>> foo.setQueryString('SELECT * WHERE {?s ?p ?o . }')
>>> results = foo.query()
```

Currently, the client library provides little more than the ability to retrieve the XML returned from the

server, through a file-like object returned by `query`. It would not be difficult, however, for the client to parse the returned XML into an RDF graph, using `rdflib` or something similar.

5 Conclusions and Lessons Learned

One particular lesson reinforced by this project was the importance of adequate documentation. The client and server needed functionality from several existing libraries that were poorly documented, and the difficulty of using them was eased only by either asking the library's maintainer for assistance, or manually analyzing the library's source code. Of course, doing either of those requires that either the source code or the maintainer is readily available, which is certainly not always the case.

Another lesson that this project taught me was the importance of immediately implementing a clear and extendible structure in the project's code, keeping in mind the flow of events and alternative paths that may be needed to address anything that happens. Several times, major feature implementations and debugging were made difficult by the improperly-structured code, and the only simple remedy was a complete reorganization of the code.

References

[1] T. Berners-Lee, D. Connolly, L. Kagal, J. Hendler, and Y. Schraf. N3Logic: A Logical Framework for the World Wide Web. *Journal of Theory and Practice of Logic Programming (TPLP), Special Issue on Logic Programming and the Web*, 2008.

[2] O. Foundation. Openid. urlhttp://openid.net/.

[3] L. Kagal, C. Hanson, and D. Weitzner. AIR: Explanations for Policy Decisions via Dependency Tracking. In *IEEE Workshop on Policies for Distributed Systems and Networks*, June 2008.

[4] D. Krech. RDFLib. `http://rdflib.net/`.

[5] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, W3C Recommendation 15 January 2008. `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`, 2008.

[6] D. Weitzner, H. Abelson, T. Berners-Lee, C. Hanson, L. Kagal, D. McGuinness, and K. Waterman. Transparent Accountable Data Mining: New Strategies for Privacy Protection. `http://hdl.handle.net/1721.1/30972`, 2007.