Computer Systems Security

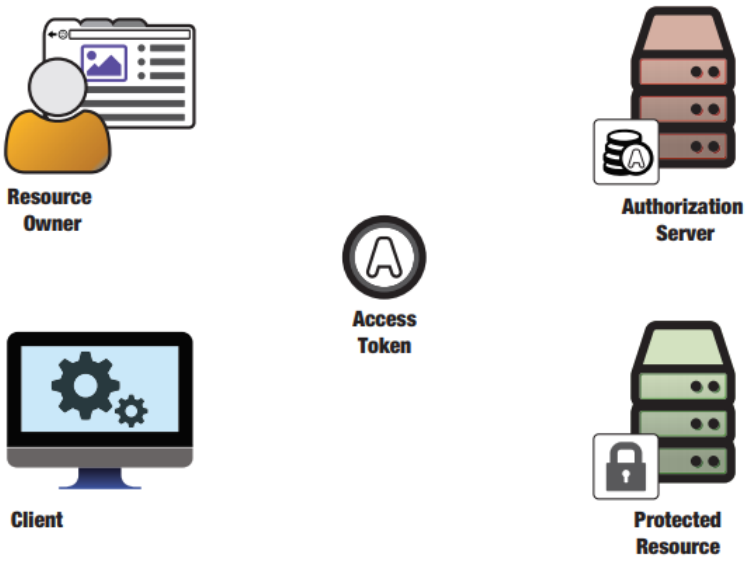
Practical Assignment / Protocol Implementation

OAuth 2.0 in a web application

In this assignment it is intended to protect, by controlling the access, a resource where some operations are defined. To concretize, in this assignment let's assume that the resource is a list of words (like a dictionary), and the operations are the following: *search a word*, *insert a new word*, *delete a word*. These operations have associated three different permissions (called *scopes* in OAuth 2.0): **read** for the search, **write** for the insert, and **delete** for the delete operations.

When a user (called ***resource owner***) selects one operation to be performed on the resource, it needs to possess, and authorized the use, the corresponding *scope*.

The main actors of such a system, using the OAuth 2.0 protocol is shown next.



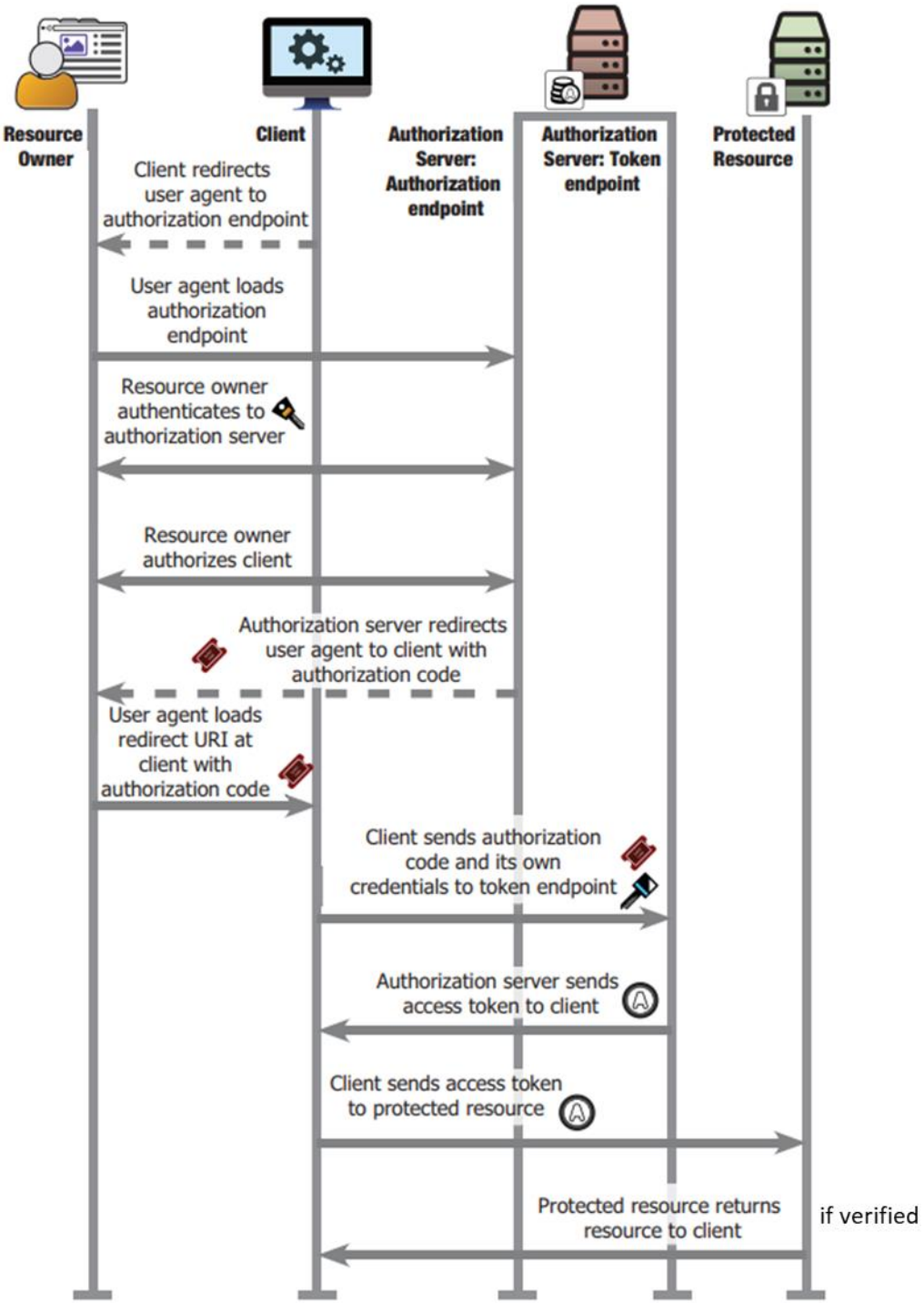
The **protected resource** is managed by a separated server, that accepts the request of an operation (and replies with the result) using a single HTTP request. It needs to know who (a **resource owner**) ordered the request, and if he possesses and has authorized the use of the needed *scope*.

The user (**resource owner**) is accessing a web application (called **client** by OAuth 2.0) that is the interface to the protected resource, allowing him to specify the operations and see the results. The user is sitting in front of a standard browser in his own machine.

When an operation in the **protected resource** is requested in the **client**, and before it calls the **protected resource** to perform it, the client should obtain an **access token** from an **authorization server**. The **authorization server** should obtain directly the **resource owner** identity and ask him what *scopes* he authorizes the **client** to use with the **protected resource**. All that information, supplied directly by the user (**resource owner**), is stored in the **authorization server**, and represented in an **access token** that it sends to the **client**. The **client** than sends the **token**, together with the operation request, to the **protected resource**. The **protected resource** then asks the **authorization server** information about the **token**, which includes the user identity, the *scopes* authorized by him, and the expiration time. Then, using its own internal information, the **protected resource** finds out if that user has the needed *scope* and if he authorized it. If so, it performs the requested operation and returns the result to the **client**.

This is known as the OAuth dance, and constitute the main features of the protocol main variant called the *authorization code grant*.

The next figure shows these interactions between the actors:



As we can see the **resource owner** talks directly with the **authorization server**, through a redirection from the **client** (after he activates an operation to be performed by the **protected resource**).

The **client** (through another redirection from the **authorization server**) does not receive directly the **access token**, but an *authorization code*, that then he exchanges directly with the **authorization server**.

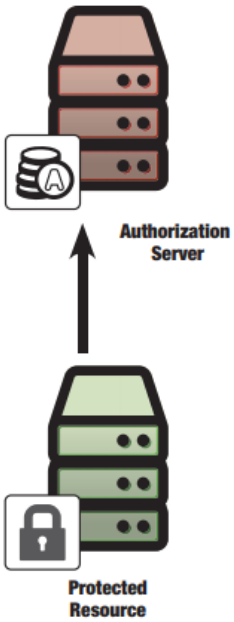
For protection, all the shown interactions should be done with an underlying TLS data exchange (HTTPS), but for this assignment, to not complicate the implementation, we can use plain HTTP.

Other aspects

Access tokens can have several formats. They can contain all the information in them (called JWT – *JSON Web Tokens*, because usually in that case the information is coded in JSON), but must be protected (using, for instance, a JOSE – *JSON Object Signing and Encryption*), or can be *opaque* tokens (just random strings with a certain size).

Access tokens also have a small lifetime. When they expire the **protected resource** does not accept them anymore, and the **client** must obtain another. Sometimes, the **authentication server** supplies the **client** with the *access token* and a *refresh token*, which has a larger lifetime. In that case, and if the **client** needs to do more operations with the same user (**resource owner**), instead of performing a new complete authorization he can use the *refresh token* to obtain a new **access token**, avoiding a new similar dialog between **authorization server** and **resource owner**.

For validation of the **access token**, if it is like the JOSE one, the **protected resource** could do it directly, but needs to share some key with the **authorization server**, which could be complicated. For an *opaque* token it needs to ask the **authorization server** about it, using what is called the *introspection* endpoint (this call should also be authenticated with convenient credentials).



Sample request and reply:

```
POST /introspect HTTP/1.1
Host: localhost:9001
Accept: application/json
Content-type: application/x-www-form-encoded
Authorization: Basic
   cHJvdGVjdGVkLXJlc291cmNlLTE6cHJvdGVjdGVkLXJlc291cmNlLXNlY3JldC0x

token=987tghjkiu6trfghjuytrghj
```

```
HTTP 200 OK
Content-type: application/json

{
  "active": true,
  "scope": "foo bar baz",
  "client_id": "oauth-client-1",
  "username": "alice",
  "iss": "http://localhost:9001/",
  "sub": "alice",
  "aud": "http://localhost:/9002/",
  "iat": 1440538696,
  "exp": 1440538996,
}
```

OAuth 2.0 needs to identify the user (**resource owner**), but OAuth does not perform authentication. Usually it works together with a separated authentication server (for instance OpenID). In our case and for not complicating too much, we allow the **authorization server** to perform a very simple authentication (using user id + password) with data already in the **authorization server** (but the authorization data (users/permissions) should be in the protected resource).

The tasks to do

Implement 3 web applications (for instance in Node.js, or any other web server technology) representing the **client**, **authorization server**, and **protected resource**. They can run on the same machine (different ports). The **authorization server** performs user authentication and knows about user id's and credentials, and the scopes used by the **protected resource**. The **protected resource** knows the user id's and, for each, the scopes that are associated (the authorization table).

Implement the OAuth 2.0 protocol using its specification (see references below), using at least its main features in the *authorization code grant* (begin with the simplest implementation using a step by step addition).

For proof of correctness, let the **client**, **authorization server** and **protected resource** show in real time, and in a browser window, a 'log' of all the information and operations that they perform (stating requests, data (like the tokens …), responses, etc.).

Study and implement correct countermeasures for some possible vulnerabilities, resulting from the threat model of this protocol and interactions, when a deficient implementation is done (see reference - RFC6819).

Report all the features (in terms of the protocol) implemented, how they were, and why, including a comment about your opinion concerning the efficacy of the OAuth 2.0 protocol.

This protocol is now one of most used by major players (Google, Microsoft, etc.) for protecting their resources accessed from web applications. It is a standard described by the IETF in several of their RFCs.

There are many web resources describing and teaching its implementation.

Reference texts

Main reference – RFC 6749 - The OAuth 2.0 Authorization Framework (https://tools.ietf.org/html/rfc6749)

RFC 6750 - The OAuth 2.0 Authorization Framework: Bearer Token Usage (similar URL)

RFC 6819 - OAuth 2.0 Threat Model and Security Considerations (similar URL)

RFC 7636 - Proof Key for Code Exchange by OAuth Public Clients (similar URL)

RFC 7662 - OAuth 2.0 Token Introspection (similar URL)