COMPUTER SYSTEMS SECURITY
OAUTH 2.0 IN A WEB APPLICATION

May 25, 2020

César Pinho - up201604039@fe.up.pt
João Barbosa - up201604156@fe.up.pt
Rui Guedes - up201603854@fe.up.pt
Faculdade de Engenharia da Universidade do Porto

Introduction

Within the scope of the Computer Security Systems course, we were tasked with the implementation of the OAuth 2.0 authorization specification (RFC 6749). The reported implementation of OAuth 2.0 strictly follows this specification and implements, whenever feasible, the suggested recommendations.

It is assumed in this report that the reader is familiarised with OAuth 2.0. Consequently, we will focus on reference key decisions and security features taken into consideration while implementing this protocol.

Regarding the authorization grant, we chose the authorization code grant variation, and in order to prevent the interception of Authorization Codes by third parties, we have used the Proof Key for Code Exchange (PKCE), an extension to the Authorization Code flow.

In relation to the token issuing process, the bearer token type was used for the access token and therefore, the access to the protected resource is made by passing the access token via the authorization header to prevent external entities from obtaining it. Token introspection is used so that resources may query the authorization server for the state of an access token. Also, the refresh token, which can be used to obtain new access tokens, was implemented to increase the protocol's robustness.

Security Considerations

Client Authentication and Impersonation

Client Authentication occurs whenever the Token and Introspection endpoints of the Authorization Server are used. For the purposes of this project, the Client is already pre-registered in this server, with the identifier "oauth-client-1" (Instead of its encoded URL, as specified in Section 2.3 of RFC 6749).

The Authorization Server supports the HTTP Basic authentication scheme [RFC2617], which is used by the Client to transmit its credentials. Authentication is performed via salted hashes, using **PBKDF2** with **SHA-256**, for increased protection against brute-force attacks and rainbow tables. All client credentials are kept within the Authorization Server and are never shared with other entities.

Access to the Authorization endpoint does not require **Client Authentication**, but measures are taken to prevent the exposure of credentials to a potentially malicious client, by authenticating the **Resource Owner**, explicitly requiring him to authorize every permission requested by the client, and validating the redirection URI by ensuring it is registered in the server (In this project's scenario, it is pre-registered with the client).

In an implementation of **OAuth** for use in production, **TLS** must be used on every endpoint used to interact with the end-user, in order to reduce the risk of **Phishing Attacks** (The end-user would need to be educated on this threat in order to be an effective measure).

Access and Refresh Tokens

Both **Access and Refresh Tokens** are hashed, generated with 256 bits of entropy and shared only among their pertaining entities - The **Authorization Server**, the Client whom it was issued, and, for the Access Token, the Resource Servers in which it is valid.

As aforementioned, the client is authenticated before a token is generated. The Authorization Server keeps track of the valid tokens and its attributes.

Access Tokens are short-lived and sent as Bearer Tokens to Protected Resources, via the Authorization Request Header Field.

In an implementation of OAuth for use in production, the transmission of these credentials must be done using TLS.

Authorization Codes and Redirection URI Manipulation

Authorization Codes are hashed, generated with 256 bits of entropy and have a short expiration time. When an Authorization Code is used to redeem an Access Token, the Authorization Server verifies if this code has already been used. If it is attempted to be used twice, the Authorization Server revokes all Access Tokens based on the compromised Authorization Code.

In an implementation of **OAuth** for use in production, the transmission of these credentials must be done using TLS. Furthermore, the redirection URI should use TLS if it points to a resource, and the redirection endpoint must require the use of TLS if the Authorization Code is being used to authenticate the Resource Owner on the Client.

The redirection URI is not only required to be registered but is also bound to the Authorization Code. In order to detect and prevent manipulation of the Redirection URI, the Authorization Server ensures that the redirection URI used to obtain the Authorization Code is the same as the one provided when exchanging it for an Access Token.

PKCE was implemented to prevent the interception of Authorization Codes by third parties in means of transfer not protected by TLS (at the OS level, for example). The **Code Verifier** generated by the Client has 256 bits of entropy, and it uses the **S256** method for the **code challenge**.

Cross-Site Request Forgery

There's a potential CSRF attack where an attacker injects his own Authorization Code / Access Token to the redirection URI, tricking the victim into accidentally exposing sensitive information to an attacker-controlled resource, rather than their own. This is counteracted by adding a state value (hash of the user-agent session cookie id) to the redirection URI. The Client validates the request by matching this state value with its own when the end user's user-agent is redirected back to the Client by the Authorization Server.

This state value is sent to the Authorization Server on the very first request of the authorization flow, allowing for the detection and prevention of CSRF attacks on one of the authorization endpoints as well.

Clickjacking

To prevent a **clickjacking attack** on the authorization flow, where a resource owner is tricked into authorizing a client without realizing, the Authorization Server has the **X-Frame-Options HTTP response header set to DENY**, disallowing its pages from being rendered inside an invisible frame on newer browsers.

Introspection Endpoint

When a Resource Server uses the Introspection Endpoint, the Authorization Server authenticates the Resource and ensures that the token hasn't been expired or revoked.

To avoid the values of Access Tokens from leaking into logs, only the HTTP POST method is allowed. Furthermore, the introspection response is done so that the internal state of the Authorization Server is not disclosed in case of an invalid token.

In an implementation of OAuth for use in production, the Introspection Endpoint, requests, and responses must use TLS.

Bearer Token

Threat Model [RFC 6819]

The OAuth 2.0 Threat Model was followed in order to verify how the presented security features may serve as countermeasures to known attacks. We present the following list of threats that are Covered - all countermeasures are in place to detect and prevent the threat, and Mitigated - some countermeasures are in place to mitigate the threat. For the mitigated threats, a list of the implemented countermeasures per threat is presented.

Client

Covered

- **4.1.5. Open Redirectors on Client**
- **4.2.3. Malicious Client Obtains Existing Authorization by Fraud**
- **4.2.4. Open Redirector**
- **4.4.1.3 Online Guessing of Authorization "codes"**

- **4.4.1.5. Authorization "code" Phishing**
- **4.4.1.7. Authorization "code" Leakage through Counterfeit Client**
- **4.4.1.8. CSRF Attack against redirect-uri**
- **4.4.1.9. Clickjacking Attack against Authorization**
- **4.5.3. Obtaining Refresh Token by Online Guessing**
- **4.6.3. Guessing Access Tokens**
- **4.6.6. Threat: Leak of Confidential Data in HTTP Proxies**

Mitigated

- **4.1.2. Obtaining Refresh Tokens**
  - Client authentication on every refresh request.
- **4.1.3. Obtaining Access Tokens**
  - Short access token lifetime.
- **4.3.2. Obtaining Access Tokens from Authorization Server Database**
  - Access tokens are hashed.
- **4.3.4. Obtaining Client Secret from Authorization Server "Database"**
  - Client Secrets are hashed.
- **4.4.1.1. Eavesdropping or Leaking Authorization "codes"**
  - Client authentication during the authorization flow, allowing reliable binding of the authorization code to the client id.
  - Short authorization code lifetime.
  - Authorization code has a one-time usage restriction.
  - On multiple attempts of authorization code redemption, all tokens granted based on that authorization code are revoked.

- **4.4.1.2. Obtaining Authorization "codes" from Authorization Server Database**
  - Authorization “codes” are hashed.
- **4.4.1.4. Malicious Client Obtains Authorization**
  - Client authentication during the authorization flow.
  - Validation of a pre-registered redirection URI.
  - End-user must consent to the authorization (however the Client identity, and the purpose and scopes of the authorization are not clearly explained, mostly due to the context of the project).
- **4.4.1.11. DoS Attacks That Exhaust Resources**
  - Credentials have a high entropy of 256 bits.
- **4.4.1.13. Code Substitution (OAuth Login)**
  - Client authentication when exchanging an authorization code for an access token.
- **4.5.2. Obtaining Refresh Token from Authorization Server “Database”**
  - Refresh Token is bound to the Client Id (In fact, the client must be authenticated when attempting to obtain a new access token with a refresh token).
- **4.6.1. Eavesdropping Access Tokens on Transport**
  - Short access token lifetime.
  - Access token is bound to the client id via the bearer token.
- **4.6.7. Token Leakage via Log Files and HTTP Referrers**
  - Access tokens are sent as POST parameters.
  - Bearer token preventing the abused of leaked access tokens.
  - Short access token lifetime.

Conclusion

The implementation of OAuth 2.0 allowed us to not only get an insight into how it works but also motivated us to dive deep into its specification and get to know in detail how its various intricacies contribute to OAuth's security and usability (For example, Refresh tokens allow clients to request a new token when it expires, permitting the usage of short-lived access tokens without compromising usability).

Despite its apparent complexity, OAuth 2.0 is actually focused on simplicity and applicability, particularly in relation to its predecessor. There are multiple grant types to support multiple types of clients (web-based, native, IoT), client applications are not required to have cryptography, relying on HTTPS instead (bearer tokens), and the concerns of user authorization (Authorization Server) and API calls (Resource Server) are separated.

For these reasons, OAuth 2.0 is the industry-standard protocol for authorization, enabling third-party applications (Clients) to obtain access to a service on behalf of a Resource Owner securely.