Ari-Pekka Koponen

A secure OAuth 2.0 implementation model

Master's Thesis in Information Technology

August 12, 2016

University of Jyväskylä

Department of Mathematical Information Technology

Author: Ari-Pekka Koponen

Contact information: arkopone@student.jyu.fi Timo Hämäläinen

Title: A secure OAuth 2.0 implementation model

Työn nimi: Tietoturvallinen OAuth 2.0 toteutusmalli

Project: Master's Thesis

Page count: 66+0

Abstract: A growing amount of data is stored in the cloud and the number of web services is soaring. This has created a need for users to authorize third party applications to access their resources. The OAuth 2.0 authorization framework aims to offer an open and standardized protocol for authorization. However, implementing OAuth 2.0 securely requires a great deal of knowledge of both the OAuth 2.0 specification and web security in general. The present research will take a form of a constructive research study. The aim is to construct a secure model for web developers implementing OAuth 2.0. The features of a secure OAuth 2.0 implementation are identified. Then, OAuth 2.0 is implemented. The security of this implementation is tested and the results reviewed.

Keywords: OAuth 2.0, authorization, security

Suomenkielinen tiivistelmä: Pilveen tallennetaan yhä enemmän dataa ja verkkopalveluiden määrää kasvaa jatkuvasti. Tämän vuoksi käyttäjillä on yhä useammin tarve sallia kolmannen osapuolen sovelluksille pääsy verkkopalveluihin tallennettuun dataan. OAuth 2.0 valtuutuskehys pyrkii tarjoamaan avoimen ja standardoidun protokollan valtuuttamiseen. OAuth 2.0:n tietoturvallinen toteutus vaatii kuitenkin laajaa tuntemusta OAuth 2.0:n spesifikaatiosta ja verkkopalveluiden tietoturvasta yleisesti. Tämän konstruktiivisen tutkimuksen tarkoituksena on konstruktoida web-kehittäjille tietoturvallinen malli OAuth 2.0 sovelluskehyksen toteutusta varten. Tutkimuksessa tunnistetaan tietoturvallisen OAuth 2.0 toteutuksen ominaisuudet. Tämän pohjalta tehdään OAuth 2.0 toteutus. Toteutuksen tietoturva testataan ja tulokset analysoidaan.

Avainsanat: OAuth 2.0, valtuutus, tietoturva

List of Figures

List of Tables

Contents

1 Introduction

1.1 Background

As noted by IDC and Cisco the amount of data stored in the Internet is growing rapidly (Cisco 2016; Gantz and Reinsel 2011). People store an increasing amount of their data in the Cloud and often have a need to authorize third party applications to access the data.

The OAuth 2.0 authorization framework aims to offer an open and standardized protocol for authorization (*OAuth 2.0* 2016). It is a flexible and extensible framework that has four different grant types with accompanying authorization flows and a number of optional implementation choices (Hardt 2012). As a result, implementing OAuth 2.0 securely requires a great deal of knowledge of both the OAuth 2.0 specification and web security in general (Sun and Beznosov 2012).

Internet Engineering Task Force (IETF) responsible for the OAuth 2.0 specification has provided a number security considerations for OAuth 2.0 (Hardt 2012, 49-56; Lodderstedt, McGloin, and Hunt 2013). In addition, a number of studies assessing the security of OAuth 2.0 and some of it's most prominent implementations have been published (Bansal, Bhargavan, and Maffeis 2012; Chari, Jutla, and Roy 2011; Pai et al. 2011; Sun and Beznosov 2012; Xu, Niu, and Meng 2013; Yang and Manoharan 2013). Some of these studies provide also guidelines for OAuth 2.0 implementations (Bansal, Bhargavan, and Maffeis 2012, 257-258; Sun and Beznosov 2012, 387-388; Yang and Manoharan 2013, 276). However, none of these resources provide a full model presenting a secure OAuth 2.0 implementation.

The present research will take a form of a constructive research study. The aim is to construct a secure implementation model for web developers implementing OAuth 2.0. First, with the help of current literature the features of a secure OAuth 2.0 implementation are identified. Based on this, OAuth 2.0 is implemented. The security of the resulting implementation is tested against threats found in the literature. In the end, the results will be reviewed and conclusions provided.

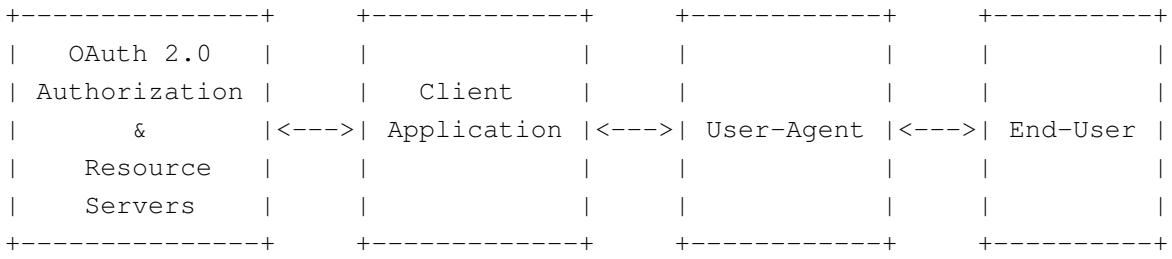


Figure 1. Abstraction of the actors in the model.

1.2 Research Method

The aim of the present research is to construct a secure model for web developers implementing OAuth 2.0. In order to be able to provide a model that is useful, the implementation model has to have a sufficiently narrow target audience. Thus the implementation will be directed towards web developers, who are developing a web application with an application programming interface (API) utilizing OAuth 2.0. The client applications use OAuth 2.0 for authorization and have end-users, who use the client through a user-agent, usually a web-browser. These actors and their interactions are illustrated in Figure 1 .

This research will take the form of constructive research study. The key idea of constructive research is to use and combine the existing knowledge in novel ways, which leads to a construction. Typical constructs used in research and engineering include artifacts like models, diagrams, plans, algorithms, and software development methods. Constructivist solutions are usually designed and developed, not discovered. Even though a lot of inspiration is found e.g. from nature. Constructive research artifacts include knowledge about how a domain specific problem can be solved. The results can have both practical and theoretical relevance. (Crnkovic 2010, 360,363.) Lindholm (2008) summarizes the seven steps of the constructive research approach:

(1) ”to find a practically relevant problem, which also has research potential;”
(2) ”to examine the potential for long-term research co-operation with the target organisation;”
(3) ”to obtain a general and comprehensive understanding of the topic;”

(4) ”to innovate and construct a theoretically grounded solution idea;”
(5) ”to implement the solution and test whether it works in practice;”
(6) ”to examine the scope of the solution’s applicability; and”
(7) ”to show the theoretical connections and the research contribution of the solution.”

The research question in hand is practically relevant. The difficulty of implementing OAuth 2.0 is found on the academic literature and is expressed even by Hammer (2012a), who was the previous editor of the OAuth 2.0 specification. Also, even big web service providers like Facebook had problems with implementing OAuth 2.0 securely (Hammer 2012b; cf. Cluley 2011).

Thus, the research will start with step 3. In this study, the security features of a secure OAuth 2.0 implementation are identified using related publications and best practices recommended by major web service providers, who have implemented OAuth 2.0 in their APIs. Based on these theoretically grounded security features and implementation choices, an implementation model is formed. Using the model, OAuth 2.0 will be implemented. The implementation will be assessed and tested against common web service attack patterns.

To conclude, the scope of the solution’s applicability is examined, the theoretical connections and the research contribution of the solution are shown. Further research ideas are also provided.

2 Previous work

The security of OAuth 2.0 has been considered in a number of publications. The OAuth 2.0 specification itself has a section dedicated to security considerations (Hardt 2012, Section 10, 49-56). Probably the most comprehensive take on OAuth 2.0 security is the OAuth 2.0 Threat Model published by IETF (Lodderstedt, McGloin, and Hunt 2013), which provides a comprehensive threat model for the OAuth 2.0 framework and offers respective security considerations. In addition, a number of studies assessing the security of OAuth 2.0 and some of it's most prominent implementations have been published (Bansal, Bhargavan, and Maffeis 2012; Chari, Jutla, and Roy 2011; Pai et al. 2011; Sun and Beznosov 2012; Xu, Niu, and Meng 2013; Yang and Manoharan 2013). Some of these studies provide also guidelines for OAuth 2.0 implementations (Bansal, Bhargavan, and Maffeis 2012, 257-258; Sun and Beznosov 2012, 387-388; Yang and Manoharan 2013, 276). However, none of these resources provide a full model presenting a secure OAuth 2.0 implementation for web developers.

Bansal, Bhargavan, and Maffeis (2012), Li and Mitchell (2014), Sun and Beznosov (2012) and Yang and Manoharan (2013), studied prominent SSO (Single sign-on / Social sign-on) services, which implemented OAuth 2.0. Bansal, Bhargavan, and Maffeis (2012) used formal analysis to analyze multiple OAuth 2.0 implementations on popular websites (including Facebook, Yahoo and Twitter). Li and Mitchell (2014) studied a number of major Chinese identity providers and their relying parties. Sun and Beznosov (2012) studied the server-side implementations of three big Identity Providers (IdP; Facebook, Google and Microsoft) and the client-side implementations of 96 popular web services that supported Facebook's SSO. Yang and Manoharan (2013) did an experimental analysis of the weaknesses of OAuth 2.0 based on their attacker model using both a local and Google's implementation.

All of these studies found weaknesses in the implementations, which made the interception and misuse of tokens possible or allowed access to the user's resources through the service provider's website. Some of the weaknesses were due to lack of adhering to the specification, e.g. connections lacking Transport Layer Security (TLS) or unprotected authorization endpoints. However, in other cases the weaknesses were caused by choices allowed by the

specification, e.g. choosing not to use the ”state”-parameter (cf. Hardt 2012, 25). (Bansal, Bhargavan, and Maffeis 2012, 256-257; Li and Mitchell 2014, 538-540; Sun and Beznosov 2012, 382-384; Yang and Manoharan 2013, 275-276.)

The difficulty of implementing OAuth 2.0 securely is a recurring theme in the publications dealing with OAuth 2.0 security (cf. e.g. Cherrueau et al. 2014, 235-236; Sun and Beznosov 2012, 378.). When developing software, the developer often has to make choices between simplicity and security (Yang and Manoharan 2013, 276). Many of the found weaknesses were due to choosing a simpler option instead of the more secure one. The flexibility of the OAuth 2.0 specification leaves these implementation choices to the developer. Even though it is possible to implement OAuth 2.0 securely and it offers one of the best frameworks for authorization (Bansal, Bhargavan, and Maffeis 2012, 258), without a deep knowledge of web security and the OAuth 2.0 specification the implementation is likely to be insecure. (Sun and Beznosov 2012, 378.)

3 Secure Authorization Methods

Traditionally, authentication has been done using a client-server authentication server model, where the client requests a protected resource on the server by authenticating using the resource owner's credentials. In this model, the only way for the resource owners to provide third-party applications access to protected resources is to share the resource owner's credentials with the third party. This approach has several problems and limitations (Hardt 2012, 3-4):

- The third-party applications have to save these credentials for future use, often in a clear-text form.
- Servers are required to support password authentication, despite the fact that passwords have inherent security weaknesses.
- The resource owner cannot restrict the duration or the scope of access, which gives the third-party applications overly broad access to the resource owner's protected resources.
- The only way to revoke access to an individual third party is to change the password, which will revoke the access of all third parties.
- Compromise of any third-party application that compromises the end-user's password will compromise all the data protected by the password.

A solution for these issues requires additional means for authorization. OAuth and the Security Assertion Markup Language (SAML) are two common standards that enable authorization for use with HTTP. Both have two major versions that are not backwards compatible. Where SAML 2.0 shares same uses cases with it's predecessor (*Differences between SAML 2.0 and 1.1* 2008), OAuth 2.0 is significantly different from 1.0. The OAuth 1.0 protocol (Hammer-Lahav 2010) was the result of a small ad hoc community effort. The OAuth 2.0 framework shares very few implementation details with OAuth 1.0 and is based on additional use cases and extensibility requirements. (Hardt 2012, 5.) While OAuth 1.0 was very popular and widely deployed, most big service providers now support and recommend OAuth 2.0.

OAuth aims to solve the issues of the client-server model by introducing and authorization layer and separating the role of the client from that of the resource owner. In OAuth the client is issued a different set of credentials than those of the resource owner. Instead of the resource owner's credentials the client obtains an access token from an authorization server. Access tokens are issued with the approval of the resource owner. Once issued, the client uses the access token to access the protected resources hosted by the resource server. (Hardt 2012, 5.)

In SAML security information is expressed in the form of assertions that are included in requests. Assertions can convey information about previously performed authentication acts and authorization decisions about whether subjects are allowed to access certain resources. Assertions are issued by SAML authorities. SAML defines a protocol by which clients can request assertions and get a response from SAML authorities. (Cantor et al. 2005, 11; Maler, Mishra, and Philpott 2003, 8, 11.)

One difference between SAML and OAuth is how authorization information is conveyed. In OAuth the authorization information is connected to an access token. An access token is expressed as a string (a sequence of characters). It might have the authorization information encoded in it or it can be only a reference to the authorization information that is stored on the authorization server. In the latter case, when an access token is used for authentication, the resource server has to make a round trip to the authorization server in order to check if the token allows access to the protected resource. In SAML the authorization is signed and stored in the assertion that is included in the request. The server hosting the protected resource can then check the signed assertion to check whether it is valid and if it allows access to the protected resource.

A second difference is in the intended scope of the two standards. OAuth 2.0 is originally designed for authorization and not for authentication. Therefore, access tokens do not always carry information about who can use them. When using bearer tokens (cf. subsection 5.1.5) any party in possession of the token can use it to access the protected resources allowed by the token. (Jones and Hardt 2012, 3.) SAML was designed both for authorization and authentication. SAML assertions usually carry a subject and all SAML-defined assertion statements are associated with a subject. (Cantor et al. 2005, 11) Thus, a SAML assertion

with a subject cannot be used by anyone else but the subject.

The two standards have some interoperability. IETF has specified a SAML 2.0 profile for OAuth 2.0 client authentication and authorization grants (Campbell, Mortimore, and Jones 2016), which allows using SAML 2.0 assertions in order to gain an access token. The profile has already been implemented by e.g. Salesforce (cf. *OAuth 2.0 SAML Bearer Assertion Flow*).

In the web development context, OAuth 2.0 is widely used for authorizing access to web service APIs and for Social/Single Sign-On (e.g. by Google, Twitter, Microsoft, Facebook). Usual use case for SAML in major web services' public APIs is integrating Single Sign-On services (as done e.g. by Google Apps and Microsoft Azure).

4 OAuth 2.0 Overview

4.1 OAuth 2.0 Introduction and Definitions

The OAuth 2.0 authorization framework aims to offer an open and standardized protocol for authorization. OAuth 2.0 makes it possible for end-users to allow third-party applications limited access to their personal data without exposing their own credentials. Instead, the third-party are issued their own set of credentials upon authorization, called access tokens. These tokens have a limited life-time and can be optionally refreshed using refresh tokens.

OAuth 2.0 defines four different roles: resource owner, resource server, client, and authorization server. Resource owner is "[a]n entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user." Resource server is "[t]he server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens." Client is "[a]n application making protected resource requests on behalf of the resource owner and with its authorization." Authorization server is "[t]he server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization." (Hardt 2012, 6.)

OAuth 2.0 offers multiple different options how the authorization flow can be implemented. The specification has four grant types and the IETF has published two additional profiles (cf. Jones, Campbell, and Mortimore 2015; Campbell, Mortimore, and Jones 2016), a framework for using assertions with OAuth 2.0 (cf. Campbell et al. 2015) as well as drafted a specification for using Message Authentication Codes (MAC) with OAuth 2.0 tokens (cf. Richer et al. 2014). However, a general picture of the protocol flow can be abstracted. Figure 3 illustrates the abstracted version of the protocol flow and the interactions between the four different roles in authorization and acquiring a protected resource. The flow includes the following steps: (Hardt 2012, 7-8.)

(A) ”The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or preferably indirectly via the authorization server as an intermediary.”

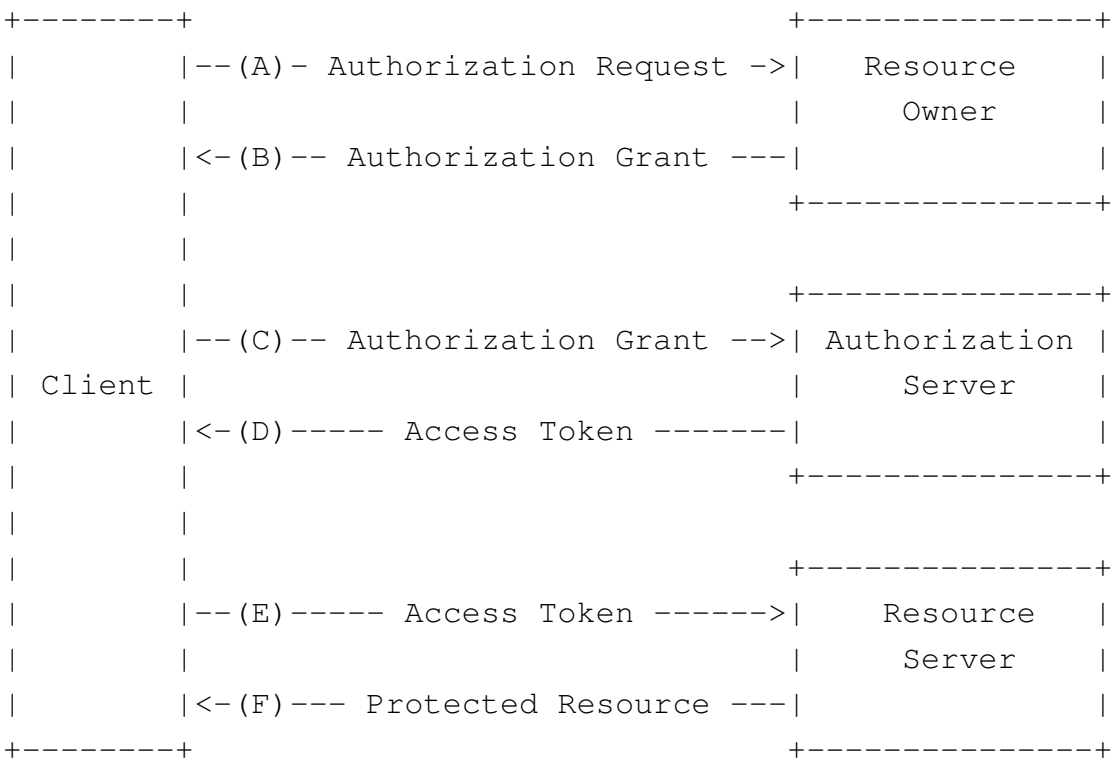


Figure 2. Abstract Protocol Flow (Hardt 2012, 7).

(B) ”The client receives an authorization grant, which is a credential representing the resource owner’s authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.”

(C) ”The client requests an access token by authenticating with the authorization server and presenting the authorization grant.”

(D) ”The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token.”

(E) ”The client requests the protected resource from the resource server and authenticates by presenting the access token.”

(F) ”The resource server validates the access token, and if valid, serves the request.”

OAuth defines two client types, based on their ability to maintain the confidentiality of their client credentials: confidential and public. Confidential clients can maintain the confiden-

tiality of their credentials. A confidential client could be e.g. a web application on a secure server. Public clients cannot maintain the confidentiality of their credentials. E.g. a client executing on a resource owner's device, such as an installed native application or a user-agent-based application. (Hardt 2012, 14-15.)

4.2 OAuth 2.0 and Web Security Weaknesses

As McGraw (2012, 662) well defines, "[s]oftware security is the idea of engineering software so that it continues to function correctly under malicious attack." Different software have different definitions of correct operation and different expectations for security. For a simple web application the sufficient authentication method might be a simple username-password combination for signing in and a specific cookie in the following requests, where as a high-security application might require a biometric identification such as a correct fingerprint scan or an iris recognition. As a whole, security is a combination of multiple different factors. Different software have different kinds security requirements and threats, and therefore they may need to ensure correctness of operation differently and provide different kinds of security mechanisms.

OAuth 2.0 is a authorization framework developed for the web and therefore it faces the same threats as all web applications. In order to better understand these threats, the most important web application security weaknesses presented in the OWASP Top 10 list (OWASP 2013), later referred simply as OWASP T10, and their relationship to OAuth 2.0 are shortly introduced in this section. A more through treatment of OAuth 2.0 security weaknesses is available in the OAuth 2.0 threat model (Lodderstedt, McGloin, and Hunt 2013).

First weakness on the OWASP T10, A1, is "Injection". "Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization." (OWASP 2013, 6.) OAuth 2.0 requires input data both from the third-party clients as well as from the user. Some of this data can be included in database queries, which can result in injection unless the data is properly sanitized and escaped. Also, depending on the implementation, tokens might need

encoding and decoding. Similar to parsing, it is prone to injection attacks (OWASP 2013, 7).

OAuth 2.0 offers an authentication method. Also, an OAuth 2.0 authorization server is likely to use some sort of session management when authenticating an resource owner. A proper implementation is needed to mitigate risk A2, "Broken Authentication and Session Management". "Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities." (OWASP 2013, 6.) If the OAuth 2.0 implementation has flaws, some or all of these threats can be realized.

Risk A3, "Cross-Site Scripting (XSS)", is likely to happen on resource owner authorization screens, unless the data shown to the user, such as the requested scope, is properly validated. "XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites." (OWASP 2013, 6.)

OAuth 2.0 is often used to protect web resources that have a single Uniform Resource Identifier (URI) that is publicly available or relatively easy to guess. Therefore it is especially important to mitigate weakness A4, "Insecure Direct Object References". "A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data." (OWASP 2013, 6.) When direct object references are available and it is not possible to use per user or session indirect object references, resource servers using OAuth 2.0 have to issue proper access checks every time they serve protected resources.

Security weakness A5, "Security Misconfiguration" is at the very center of this study. "Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date." (OWASP 2013, 6.) Our model aims to define a secure

implementation model that includes a secure configuration for OAuth 2.0 in order to allow developers to avoid security misconfiguration.

OAuth 2.0 introduces new security credentials, authorization codes and tokens, to applications. These credentials need to be protected from risk A6, "Sensitive Data Exposure". "Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data [...]. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser." (OWASP 2013, 6.)

Although OAuth 2.0 implementations do not have a large user interface (UI), it is important to ensure that the few requests generated from the UI are properly checked in order to mitigate A7, "Missing Function Level Access Control". "Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization." (OWASP 2013, 6.)

In some of the OAuth 2.0 authorization flows the authorization happens within a user-agent. It is important that these authorization flows use security tokens in the authorization screens, in order to mitigate threat A8. "Cross-Site Request Forgery (CSRF)". "A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim." (OWASP 2013, 6.)

Weakness A9, "Using Components with Known Vulnerabilities", is not directly related to implementing OAuth 2.0, but rather a more general security consideration. "Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts." (OWASP 2013, 6.)

Some OAuth 2.0 grant types use redirection as a means to take the user back to the third-party

client after authorization. This makes OAuth 2.0 vulnerable to the threat A10, "Unvalidated Redirects and Forwards". "Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages." (OWASP 2013, 6.)

As a whole, 9 out of 10 of the most important web application security weaknesses presented in OWASP T10 are directly related to any OAuth 2.0 implementation. These are only a part of the weaknesses and threats a web developer implementing OAuth 2.0 should be able to take into consideration and mitigate in order to implement OAuth 2.0 securely.

5 A Secure OAuth 2.0 Implementation

5.1 Security Features

As outlined by Meucci and Muller (2014, 13-15) there are multiple techniques to assessing and testing software: Manual inspections and reviews, source code review, threat modeling, and penetration testing. Manual inspections "are human reviews that typically test the security implications of people, policies, and processes" (Meucci and Muller 2014, 13). Source code review, on the other hand, "is the process of manually checking the source code of a web application for security issues" (Meucci and Muller 2014, 13). In threat modeling, a model of the threats facing the application is constructed and documented (Meucci and Muller 2014, 13). Penetration testing "is essentially [...] testing a running application remotely to find security vulnerabilities, without knowing the inner workings of the application itself" (Meucci and Muller 2014, 14). Manual inspections, source code reviews and penetration testing are used to assess the security of a single implementation, rather than to construct a general implementation model. Therefore, this study utilizes the threats outlined in previous work (Chapter 2), especially in the OAuth 2.0 threat model (Lodderstedt, McGloin, and Hunt 2013), in order to identify the necessary security features to mitigate these threats.

In this section only security features and choices that are crucial for the developer to understand while using the implementation model are considered. For a fuller picture of the security of OAuth 2.0, one should refer to previous work (Chapter 2).

5.1.1 Grant type

The first part of implementing Auth 2.0 involves choosing the grant types that are supported by the implementation. The four grant types defined by the OAuth 2.0 specification are authorization code, implicit, resource owner password credentials, and client credentials. As previously noted, there is also an extensibility mechanism which makes it possible to define additional types, two of which have already been specified by the IETF (cf. Campbell, Mortimore, and Jones 2016; Jones, Campbell, and Mortimore 2015). (Hardt 2012, 8.) The implementation model in this study aims to be easy to implement and understand without

compromising security. The fewer grant types are implemented the simpler the model will be and the less security considerations the developer has to face. Therefore, the model aims to include as few grant types as possible that securely support most use cases.

As the OAuth 2.0 implementation will be a web application API with end-users, support for resource owner password credentials and client credentials grant types are not a priority. When using resource owner password credentials to acquire an access token, the application uses the end-user's credentials (i.e. a combination of a password and a username). Thus the users have to share their credentials with the client application, which requires a high degree of trust between the resource owner and the client application. This is not a desirable flow, when the API is accessible to independent client application developers and should be used only with native client applications developed by the API-provider. The client credentials grant type can be used when the protected resource is owned by the client. It is often suitable for machine-to-machine based authentication. In our case, however, there are end-users involved as resource owners. (Hardt 2012, 9.)

Thus, the prominent grant types to consider for a web service API with end-users are authentication code and implicit. Out of these two, the authentication code grant has less security considerations and supports more use cases than the implicit grant type. The implicit grant type only supports access tokens and not refresh tokens. Authentication code grant type supports both. Implicit grant type is optimized for public clients, which are not able to maintain the confidentiality of access tokens. These clients are typically run directly in a web-browser using JavaScript. The authorization code grant type is, on the other hand, optimized for confidential clients, which are able to maintain the confidentiality of access and refresh tokens. In the implicit grant type access token is encoded into a redirection URI that the authorization server returns to the client in the end of the authorization flow. Therefore it may be exposed to the resource owner and other applications residing on the same device, e.g. though a Cross-site scripting (XSS) attack. When using the authorization code grant type the access token can be transmitted directly to the client. This eliminates the risk of exposing the access token when passing it through the resource owner's user-agent. The authentication code grant type also makes it possible to authenticate the client. (Hardt 2012, 8-9, 24, 31.)

Therefore, the implementation model will include only one grant type: authorization code.

The use of only one grant type simplifies the implementation process. It also removes the need for client authorization flow registrations (Sun and Beznosov 2012, 387). The implementation model will then include only two flows: The authorization code flow (cf. 5.1.2) and the flow for exchanging a refresh token to a new access token (cf. 5.1.6).

5.1.2 Authorization Code Flow

The authorization code flow illustrated in Figure 2 includes the following steps (Hardt 2012, 25):

(A) ”The client initiates the flow by directing the resource owner’s user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).”

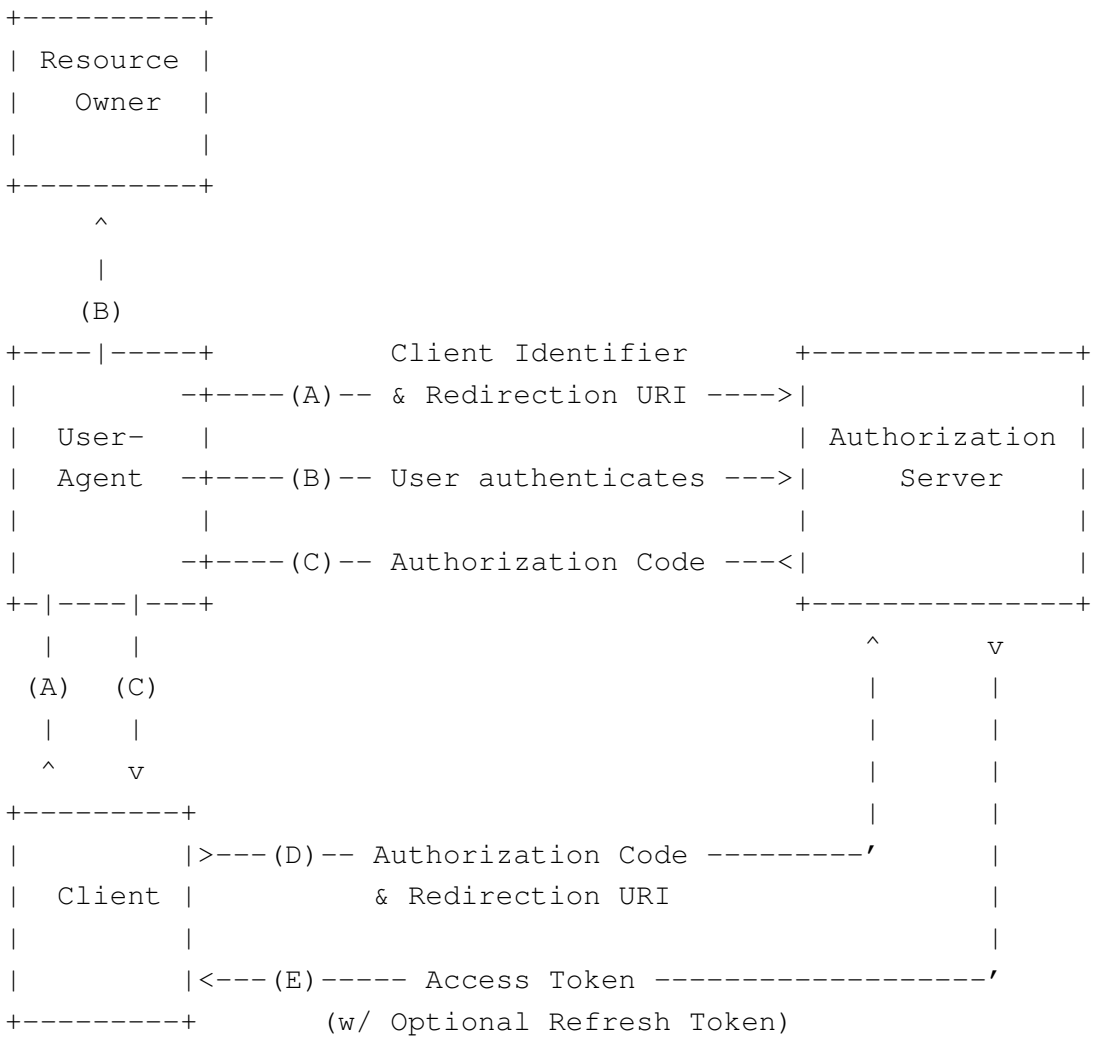
(B) ”The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client’s access request.”

(C) ”Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client earlier.”

(D) ”The client requests an access token from the authorization server’s token endpoint by including the authorization code received in the previous step. When making the request, the client authenticates with the authorization server. The client includes the redirection URI used to obtain the authorization code for verification.”

(E) ”The authorization server authenticates the client, validates the authorization code, and ensures that the redirection URI received matches the URI used to redirect the client in step (C). If valid, the authorization server responds back with an access token and, optionally, a refresh token.”

It is important to allow only a single use of the authorization code. This will protect the implementation from authentication code replay attacks. In addition the client application must send it’s client ID together with the access code. Although it does not provide any



Note: The lines illustrating steps (A), (B), and (C) are broken into two parts as they pass through the user-agent.

Figure 3. Authorization Code Flow (Hardt 2012, 24).

additional security for the protected resource, it will protect the client from an attack, where the attacker tries to substitute the client applications authentication code with it's own in order to gain access to protected resources.(Hardt 2012, 23, 56; Sun and Beznosov 2012, 387.) In addition, clients must be required to register redirection URIs in order to prevent open redirector attacks such as client impersonation (Bansal, Bhargavan, and Maffeis 2012, 257-258; Hardt 2012, 20; Sun and Beznosov 2012, 387).

Authorization codes must be short lived in order to mitigate damage that could be done, if someone hijacks an authorization code (Hardt 2012, 56). In this implementation model authorization codes will expire after 30 seconds.

5.1.3 Securing the traffic using Transport Layer Security (TLS)

The necessity of using TLS with OAuth 2.0 cannot be overly stressed. Implementing TLS requires additional resources and knowledge as the developer has to acquire a TLS certificate and have it configured on the server. In addition, TLS support and certificate checks should be implemented to the server and client applications. In this situation the developer might be tempted not to use TLS at all and use the unprotected HTTP-protocol or skip some TLS-features such as checking certificate authenticity (cf. examples by Hammer (2010b)).

However, as is clear from the specification and further verified by Xu, Niu, and Meng (2013) OAuth 2.0 has no built-in mechanism to protect the traffic between the client and the server. Therefore, without TLS all authorization information will be sent in the clear and is available to anyone eavesdropping the traffic. By simply following the specification and using TLS a number of security issues can be avoided (cf. Lodderstedt, McGloin, and Hunt 2013, 20, 23-24, 26, 38-39, 44, 46; Sun and Beznosov 2012, 382-383, 385, 388; Yang and Manoharan 2013, 275-276).

According to the specification, the authorization server must require TLS at the authorization and token endpoints as they involve transferring access and refresh tokens. However, it is not required at the redirection endpoint, because it would cause difficulties for many client developers. Instead, the risks related to possible authentication code eavesdropping are mitigated by having a very short expiration for the authorization codes and a mechanism for

revoking all tokens issued with the authorization code, if it is used more than once. (Hardt 2012, 20, 56, 59.)

Furthermore, the client must validate the authorization server's TLS certificate chains in order to prevent man-in-the-middle attacks using DNS hijacking. Otherwise the attacker might gain access to the data that is transferred including authorization codes and tokens. (Hardt 2012, 58; Jones and Hardt 2012, 12.)

5.1.4 Encoding requests using a Message Authentication Code (MAC)

The security of the implementation could be further improved using signed "proof tokens" instead of plain text bearer tokens (cf. subsection 5.1.5). The implementation could be done e.g. by using a Message Authentication Code (MAC). This would mitigate a large range of threats, and is advocated by Hammer (2010a) among others. However, the OAuth 2.0 MAC tokens grant type (cf. Richer et al. 2014) is still being drafted and implementing MAC token adds an additional level on complexity to the implementation, that is not absolutely necessary for the security of the implementation. (Jones and Hardt 2012, 11.)

Rather than encoding the access token information and preparing for access token theft, the current implementation will aim to protect all communications from eavesdropping and token theft. This is supported by the use of the authentication code grant type and strict usage of TLS in protecting the traffic.

5.1.5 Bearer Tokens

This implementation model, like many other OAuth 2.0 implementations, will implement access and refresh tokens as bearer tokens (cf. Jones and Hardt 2012). Bearer tokens are simple to implement and easy to use as any client who has the bearer token can use it as is, without authentication (Jones and Hardt 2012, 3).

However, the simplicity of bearer tokens causes them to be prone to a number of attacks. When manufacturing or modifying a token, an attacker may generate a bogus token or modify the contents of an existing token in order to gain unauthorized access to resources. An

attacker may also try to use an access token with a resource server that it is not meant for, which may mistakenly believe that the token is for it. (Jones and Hardt 2012, 10-11.)

Therefore it is important to safeguard bearer tokens from theft. In addition to properly implementing and using TLS, bearer tokens are not to be stored in cookies or passed in page URLs, because cookies are often sent out in the clear and page URLs may not be adequately secured in user-agents or servers and an attacker might be able to extract tokens from the history data, logs, or other unsecured locations. (Jones and Hardt 2012, 11, 13.)

Access tokens should also be short lived and scoped. This mitigates the damage that stolen bearer tokens can cause. In addition, if there are multiple resource servers, bearer tokens should include information on the audience (resource servers) they are meant for. (Jones and Hardt 2012, 13.) In this implementation model there will be only one resource server.

The OAuth 2.0 specification does not specify how bearer token authorization information should be stored. It states that an access "token may denote an identifier used to retrieve the authorization information or may self-contain the authorization information in a verifiable manner" (Hardt 2012, 10) and a refresh "token denotes an identifier used to retrieve the authorization information" (Hardt 2012, 10-11). In this implementation model access tokens will be alphanumerical strings that are longer than 27 characters (cf. subsection 5.1.8) referencing the actual authorization data. The information related to the tokens will be stored into a database. The token reference will be hashed. Both the authorization server and resource server will have access to the database, where authorization information is stored. In this way the developer does not have to implement encryption for token creation and does not have to worry about token manufacturing or modification. (Jones and Hardt 2012, 13.)

5.1.6 Refresh Token

Supporting refresh tokens is essential as it allows access tokens to have narrower scope and shorter lifetime without causing extra effort for end-users (Sun and Beznosov 2012, 387). A client may exchange a refresh token for a new access token and a refresh token, when an old access token expires without having to extra authorizations.

The flow for refreshing an expired access token illustrated in figure 4 includes the following

steps (Hardt 2012, 11-12):

(A) ”The client requests an access token by authenticating with the authorization server and presenting an authorization grant.”

(B) ”The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token and a refresh token.”

(C) ”The client makes a protected resource request to the resource server by presenting the access token.”

(D) ”The resource server validates the access token, and if valid, serves the request.”

(E) ”Steps (C) and (D) repeat until the access token expires. If the client knows the access token expired, it skips to step (G); otherwise, it makes another protected resource request.”

(F) ”Since the access token is invalid, the resource server returns an invalid token error.”

(G) ”The client requests a new access token by authenticating with the authorization server and presenting the refresh token.”

(H) ”The authorization server authenticates the client and validates the refresh token, and if valid, issues a new access token (and, optionally, a new refresh token).”

5.1.7 Client authentication

In order to be able to authenticate the clients, all clients should be issued client credentials that consist of a client ID and a client secret. These should be passed to the token endpoint in the request body using client_id and client_secret parameters. (Hardt 2012, 15-17.)

As opposed to the recommendation of the OAuth 2.0 specification, the implementation will not support the HTTP Basic authentication scheme (Hardt 2012, 16-17). It does not provide any extra security for the implementation. "The Basic authentication scheme is not a secure method of user authentication. – [I]t results in the essentially cleartext transmission of the user’s password over the physical network." (Franks et al. 1999, 19-20.)

Due to how modern web-browsers work, supporting the HTTP Basic authentication might even hinder the security of the client. This is true especially if the implementation is later extended to support the implicit grant type. Many web-browsers cache the Basic authentication

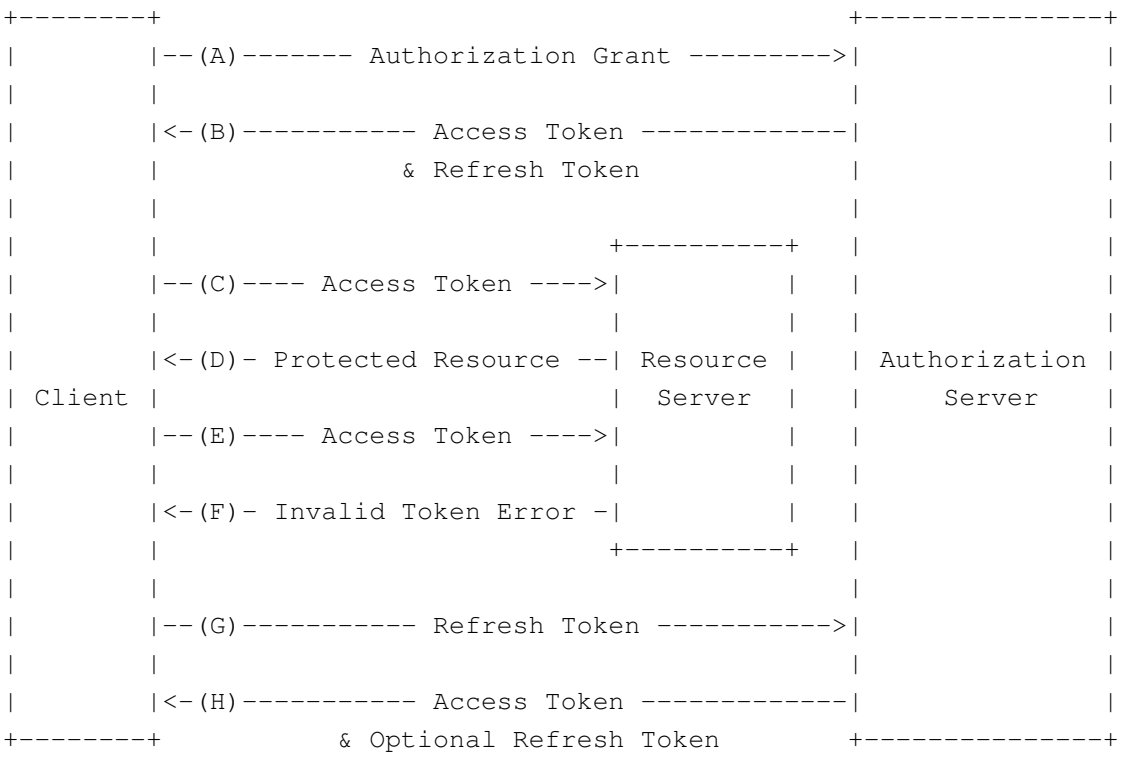


Figure 4. Refreshing an Expired Access Token (Hardt 2012, 11).

credentials automatically for the duration of the user session. They also often provide the user with the ability to save these credentials. This means that the credentials can be silently reused with any other request to the server, making Cross-site request forgery (CSRF) attacks possible. In addition, the credentials could be stolen by another user or process using the machine.

5.1.8 Generated credentials

In order to be able to prevent attackers from guessing generated credentials (access and refresh tokens, authorization codes, resource owner passwords, and client credentials) the probability of guessing the generated credentials has to be high enough. The specification requires that the probability of guessing them must be less than

$$2^{-128} \approx 2.9 * 10^{-39}$$

and should be less or equal to

$$2^{-160} \approx 6.8 * 10^{-49}$$

(Hardt 2012, 58.)

In order to achieve this without having to handle special characters (e.g. character ” that has to be escaped for e.g. JavaScript Object Notation (JSON)), the implementation will use and generate unique alphanumerical ASCII strings that have a minimum length of 27 characters. In this way probability of guessing the credentials will be greater or equal to

$$\frac{1}{62^{27}} \approx 4.0 * 10^{-49}$$

.

5.1.9 Brute force protection

Since our client authentication method involves a password (the client secret), the authorization server must protect the access and refresh token endpoint against brute force attacks (Hardt 2012, 17). In this implementation model the brute force protection is done by issuing credentials that have a very high entropy (cf. 5.1.8).

5.1.10 The ”State”-Parameter

In order to prevent CSRF attacks directed e.g. at the redirection URI a ”state”-parameter should be included in authorization requests. The ”state”-parameter is a value, which is used to maintain state the authorization request and callback. The client can use the value to validate the request by comparing it with the user-agent state. (Hardt 2012, 26; Lodderstedt, McGloin, and Hunt 2013, 32,39,66.).

5.1.11 End-user Security Considerations

A number of security features should be considered regarding end-users security.

First of all, all end-user sign-in points should be protected using TLS. This is is required by the OAuth 2.0 specification (Hardt 2012, 58) as it protects the end-user credentials from eavesdropping and reduces the risk of phishing attacks.

End-user password should not be stored as clear-text, but should be protected by encoding them. This is required by the OAuth 2.0 specification (Hardt 2012, 58), while it makes the credentials harder to use when stolen.

When end-users are authenticated, a session-cookie should be used instead of a normal cookie. This protects the cookie information from eavesdropping as the cookie will only have are reference to the session data stored on the server.

Related to the usage of the ”state”-parameter 5.1.10, end-user interactions should be protected using the the synchronizer token pattern. This protects the end-user interactions from CSRF-attacks. (Petefish, Sheridan, and Wichers 2016.)

5.1.12 Summary Table

Table 1 provides a summary of the implementation model’s security features and choices outlined in this section. These security features and choices will later be referenced as model requirements. The summary in the table acts as a security checklist for the developer implementing the model. As Bellovin (2008) notes, a checklist without a proper understanding

of the security issues may not benefit the developer and the security of the implementation. Therefore, a rationale, why the security feature or choice is important, is included for each requirement in the list.

Feature	**Requirement**	**Rationale**
OAuth 2.0 flows		
Grant type	Support as few as necessary.	Simpler to implement and less security considerations.
	Use only authentication code grant type.	Works well with web application use cases, more secure than implicit and supports both access and refresh tokens. Using only one grant type also removes the need for authorization flow registrations.
Redirection URI	Require clients to register redirection URIs	Prevent from open redirector attacks such as client impersonation.
Authorization codes	Make short lived (30 seconds) and single-use.	Required by the specification. Protects from authorization code replay attacks and mitigates damage in case the authorization code is hijacked.
	Revoke tokens issued with an authorization code, if the authorization code used more than once.	Mitigates damage in case the authorization code is hijacked and used by an attacker.
TLS	Require at the authorization and token endpoints.	Required by the specification. Protects tokens from eavesdropping.
	Validate TLS certificate chains.	Protects from DNS hijacking attacks.
Bearer tokens	Use bearer tokens.	Simple to implement and use by clients.

	Do not store bearer tokens to a cookie.	Prohibited by the bearer token specification. Protects the access token from eavesdropping, as cookies are sent in the clear.
	Do not pass bearer tokens in the page URL.	Protects the access token from eavesdropping.
	Limit scope and lifetime.	Mitigates damage in case access tokens are hijacked.
	Use references instead of encoding contents	Simplicity, protects from token manufacturing or modification.
Refresh tokens	Support refresh tokens	Improves security by a allowing a shorter lifetime of access tokens without causing extra effort for end-users.
	Allow only single use.	Required by the specification. Protects from replay attacks.
	Revoke tokens issued with a refresh token, if the refresh token used more than once.	Mitigates damage in case an refresh token is stolen and used by an attacker.
Generated credentials	Generated credentials length should be greater than 27 characters.	Prevents credential guessing attacks.
The state parameter	Use the state parameter when passing redirect URIs.	Prevent attacks targeting the redirection URI.
End-user security considerations		
End-user sign in	Use TLS at End-user sign-in points.	Required by the specification. Protects the credentials of the end-user from eavesdropping and reduces the risk of phishing attacks.

End-user password	Encode the password.	Required by the specification. Makes the credentials harder to use when stolen from DB.
End-user authentication	Use a session-cookie instead of a normal cookie.	Protects the cookie information from eavesdropping.
	Use the Synchronizer Token Pattern	Protects the site from CSRF attacks.

Table 1: Summary of the implementation model requirements.

5.2 Implementation

For the purpose of this study, the previously described OAuth 2.0 implementation model was implemented. The implementation provides a real life example of the implementation model and a platform for security testing.

The OAuth 2.0 server was implemented using PHP as a WordPress (*About WordPress* 2016) plugin for the WordPress REST API. PHP was chosen as the programming language, because it is one of the most popular programming languages (O'Grady 2016; Diakopoulos and Cass 2015; *TIOBE Index for July 2016* 2016). Accordingly, WordPress was chosen as a platform for the server, because it is a very popular open source project with a large user base and a legacy architecture. According to Protalinski (2015) by November 2015 one in four websites used WordPress. The legacy architecture and existing code base exposes the implementation to multiple developer challenges that are brought by the predefined architectural choices. For example, in the resulting implementation authorization endpoint actions are protected from CSRF attacks using the inbuilt security token system of WordPress (*WordPress Nonces* 2016), rather than a different Synchronizer Token Pattern implementation that would be unfamiliar to WordPress developers.

On the client side, The PHP League's OAuth 2.0 specification compliant client library (*OAuth 2.0 Client* 2016) was used with the provided GenericProvider class.

The source code for both the server and client is available on Github (cf. *WP REST API - OAuth 2.0 Server* 2016; *OAuth 2.0 Test Suite* 2016. The client code is included together with the test suite (cf. 5.3). The OAuth 2.0 WordPress plugin is licensed under GNU General Public License, version 2 (*GNU General Public License, version 2* 2016) or later and the client under the MIT license (*The MIT License (MIT)* 2016).

All the security features discussed above have been implemented. This section will go through all of them. Most of the implementation details are specified in detail by tests included in the test suite (cf. 5.3). Features that cannot be tested from the outside and therefore are not covered by the test suite, will be explained more in detail.

5.2.1 Grant type

The implemented authorization server only supports the authorization code grant type. Requests providing a different response_type parameter for the authorization endpoint receive a unsupported_response_type error.

5.2.2 Redirection URI

New clients can be registered by admin users from the WordPress admin panel. Registering the client requires a name, a description and a redirection URI. The name and description are shown to the user when they are asked to authorize the client. The redirection URI is used on the authorize endpoint to check, if the provided redirect_uri matches.

5.2.3 TLS

The authorization server endpoints answer requests only if SSL is enabled for the request.

The client checks TLS certificate chains. Because "PHP Streams are entirely insecure over SSL/TLS by default" (Padraic 2015), our client utilizes cURL for the requests. cURL performs peer SSL certificate verification by default using the server's CA certificate store (*SSL Certificate Verification* 2016).

5.2.4 Scope handling

Requested scopes can be sent to the authorization endpoint using the "scope" GET-parameter. The scopes correspond to WordPress capabilities, which is the WordPress equivalent for user permissions. If scope has not been set, it defaults to '*', which grants all capabilities that the user account has (now and in the future). Effectively, this means that the third-party using the access token is treated as if they would be the resource owner.

All of the requested scopes are listed in the authorization prompt with descriptions in order to inform the resource owner, which scopes they are about to grant access to.

5.2.5 Generated credentials

All the credentials that the server generates are 32 character alphanumerical strings. This includes the client IDs, client secrets, authorization codes, access tokens, and refresh tokens. The credentials are created using the wp_generate_password -function (*Code Reference, wp_generate_password* 2016). Internally, wp_generate_password uses the wp_rand -function (*Code Reference, wp_rand* 2016) to pick one each of the available characters in a random fashion. Both of these function can be overwritten by the site owner to provide environment specific credential generation or randomization functions.

5.2.6 Authorization codes

Authorization codes have a lifetime of 30 seconds. Authorization codes are revoked after use. Authorization code information is stored to the database and referenced using a hash of the authorization code. The authorization code itself is not stored to the database.

In addition, the authorization code hash is stored to the issued access and refresh token data, when it is saved to the database. If someone tries to use an authorization code a second time all access and refresh tokens issued using the authorization code will be queried and revoked.

Code	`JAftM5ZjK6m31aYmQzp3JyOYcNnZD9Ae`
Hash	`8532085691adc93cca82d4c7dfe90c15`

Stored data *(As a PHP key-value array)*	["hash" => "8532085691adc93cca82d4c7dfe90c15", "user_id" => 1, "redirect_uri" => "http://homestead.app/client/tests", "client_id" => "lqMx20rBbFDmf8Xq1ebtsyHWkqeOi2jD", "expires" => 1463743948, "scope" => "*"]

Table 2: Authorization code storage example.

5.2.7 Bearer tokens

The authorization server uses bearer tokens for access tokens. The authorization server stores the data of the issued access tokens to the database. This data is referenced using a hash from the original access token that is issued to the client. The access token itself is not stored to the database and works only as a reference to the data in the database.

Access tokens can have a limited scope, depending on what the client requests, and have a lifetime of one month.

Code	`8RzDoQqvwrFAqxnbcdi4sF9CM70Yvim1`
Hash	`7348d974d5b71137034b0749e9715bac`

Stored data *(Represented as a PHP key-value array)*	In `wp_posts` table (relevant fields) ["ID" => 1639 "post_author" => "1", // user ID "post_date" => "2016-05-20 12:02:47", // creation date ... "post_title" => "7348d974d5b71137034b0749e9715bac", // hash ... "post_type" => "oauth2_access_token", // token type ...] In `wp_postmeta` table ["client_id" => "lqMx20rBbFDmf8Xq1ebtsyHWkqeOi2jD", "expires" => "1466337767", "scope" => "*", "authorization_code" => "b76aef3354636b3ccb2c588ca798faf4", // if authorized with authorization code "refresh_token" => "as76aef3354636b3ccb2c588ca79823as" // if authorized with refresh token]

Table 3: Access token storage example.

5.2.8 Refresh tokens

All issued access tokens come with a refresh token. Refresh tokens are references and the data is stored the same way as with access tokens (cf. 5.2.7). Refresh tokens have a limited lifetime of one year.

In addition, the refresh token hash is stored to the issued access and refresh token data, when it is saved to the database. If someone tries to use a refresh token a second time all access and refresh tokens issued using the refresh token will be queried and revoked.

Code	`rRvRHhpjcpW8GqOeOeR8u6EkSNY46bRq`

Hash	`71e77469472a72f80dd860f3fc03df90`
Stored data *(Represented as a PHP key-value array)*	In `wp_posts` table ["ID" => 1640 "post_author" => "1", // user ID "post_date" => "2016-05-20 12:02:48", // creation time ... "post_title" => "7348d974d5b71137034b0749e9715bac", // hash ... "post_type" => "oauth2_refresh_token", // token type ...] In `wp_postmeta` table ["client_id" => "uf2epI1LIpN9", "expires" => "1495283159", "scope" => "*", "authorization_code" => "b76aef3354636b3ccb2c588ca798faf4", // if authorized with authorization code "refresh_token" => "as76aef3354636b3ccb2c588ca79823as", // if authorized with refresh token "access_token" => "ed532ea0c13a74ab0a1a333af60f33af"]

Table 4: Refresh token storage example.

5.2.9 The state parameter

The authorization endpoint adds the contents of the "state" GET-parameter to the redirect URI together with the authorization code.

5.2.10 End-user security

The end-user authenticates to the server using their WordPress login credentials. The end-user is redirected to the TLS-protected (https) version of the login form.

End-user passwords are hashed when they are stored to the database. The cookie that is set for logged-in users contains hashed information and a reference to the users session stored on the authorization server.

The authorization form is protected using a security token.

5.2.11 Implementation challenges

On the client side, the GenericProvider in PHP League's OAuth 2.0 client library (*OAuth 2.0 Client* 2016) fulfilled the implementation model for the most part. Only two of the model requirements had to be explicitly enforced. First, the library allows connecting to non-TLS endpoints. Therefore, it was essential to configure the endpoint URLs to use the https-protocol. In addition, the library does not impose usage of the "state"-parameter, which had to be implemented using PHP sessions.

Because WordPress was chosen as a platform for the server, a few implementation choices had to be made in order to fulfil the model requirements. A secure method for the token storage had to be chosen and a way to enable limited scope authentication had be implemented.

Authorization codes and tokens could have been stored into new database tables created for the plugin. However, using WordPress internal storage methods is preferred in WordPress development (*Creating Tables with Plugins* 2016). Apart from adding new database tables, WordPress offered two methods for storage. Custom post types, which are queryable by each property in the object, and options that is a key-value storage, where the whole object is stored in a serialized form. The refresh and access tokens needed to be queried based on their properties in order to revoke them, if the associated authorization code or refresh token is reused. Authorization codes needed only to be queried based on their key. Therefore, a non-public custom post type was chosen to be used as a storage for the tokens and authorization codes were stored as options. The tokens and authorization codes were referenced using their

hashes rather than the token or authorization code itself, which was especially important for their security, because when using WordPress' storage methods the credentials are stored among other WordPress data.

WordPress does not have in-built support for authentication where a user would be authenticated with a limited subset of their permissions or where a third-party would act on behalf of a user. By default, the user is assigned a set of capabilities based on their role. Access to resources is determined based on these capabilities and, in the case of content, the author of a resource. Resources have only a single author assigned, no user groups or multiple authors. As a result, in order for some of the capabilities to work, the author of the resource has to be the currently logged-in user. This limitation affects how limited scopes can be implemented. In the chosen implementation method, OAuth 2.0 scopes were chosen to be mapped as WordPress capabilities. When requesting for a limited scope, access token is permitted a subset of the user's capabilities. When issuing a request authenticated using an access token, the user in the token is logged-in. When WordPress checks for user capabilities during the request, a filter is assigned with the highest possible priority to strip out all capabilities that are not included in the token scope. This chosen implementation has some limitations. Most notably, WordPress cannot distinguish between the third-party client and the user itself. This can be an issue, when trying to track changes to content or trying to get information about the user capabilities through the REST API. In the former case the changes will seem to have been made by the user itself rather than the third-party client and in the latter the REST API would return the capabilities included in the token rather than the user's capabilities.

5.3 Tests

For the purpose of this study a simple test suite was implemented. The test suite aims to test a wide range of OAuth 2.0 implementations following our implementation model. Therefore, the test suite utilizes penetration testing, also known as black box testing, where the tests access the software only using it's public APIs. This limits the scope of security issues that our test suite can identify in the software. The tests have been designed to catch some of the most common implementation security issues and attack scenarios. (Meucci and Muller 2014, 13.).

The test suite itself is written in PHP and consists of two parts: The test client, which is used to test the authorization server and the client provider. The test server, which is used to test the client. Both parts provide the user with a visual overview of whether the tests passed or not. The source code of the test suite is available on Github (*OAuth 2.0 Test Suite* 2016) under the MIT license (*The MIT License (MIT)* 2016).

This section describes the tests included in the test suite in detail.

5.3.1 Overall flow

The overall flow tests ensure that the client is working correctly and checks the security features that can be revealed through a successful flow. client/oauth_works makes the tests from the client's perspective against a working OAuth 2.0 implementation whereas server/token_request tests the client connecting to the test server.

In the test client/oauth_works test cases 5,10, 12 and 15 ensure that the overall flow works fine, access and refresh tokens are generated, and resource server accepts the generated access tokens. Test cases 1-4 test if the token and authorization endpoints are protected using the HTTPS -protocol and using a SSL-certificate that can be peer-validated. Passing the test is essential to protect from OWASP T10 A6, "Sensitive Data Exposure". Test cases 6, 7, 11 and 13 ensure that the generated state parameters, authorization codes and tokens are long enough in order not to be easily guessed, which is required for the implementation to be protected from hijacking as per OWASP (2013, 8) T10 A2: "You may be vulnerable if: [...] Credentials can be guessed". Cases 7, 8 and 9 validate that the state parameter exists, it is long enough and that it equals the one in the authorization request in order to protect from a CSRF attack where an attacker might try to get a client to authorize with an authorization code belonging to the attacker as described by Lodderstedt, McGloin, and Hunt (2013, 32). Lastly, the test case 14 checks that the authorization server has provided an expiration for the access tokens. Limiting the lifetime of tokens reduces the harm that an attacker can cause if sensitive data is exposed (OWASP T10 A6).

The test server/token_request complements the client/oauth_works test by testing the client from the server side. These tests can be used with any OAuth 2.0 client that is able to connect

to the test server endpoints. The test cases 1 and 2 check for the existence and length of the state parameter in order to verify mitigation of CSRF attacks (OWASP T10 A8) similar to client/oauth_works cases 7, 8, and 9. Test case 3 ensures that the client is connected using HTTPS in order to mitigate sensitive data exposure (OWASP T10 A6).

Test	client/oauth_works
Description	The test goes through the overall OAuth 2.0 authorization code flow with multiple test cases.
Test case prefix	In the OAuth 2.0 authorization code flow:
Test case 1	
Description	authorize endpoint URL should use HTTPS
Success criteria	Success, if the schema of the provided URL is "https".
Rationale	Checks that the authorization endpoint against which the tests are run is SSL secured. Otherwise authorization endpoint SSL related tests might fail.
Threat	Eavesdropping, OWASP T10 A6
Test case 2	
Description	authorize endpoint should have a valid SSL certificate
Success criteria	Success, if cURL does not throw an error.
Rationale	Ensures that authorization endpoint certificate chain is valid.
Threat	Eavesdropping, OWASP T10 A6
Test case 3	
Description	token endpoint URL should use HTTPS
Success criteria	Success, if the schema of the provided URL is "https".
Rationale	Checks that the token endpoint against which the tests are run is SSL secured. Otherwise token endpoint SSL related tests might fail.
Threat	Eavesdropping, OWASP T10 A6

Test case 4	
Description	token endpoint should have a valid SSL certificate
Success criteria	Success, if cURL does not throw an error.
Rationale	Ensures that token endpoint certificate chain is valid.
Threat	Eavesdropping, OWASP T10 A6
Test case 5	
Description	should not throw an error
Success criteria	Success, if no error is thrown.
Rationale	Ensures that all tests have passed without errors.
Threat	None
Test case 6	
Description	authorize response should include a code query parameter longer than 26 chars
Success criteria	Success, if the "code" GET-parameter is longer than 26 characters.
Rationale	Ensures that the probability of guessing authorization codes is less or equal to 2^{-160}. The length is based on the assumption that the tokens include only alphanumerical characters (A-Z, a-Z and 0-9).
Threat	Credential guessing, OWASP T10 A2
Test case 7	
Description	authorize response should contain a non-empty state GET-parameter
Success criteria	Success, if the response "state" GET-parameter is non-empty.
Rationale	Checks that the authorization server returns the state parameters.
Threat	CSRF, OWASP T10 A8
Test case 8	
Description	authorize response should contain a state parameter longer than 26 chars

Success criteria	Success, if the response "state" GET-parameter is longer than 26 characters.
Rationale	Ensures that the probability of guessing the client provider created state-parameters is less or equal to 2^{-160}. The length is based on the assumption that the tokens include only alphanumerical characters (A-Z, a-Z and 0-9).
Threat	Credential guessing, CSRF, OWASP T10 A2 and A8
Test case 9	
Description	authorize response state should match the one that was sent
Success criteria	Success, if the response "state" GET-parameter matches the one sent in the authorization request.
Rationale	Checks that the authorization server returns the state parameters appropriately.
Threat	CSRF, OWASP T10 A2
Test case 10	
Description	token response should include an access token
Success criteria	Success, if the client provider can retrieve an access token from the response.
Rationale	Ensures that the flow works and access tokens are created.
Threat	None.
Test case 11	
Description	the access token should be longer than 26 chars
Success criteria	Success, if the access token is longer than 26 characters.
Rationale	Ensures that the probability of guessing generated access tokens is less or equal to 2^{-160}. The length is based on the assumption that the tokens include only alphanumerical characters (A-Z, a-Z and 0-9).
Threat	Credential guessing, OWASP T10 A2

Test case 12	
Description	token response should include a refresh token
Success criteria	Success, if the client provider can retrieve a refresh token from the response.
Rationale	Ensures that the flow works and refresh tokens are created.
Threat	None.
Test case 13	
Description	the refresh token should be longer than 26 chars
Success criteria	Success, if the refresh token is longer than 26 characters.
Rationale	Ensures that the probability of guessing generated refresh tokens is less or equal to 2^{-160}. The length is based on the assumption that the tokens include only alphanumerical characters (A-Z, a-Z and 0-9).
Threat	Credential guessing, OWASP T10 A2
Test case 14	
Description	token response should include an expiration
Success criteria	Success, if the client provider can retrieve the expiration from the response and the expiration is not empty.
Rationale	Ensures that the access tokens have a set expiration.
Threat	Eavesdropping, OWASP T10 A6
Test case 15	
Description	should be able to get the resource owner id
Success criteria	Success, if the owner id is greater than 0.
Rationale	Checks that the resource endpoint works with the access token from the authorization response.
Threat	None.

Table 5: Overall authorization code flow tests.

Test	server/token_request
Description	The test checks the client's token request for security features.
Test case prefix	Token endpoint requests should:
Test case 1	
Description	contain a non-empty state GET-parameter
Success criteria	Success, if the request "state" GET-parameter is not empty.
Rationale	Checks that the client utilizes the state parameter.
Threat	CSRF, OWASP T10 A8
Test case 2	
Description	contain a state parameter longer than 26 chars
Success criteria	Success, if the request "state" GET-parameter is longer than 26 chars.
Rationale	Ensures that the probability of guessing the state parameters that the client generates is less or equal to 2^{-160}. The length is based on the assumption that the tokens include only alphanumerical characters (A-Z, a-Z and 0-9).
Threat	CSRF, Credential guessing, OWASP T10 A6 and A8
Test case 3	
Description	be done over HTTPS
Success criteria	Success, if the request is done over HTTPS.
Rationale	Ensures that the client connects using TLS.
Threat	Eavesdropping, OWASP T10 A6

Table 6: Server side token request test.

5.3.2 Open redirect

Using redirects is an essential part of the authorization code flow. This makes it vulnerable to open redirect attacks, if the redirect_uri parameter is not validated before redirection. If utilized on the client side, an attacker could gain access to authorization "codes" or access tokens. If utilized against our authorization server, an attacker could use our authorization server to redirect the user to a malicious site in order to launch a phishing attack on our user. Our implementation model counters this threat by requiring that all clients register their allowed "redirect_uri" parameter and all redirect URIs are validated against these stored URIs. (Lodderstedt, McGloin, and Hunt 2013, 20, 22, 62).

The open redirector test cases in tests client/open_redirect_authorize_real_credential and client/open_redirect_authorize_fake_credentials test that redirect URIs are checked by the authorization server to mitigate OWASP T10 A10, "Unvalidated Redirects and Forwards".

Test	client/open_redirect_authorize_real_credential
Description	The test tries an open redirect attack on authorize -endpoint with real client credentials. The authorization request has a non-valid redirect_uri parameter with a real Client ID and Client Secret.
Test case prefix	When using an invalid Redirect URI with real credentials:
Test case 1	
Description	the authorization endpoint should not redirect.
Success criteria	Success, if the user returns to the page (manually) without an authorization code.
Rationale	Ensures that the authorization endpoint does not redirect to non-valid (possibly malicious) URLs.
Threat	Open redirection attack, OWASP T10 A10

Table 7: Open redirect authorization endpoint real credentials test.

Test	client/open_redirect_authorize_fake_credentials

Description	The test tries an open redirect attack on authorize -endpoint with fake client credentials. The authorization request has a non-valid redirect_uri parameter with a fake Client ID and Client Secret.
Test case prefix	When using an invalid Redirect URI with fake credentials:
Test case 1	
Description	the authorization endpoint should not redirect.
Success criteria	Success, if the user returns to the page (manually) without an authorization code.
Rationale	Ensures that the authorization endpoint does not redirect to non-valid (possibly malicious) URLs.
Threat	Open redirection attack, OWASP T10 A10

Table 8: Open redirect authorization endpoint fake credentials test.

5.3.3 Eavesdropping

Requests leveraging OAuth 2.0 contain sensitive information such as client credentials, access tokens, refresh tokens, or authorization codes (Lodderstedt, McGloin, and Hunt 2013, 23,25,44,46) and are therefore vulnerable to OWASP T10 A6, "Sensitive Data Exposure". An attacked might try to eavesdrop the requests in order gain access to this sensitive information during transportation. Such man-in-the-middle attacks can be countered by protecting the transmissions using TLS.

Therefore, in the eavesdropping test client/eavesdropping_no_tls test case 1 checks that the authorization server rejects non-HTTPS requests on authorization and token endpoints and the test case in test client/eavesdropping_invalid_certificate ensures that the client provider validates the authorization server's TLS certificate, as required by the specification. (Lodderstedt, McGloin, and Hunt 2013, 23,25,44,46.)

Test	client/eavesdropping_no_tls

Description	The test tries to connect to the authorization and token endpoints without SSL (using "http"-schema instead of "https").
Test case prefix	In order to prevent eavesdropping:
Test case 1	
Description	authorize endpoint should reject non-HTTPS requests
Success criteria	Success, if the request fails.
Rationale	Ensures that the authorization forces clients to use TLS when requesting authorization.
Threat	Eavesdropping, OWASP T10 A6
Test case 2	
Description	token endpoint should reject non-HTTPS requests
Success criteria	Success, if the request fails.
Rationale	Ensures that the authorization server forces clients to use TLS when requesting tokens.
Threat	Eavesdropping, OWASP T10 A6

Table 9: Eavesdropping no TLS test.

Test	client/eavesdropping_invalid_certificate
Description	Tests if the client provider will connect to a server with a non-valid certificate. This test is important as not all request implementations check certificates.
Test case prefix	In order to prevent eavesdropping:
Test case 1	
Description	client should check TLS certificate chains
Success criteria	Success, if client throws an error.

Rationale	Ensures that the client checks TLS certificate chains, which prevents MITM attacks in which the attacker might try to imitate the authorization server.
Threat	Eavesdropping, OWASP T10 A6

Table 10: Eavesdropping invalid certificate test.

5.3.4 CSRF attacks

The test cases 7, 8 and 9 in the test client/oauth_works and cases 1 and 2 in server/token_request validate the client providers and authorization servers handling of the state parameter in order to mitigate CSRF attacks. The test case in the test client/csrf_authorization_endpoint adds an additional test case, which ensures that an attacker cannot hijack an authorization URL and reuse it with a different client_id in order to create an authorization code for a malicious client.

Test	client/csrf_authorization_endpoint
Description	Tests if the authorization form enables CSRF tokens by redirecting the user to the authorization endpoint with authorization confirmed, but with an attacker client's ID and matching redirect URI. Requires that the user is logged in into the service provider.
Test case prefix	When trying to reuse an authorization URL with different client_id and matching redirect_uri:
Test case 1	
Description	the authorization endpoint should not redirect.
Success criteria	Success, if the user does not get redirected to the attacker redirect URI and user returns to the page (manually).
Rationale	Ensures that attackers cannot get authorization using a CSRF attack.
Threat	CSRF, OWASP T10 A8

Table 11: CSRF authorization endpoint test.

5.3.5 Scope handling

Improper scope handling could cause the application to be vulnerable to forced access as described by OWASP (2013, 13) T10 A7, "Missing Function Level Access Control". The scope handling tests aim to find common problems in scope handling. These tests are only valid for implementations that accept limited scope in their request, as our implementation model requires.

In the test client/scope_access_handling test cases 1, 3 and 4 ensure that the authorization server accepts limited scopes. Case 1 checks that access tokens requested with a limited scope can access protected resources within the scope. Case 3 ensures that refresh token with limited scopes can be used to get a new access tokens and case 4 that these tokens can access protected resources within the scope. Test cases 2 and 5 are actual security tests. Case 2 checks that the limited scope is properly attached to the access token and enforced by the resource server whereas case 5 executes the same test for the access token created using the refresh token received with the original access token. If either of the cases 2 or 5 fails the implementation is vulnerable to forced access as access tokens can be used to access resources outside of their requested scope.

Test client/invalid_scope_handling ensures that scopes are validated by providing a possibly malicious scope to the authorization request and expecting the server to reject it. If the authorization server accepts invalid scope, it could result in an injection attack (OWASP T10 A1) against authorization server through the scope parameter parser or into an XSS attack (OWASTP T10 A3) against the resource owner on the authorization page, where the requested scopes are displayed.

Finally, test client/refresh_scope_handling makes sure that refresh tokens can only get scopes within the original authorized scope. Case 1 tests that the authorization server rejects requests that have a wider scope than in the original request. Effectively, if the test fails, the authorization server is vulnerable to OWASP T10 A4 as clients can have unauthorized access to some or all of the resource owner's resources. Case 2 complements the case 1 by testing that the client can request a more limited scope than in the original authorization request. This is good for the security of the implementation, as it encourages clients to limit their access,

if they no longer require some of the scopes in the original request, because they can do so without requiring the resource owner to perform any additional tasks.

Test	client/scope_access_handling
Description	Tests that the authorization and resource server handles scopes properly. Necessary for other scope handling tests.
Test case prefix	Using a token with a limited scope:
Test case 1	
Description	should be able to access a protected resource in scope.
Success criteria	Success, if the request does not throw an error.
Rationale	Ensures that access tokens with a limited scope can be used to access protected resources.
Threat	None.
Test case 2	
Description	should not be able to access a protected resource not in scope.
Success criteria	Success, if the request throws an error.
Rationale	Ensures that access tokens with a limited scope can only be used to access protected resource within scope.
Threat	Unauthorized access, OWASP T10 A4
Test case 3	
Description	should be able to get a second access token with the refresh token.
Success criteria	Success, if the request does not throw an error.
Rationale	Ensures that refresh tokens with a limited scopes can be used to retrieve a new access token.
Threat	None.
Test case 4	

Description	the second access token should be able to access a protected resource in scope.
Success criteria	Success, if the request does not throw an error.
Rationale	Ensures that access tokens authorized using refresh tokens with a limited scope can be used to access protected resources.
Threat	None.
Test case 5	
Description	the second access token should not be able to access a protected resource not in scope.
Success criteria	Success, if the request throws an error.
Rationale	Ensures that access tokens authorized using refresh tokens with a limited scope can only be used to access protected resources within the original scope.
Threat	Unauthorized access, OWASP T10 A7

Table 12: Scope handling test.

Test	client/invalid_scope_handling
Description	Tests that the authorization server rejects authorization requests with an invalid scope.
Test case prefix	If trying to authorize with an invalid scope:
Test case 1	
Description	the authorization should fail.
Success criteria	Success, if the user is redirected back to the test page without an authorization code.
Rationale	Ensures that invalid scopes are properly handled by the authorization server. Invalid scopes can cause issues with later scope handling e.g. through errors in scope parsing.

Threat	Injection, XSS, OWASP T10 A1 and A3

Table 13: Invalid scope handling test.

Test	client/refresh_scope_handling
Description	Tests that the server handles refresh token scopes properly, when the token request has them included.
Test case prefix	When using a refresh token with a limited scope:
Test case 1	
Description	token request with scopes not included in refresh token scope should fail.
Success criteria	Success, if the request with scopes not in original scope fails.
Rationale	Ensures that the third-party client cannot get a wider scope than authorized in the original authorization request.
Threat	Unauthorized access, OWASP T10 A7
Test case 2	
Description	token request with scopes included in refresh token scope should succeed.
Success criteria	Success, if the request with a scope subset of the original scope succeeds.
Rationale	Ensures that the third-party client can limit their new access token scope, if necessary. Complements test case 1.
Threat	None.

Table 14: Refresh token scope handling test.

5.3.6 Replay attacks

In a replay attack, an attacker might use an authorization token or a refresh token that they have acquired through guessing (OWASP T10 A2) or stealing (OWASP T10 A6). This might happen before or after the client has been able to use them. In order to catch and mitigate these attacks authorization codes must be single-use and reuse must be tracked. The test case in test client/authorization_code_reuse checks that the authorization server rejects authorization codes, if they are being used a second time. The test case in test client/refresh_token_reuse does the same for refresh tokens.

After these tests have been executed, it should be ensured that the authorization server has logged the incidences, removed all tokens related to the replayed authorization code or refresh token and informed the administrators appropriately (logs, email or some other type of alert).

Test	client/authorization_code_reuse
Description	Obtains a valid authorization code and tries to use it twice.
Test case prefix	An authorization code:
Test case 1	
Description	should be single-use.
Success criteria	Success, if client throws an error when trying to use the authorization code for a second time.
Rationale	Mitigates the risk that an attacker could use a stolen authorization code or allows the resource owner to notice it (when using the authorization code fails).
Threat	Token replay and eavesdropping, OWASP T10 A2 and A6

Table 15: Authorization code replay attack test.

Test	client/refresh_token_reuse
Description	Obtains a valid refresh token and tries to use it twice.
Test case prefix	A refresh token:

Test case 1	
Description	should be single-use.
Success criteria	Success, if client throws an error when trying to use the refresh token for a second time.
Rationale	Mitigates the risk that an attacker could use a stolen refresh token or allows the resource owner to notice it (when using the refresh token fails).
Threat	Token replay and eavesdropping, OWASP T10 A2 and A6

Table 16: Refresh token replay attack test.

5.3.7 Brute-force attacks

Our tests ensure that brute-force attacks are mitigated by checking that the generated credentials are extremely hard to guess (test client/oauth_works test cases 6, 8, 11 and 13). Apart from ensuring that credentials are hard to guess, there are a multitude of ways to detect and protect from brute force attacks. One important measure is to limit the amount of guesses a single client can make. (*Blocking Brute Force Attacks* 2016.) Therefore, the test case in test client/bruteforce_client_credentials ensures that one client cannot create a multitude of requests to the server by issuing a 1000 asynchronous HTTP requests with different client IDs to the authorization endpoint and expects that the authorization server starts to reject these requests. This is a very broad test that aims to ensure that at least some brute force protection has been put into place.

Test	client/bruteforce_client_credentials
Description	Tests if the server will respond to a flood of requests from one client trying to brute-force a parameter by issuing 1000 asynchronous requests with different client IDs.
Test case prefix	When trying to brute-force client ID:

Test case 1	
Description	some of the 1000 requests should fail
Success criteria	Success, if client throws an error.
Rationale	Ensures that the server cannot be flooded with requests.
Threat	Credential guessing, OWASP T10 A5 and A6

Table 17: Brute-force client credentials test.

5.3.8 Test summary

Our test suite consists of multiple penetration tests that aim to ensure that the OAuth 2.0 specification and our implementation model security requirements have been been followed. The tests target a wide range of threats and features: The overall execution of the OAuth 2.0 flow, open redirects, eavesdropping, CRSF attacks, scope handling, and replay and brute-force attacks. The threats covered by the test suite include 9 out of the 10 OWASP T10 web application security weaknesses (A1, A2, A3, A4, A5, A6, A7, A8, A10). The OAuth 2.0 implementation described in the previous section 5.2 passes the tests (although brute force protection (5.3.7) may require additional policies to be put in place).

Passing these black box tests, however, does not guarantee the security of an OAuth 2.0 implementation. They only test that the particular attacks which the tests have been built for have been mitigated. (Meucci and Muller 2014, 13.). For proper assessment of an implementation's security, other software testing techniques must also be employed (cf. 5.1).

Nonetheless, the results of the test suite do provide a way to assess the security of an OAuth 2.0 implementation and to catch some of the more obvious vulnerabilities that can be observed from the outside.

6 Conclusions

In this study, a secure implementation model for web developers implementing OAuth 2.0 was constructed. The applicability of the implementation model presented in this study was verified by implementing it in a real life context on top of existing, widely used software. The created test suite was used to verify that the implementation meets the security goals of the implementation model.

Building the model was not simply a process of reading the previous studies and threat models on the subject and deciding what implementation choices are important. Although most of the model was constructed using previous literature, many aspects of the model and the test-suite were refined based on the problems faced while actually implementing the model. For example, although it was clear from the Web Security Weaknesses analysis (cf. 4.2) that the implementation would have to mitigate XSS attacks and by default all user input based output was escaped, at one point of development an XSS vulnerability related to how provided scope variables were handled crept into the code. This was caused by missing scope parameter validation. As a result, the Web Security Weaknesses analysis on OWASP T10 risk A3 was was extended to include a notion of the scope and additional tests added to the test suite (cf. 5.3.5).

On the surface, implementing a specification seems like a quite straight-forward process. However, as the OAuth 2.0 specification has multiple implementation options and optional features, making security related choices while trying to implement the overall functionality can be a quite tedious job. Based on experiences gathered while doing this research and based on the findings of previous research, the constructed implementation model offers a web developer implementing OAuth 2.0 a list of requirements that should make their implementation more secure than what it would be when utilizing only the specification. The constructed implementation model provides a starting point for developers to start implementing OAuth 2.0 or to evaluate their implementation choices against a tested model. It also provides a rationale for the security features and outlines the threats that they mitigate. Ideally, using the model should result in a more secure implementation in less time.

However, although checklists are widely employed in software security contexts, there is no guarantee that the model actually results in a better implementation. There is no way to ensure that all possible security threats are taken into consideration when building an implementation model such as the one presented in this study and that the checklist does not give the developer a false sense of security, which in turn would cause them to overlook some security issues. Still, the model mitigates threats that in previous studies have been identified to exist in multiple different implementations, such as the lack of the usage of the "state" parameter. The model has the definite benefit of making the developer aware of at least some of the most common security issues and ways to mitigate them.

Additionally, the model is restricted to a subset of the use cases that OAuth 2.0 can cover. For example, it can only be used with clients that are able to use the authorization code grant. This requires that "the the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server" (Hardt 2012, 31). In addition, the authorization grant is optimized for confidential clients. Thus, our implementation model is not well suited for public clients, including JavaScript applications running inside a browser.

The example PHP implementation of the implementation model provides a real life example for a developer to look at and reason how specific parts of the software have been implemented, while also verifying the usability of the implementation model and providing a platform for testing. However, even though PHP is one of the most popular programming languages and WordPress one of the single most used platforms, the example implementation might not be of much use for developers coming from other languages and platforms. This limits the usability of the implementation example.

The created test suite can be used as is to test new or existing OAuth 2.0 implementations. However, in order to use it to test a client provider the provider needs to be implement a specific PHP interface. This limits the clients that can be tested with the suite. Nevertheless, a subset of the tests work both against any valid OAuth 2.0 client and server implementations. Moreover, when the test suite cannot be used, it could still be used as a starting point for building unit and integration tests for other implementations. One of the biggest weaknesses of the test suite is that is limited to black box testing and can only test against the public

APIs of the server or client implementation. As a result, it does not provide visibility to the internal parts and code of the implementation that is being tested.

As a recommendation for further research, the implementation model presented in this study could be extended to include other OAuth 2.0 grant types, most notably the implicit grant. This would broaden the applicability of the implementation model to include also public clients. Also, the model could be modified to use MAC tokens instead of bearer tokens and research, how this affects the security choices included in the model. It might, for example, allow the implementation to be less strict about using TLS.

In addition, a study could be conducted in order to validate that using the checklist based implementation model presented in this study actually results in a more secure implementation. The study could, for example, be constructed to have two groups of developers with similar backgrounds that would implement the OAuth 2.0 authorization framework. One with the help of the implementation model and one without.

Also, more reference implementations of the implementation models could be created for other programming languages and platforms. This would help developers to understand the model as well as test the applicability of the model in different contexts.

Lastly, the test suite could be extended to cover more programming languages, use cases and threats. Different testing techniques could be utilized, such as code reviews, static code analysis or different types of black box testing. For example, client/invalid_scope_handling test could be easily extended to utilize a fuzzy tester in order to generate different types of invalid tokens in order to validate proper scope validation.

One of the notable themes in previous work was the difficulty of implementing OAuth 2.0 securely. The constructed model well illustrates this problem. Implementing OAuth 2.0 securely is no trivial task. Although the implementation itself is not very complex, the amount and detail of implementation choices affecting security makes the process complex. The developer choices that the constructed implementation model dictates range all the way from generated credentials length and form to the choice of supporting refresh tokens.

Bibliography

About WordPress. 2016. Retrieved July 7, 2016, from https://wordpress.org/about/.

Bansal, Chetan, Bhargavan, Karthikeyan, and Maffeis, Sergio. 2012. "Discovering Concrete Attacks on Website Authorization by Formal Analysis." In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th,* 247–262.

Bellovin, Steve. 2008. "Security by Checklist." *IEEE Security Privacy* 6, no. 2 (March): 88–88. ISSN: 1540-7993.

Blocking Brute Force Attacks. 2016. Retrieved July 7, 2016, from https://www.owasp.org/index.php/Blocking_Brute_Force_Attacks.

Campbell, B., Mortimore, C., and Jones, M. 2016. *Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants.* Retrieved July 7, 2016, from https://tools.ietf.org/html/rfc7522.

Campbell, B., Mortimore, C., Jones, M., and Goland, Y. 2015. *Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants.* Retrieved July 7, 2016, from https://tools.ietf.org/html/rfc7521.

Cantor, Scott, Kemp, John, Philpott, Rob, and Maler, Eve, eds. 2005. *Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0 – OASIS Standard, 15 March 2005.* Retrieved July 7, 2016, from http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.

Chari, S., Jutla, C., and Roy, A. 2011. *Universally composable security analysis of OAuth v2.0.* Retrieved July 7, 2016, from http://eprint.iacr.org/2011/526.pdf.

Cherrueau, Ronan-Alexandre, Douence, Rémi, Royer, Jean-Claude, Südholt, Mario, Oliveira, AndersonSantana de, Roudier, Yves, and Dell'Amico, Matteo. 2014. "Reference Monitors for Security and Interoperability in OAuth 2.0." In *Data Privacy Management and Autonomous Spontaneous Security,* edited by Joaquin Garcia-Alfaro, Georgios Lioudakis, Nora Cuppens-Boulahia, Simon Foley, and William M. Fitzgerald, 235–249. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Cisco. 2016. *Cisco Global Cloud Index: Forecast and Methodology, 2014–2019.* Retrieved July 7, 2016, from `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.pdf`.

Cluley, Graham. 2011. *Facebook flaw allowed websites to steal users' personal data without consent.* Retrieved July 7, 2016, from `http://nakedsecurity.sophos.com/2011/02/02/facebook-flaw-websites-steal-personal-data/`.

Code Reference, wp_generate_password. 2016. Retrieved July 7, 2016, from `https://developer.wordpress.org/reference/functions/wp_generate_password/`.

Code Reference, wp_rand. 2016. Retrieved July 7, 2016, from `https://developer.wordpress.org/reference/functions/wp_rand/`.

Creating Tables with Plugins. 2016. Retrieved July 7, 2016, from `https://codex.wordpress.org/Creating_Tables_with_Plugins`.

Crnkovic, GordanaDodig. 2010. "Constructive Research and Info-computational Knowledge Generation." In *Model-Based Reasoning in Science and Technology,* edited by Lorenzo Magnani, Walter Carnielli, and Claudio Pizzi, 314:359–380. Studies in Computational Intelligence. Springer Berlin Heidelberg.

Diakopoulos, Nick and Cass, Stephen. 2015. *Interactive: The Top Programming Languages 2015.* Retrieved July 7, 2016, from `http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2015`.

Differences between SAML 2.0 and 1.1. 2008. Retrieved July 7, 2016, from `http://saml.xml.org/differences-between-saml-2-0-and-1-1`.

Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. 1999. *HTTP Authentication: Basic and Digest Access Authentication.* Retrieved July 7, 2016, from https://www.ietf.org/rfc/rfc2617.txt.

Gantz, John and Reinsel, David. 2011. *Extracting Value from Chaos.* Retrieved July 7, 2016, from http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.

GNU General Public License, version 2. 2016. Retrieved July 7, 2016, from http://www.gnu.org/licenses/old-licenses/gpl-2.0.html.

Hammer, Eran. 2010a. *OAuth 2.0 (without Signatures) is Bad for the Web.* Retrieved July 7, 2016, from https://hueniverse.com/2010/09/15/oauth-2-0-without-signatures-is-bad-for-the-web/.

———. 2010b. *OAuth Bearer Tokens are a Terrible Idea.* Retrieved July 7, 2016, from https://hueniverse.com/2010/09/29/oauth-bearer-tokens-are-a-terrible-idea/.

———. 2012a. *OAuth 2.0 and the Road to Hell.* Retrieved July 7, 2016, from https://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/.

———. 2012b. *On Leaving OAuth.* Retrieved July 7, 2016, from https://hueniverse.com/2012/07/30/on-leaving-oauth/.

Hammer-Lahav, Eran, ed. 2010. *The OAuth 1.0 Protocol.* Retrieved July 7, 2016, from http://tools.ietf.org/html/rfc5849.

Hardt, Dick, ed. 2012. *The OAuth 2.0 Authorization Framework.* Retrieved July 7, 2016, from http://tools.ietf.org/html/rfc6749.

Jones, M., Campbell, B., and Mortimore, C. 2015. *JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants.* Retrieved July 7, 2016, from https://tools.ietf.org/html/rfc7523.

Jones, Michael and Hardt, Dick. 2012. *The OAuth 2.0 Authorization Framework: Bearer Token Usage.* Retrieved July 7, 2016, from http://tools.ietf.org/html/rfc6750.

Li, Wanpeng and Mitchell, Chris J. 2014. “Information Security: 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings.” Chap. Security Issues in OAuth 2.0 SSO Implementations, edited by Sherman S. M. Chow, Jan Camenisch, Lucas C. K. Hui, and Siu Ming Yiu, 529–541. Cham: Springer International Publishing.

Lindholm, Anna-Liisa. 2008. “A constructive study on creating core business relevant CREM strategy and performance measures.” *Facilities* 26 (7): 343–358. http://search.proquest.com/docview/219660132?accountid=11774.

Lodderstedt, Torsten, McGloin, Mark, and Hunt, Phil. 2013. *OAuth 2.0 Threat Model and Security Considerations.* Edited by Torsten Lodderstedt. Retrieved July 7, 2016, from http://tools.ietf.org/html/rfc6819.

Maler, Eve, Mishra, Prateek, and Philpott, Rob, eds. 2003. *Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V1.1 – OASIS Standard, 2 September 2003.* Retrieved July 7, 2016, from https://www.oasis-open.org/committees/download.php/3406/oasis-sstc-saml-core-1.1.pdf.

McGraw, Gary. 2012. “Software Security.” *Datenschutz und Datensicherheit - DuD* 36 (9): 662–665. ISSN: 1862-2607.

Meucci, Matteo and Muller, Andrew, eds. 2014. *OWASP Testing Guide, Version 4.0.* Retrieved July 7, 2016, from https://www.owasp.org/images/5/52/OWASP_Testing_Guide_v4.pdf.

OAuth 2.0. 2016. Retrieved July 7, 2016, from http://oauth.net/2/.

OAuth 2.0 Client. 2016. Retrieved July 7, 2016, from https://github.com/thephpleague/oauth2-client.

OAuth 2.0 SAML Bearer Assertion Flow. Retrieved July 7, 2016, from https://help.salesforce.com/HTViewHelpDoc?id=remoteaccess_oauth_SAML_bearer_flow.htm&language=en_US.

OAuth 2.0 Test Suite. 2016. Retrieved July 7, 2016, from https://github.com/apkoponen/oauth2-test-suite.

O'Grady, Stephen. 2016. *The RedMonk Programming Language Rankings: January 2016.* Retrieved July 7, 2016, from https://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/.

OWASP. 2013. *OWASP Top 10 - 2013 – Ten Most Critical Web Application Security Risks.* Retrieved July 7, 2016, from http://owasptop10.googlecode.com/files/OWASPTop10-2013.pdf.

Padraic, Brady. 2015. *Insufficient Transport Layer Security (HTTPS, TLS and SSL). Revision 328fe3aa.* Retrieved July 7, 2016, from http://phpsecurity.readthedocs.org/en/latest/Transport-Layer-Security-(HTTPS-SSL-and-TLS).html.

Pai, S., Sharma, Y., Kumar, S., Pai, R.M., and Singh, S. 2011. "Formal Verification of OAuth 2.0 Using Alloy Framework." In *Communication Systems and Network Technologies (CSNT), 2011 International Conference on,* 655–659. June.

Petefish, Paul, Sheridan, Eric, and Wichers, Dave, eds. 2016. *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet.* Retrieved July 7, 2016, from https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

Protalinski, Emil. 2015. *WordPress now powers 25% of the Web.* Edited by VentureBeat. Retrieved July 7, 2016, from http://venturebeat.com/2015/11/08/wordpress-now-powers-25-of-the-web/.

Richer, J., Mills, W., Tschofenig, H., and Hunt, P., eds. 2014. *OAuth 2.0 Message Authentication Code (MAC) Tokens.* Retrieved July 7, 2016, from http://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac-05. Work in progress.

SSL Certificate Verification. 2016. Retrieved July 7, 2016, from https://curl.haxx.se/docs/sslcerts.html.

Sun, San-Tsai and Beznosov, Konstantin. 2012. "The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems." In *Proceedings of the 2012 ACM conference on Computer and communications security,* 378–390. CCS '12. Raleigh, North Carolina, USA: ACM.

The MIT License (MIT). 2016. Retrieved July 7, 2016, from `https://opensource.org/licenses/MIT`.

TIOBE Index for July 2016. 2016. Retrieved July 7, 2016, from `http://www.tiobe.com/tiobe_index`.

WordPress Nonces. 2016. Retrieved July 7, 2016, from `https://codex.wordpress.org/WordPress_Nonces`.

WP REST API - OAuth 2.0 Server. 2016. Retrieved July 7, 2016, from `https://github.com/apkoponen/wp-rest-api-oauth2`.

Xu, Xingdong, Niu, Leyuan, and Meng, Bo. 2013. "Automatic Verification of Security Properties of OAuth 2.0 Protocol with Cryptoverif in Computational Model." *Information Technology Journal* 12 (12): 2273–2285.

Yang, Feng and Manoharan, Sathiamoorthy. 2013. "A security analysis of the OAuth protocol." In *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on,* 271–276.