Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Entwurf vom 29. November 2023

Muster des Deckblatts für Abschlussarbeiten

Masterarbeit

Intrusion detection for OAuth

vorgelegt von

Florian Nehmer

Matrikelnummer 6417446

Studiengang Informatik

MIN-Fakultät

Fachbereich Informatik

eingereicht am 06.01.2023

Betreuer: Pascal Wichmann, M. Sc. Informatik

Erstgutachter: Prof. Dr.-Ing. Hannes Federrath

Zweitgutachter: Pascal Wichmann, M. Sc. Informatik.

Aufgabenstellung

OAuth [RFC6749] is a widely used authentication protocol, which is typically used between multiple actors, such as different organizations. As authentication is at the core of application security, it is specifically essential to prevent attacks on the authentication.

The tasks of this thesis are as follows: Firstly, a systematic literature study should be performed on existing properties and attacks on the OAuth protocol or its implementations. Secondly, the thesis should design protection strategies for the threats that are not sufficiently solved in existing solutions. Two options for this step are (i) the utilization of anomaly-based intrusion detection for OAuth and (ii) specification-based intrusion detection for OAuth. Thirdly, the thesis should evaluate the security of the designed architecture and compare it to other solutions.

Zusammenfassung

Für die eilige Leserin bzw. den eiligen Leser sollen auf etwa einer halben, maximal einer Seite die wichtigsten Inhalte, Erkenntnisse, Neuerungen bzw. Ergebnisse der Arbeit beschrieben werden.

Durch eine solche Zusammenfassung (im Engl. auch Abstract genannt) am Anfang der Arbeit wird die Arbeit deutlich aufgewertet. Hier sollte vermittelt werden, warum man die Arbeit lesen sollte.

Inhaltsverzeichnis

1 Introduction

1.1 Motivation

Write Motivation

1.2 Research Question

Write down Research Question

1.3 Outline

Decide if Outline is necessary

2 Fundamental Knowledge

Write Introduction to Fundamental knowledge chapter

2.1 OAuth 2.0 protocol

The Open Authorization 2.0 protocol nowadays often referred to as the *OAuth* protocol, is an authorization framework, that allows third-party applications to gain limited access to resources in a different location on behalf of the party, that owns these resources. For many users of the Internet, it is in practice the protocol behind the "*Sign in with ...*" button. The current standard, first defined in 2012 in RFC6749, is already the successor of the OAuth 1.0 standard, which was officially published in 2010 by the IETF in RFC5849 [Ham10]. In the meantime, several extensions for the protocol were published as standards and technical reports. These extensions include new functionalities for the protocol e.g. the ability to use the protocol with devices like smart TVs and printers [Den+19] or documents, which describe several security considerations when implementing the protocol in practice [Lod+20]. As a whole, the OAuth working group of the Internet Engineering Task Force (IETF) submitted a total of 30 Request for Comments (RFCs) and 16 active drafts, from which 7 are active individual drafts. Table X shows a complete list of all OAuth 2.0. related IETF submissions by the OAuth working group.

2.1.1 Involved parties

Because the OAuth protocol is very diverse and complex, as it is a whole authorization framework it makes sense to narrow it down to its core features. Starting with the involved parties in the protocol. In general there are four parties involved in the most common OAuth protocol modes:

- Resource owner: The resource owner is the entity that owns or is allowed to manage protected resources. The resource owner might grant access to these resources.
- Resource server: The resource server is the server, where the protected resources are stored. It can accept or decline authorization tokens, which it receives from the client.
- Client: The client is an entity, which makes requests to get access to the protected resources, on behalf of the resource owner.
- Authorization server: The server, that manages access to the protected data. It issues access tokens to the client after successful authentication of the resource owner.

Depending on the protocol mode, these four parties or in some cases three parties exchange different messages in variable ways. In general, there are two different types of messages, which are explained in the next section.

2.1.2 Front-channel and Back-channel messages

Regarding security considerations messages of the OAuth framework can be categorized into two main categories, *front-channel* and *back-channel*. As the protocol is mostly used in the application layer using HTTP and TLS, *front-channel* means that the message is transported via the *Request-URI* [Fie99, Sec. 5.1.2] e.g. by using query parameters. *Back-channel* means on the other hand that the message is transported via the HTTP message body. In other words, back-channel messages are transported in one TCP connection, between caller and receiver, whereas front-channel messages use redirects. [Bel+22, p. 338].

2.1.3 Grant Types

The protocol flow is dependent on the protocol mode. In the case of OAuth the different protocol modes are called *Grant types*, as the modes differ on how the authorization is granted to the resource owner via the client.

Authorization Code Grant

According to a recent study, the authorization code grant mode of OAuth is the most used protocol mode for OAuth on the internet [PPJ22, Table1]. It is offered by more than 90% of common identity providers.

In this mode illustrated in Figure 2.1, a resource owner aiming to access protected data via a REST API utilizes a user agent, in this case the web browser, to interact with a client application that executes API requests through the user agent. The process begins as the client redirects the user agent to the authorization server through the front channel. The message contains a client ID, a redirect URL and a state. The client ID is a unique identifier of the protected resource. The redirect URI is the location the user agent is redirected to after authentication. The state can hold any string value and is most commonly used for CSRF tokens. The authorization server now asks the resource owner for authentication. If the resource owner is authenticated and is allowed to access the desired protected resources, the user agent gets redirected back using the redirect URI. This means that the message is again sent via the front channel and contains now an authorization code and the state. Using the valid authorization code and the state, the client proceeds to request an access token via the back channel at the authorization server. With the acquired access token the client follows with the request of the desired protected resource at the resource server.

Implicit Grant

The Implicit Grant was introduced at a time when there were no mechanisms like Cross-Origin resource sharing implemented in browsers, to share content from different domains. It is a predecessor of the authorization code flow and works similarly with the difference of leaving out the exchanging of the authorization code step as visualized in figure 2.2. Instead, the access token is sent via the front channel directly from the authorization server to the client. This leaves open more attack vectors for example by utilizing the browser history or by simplifying access

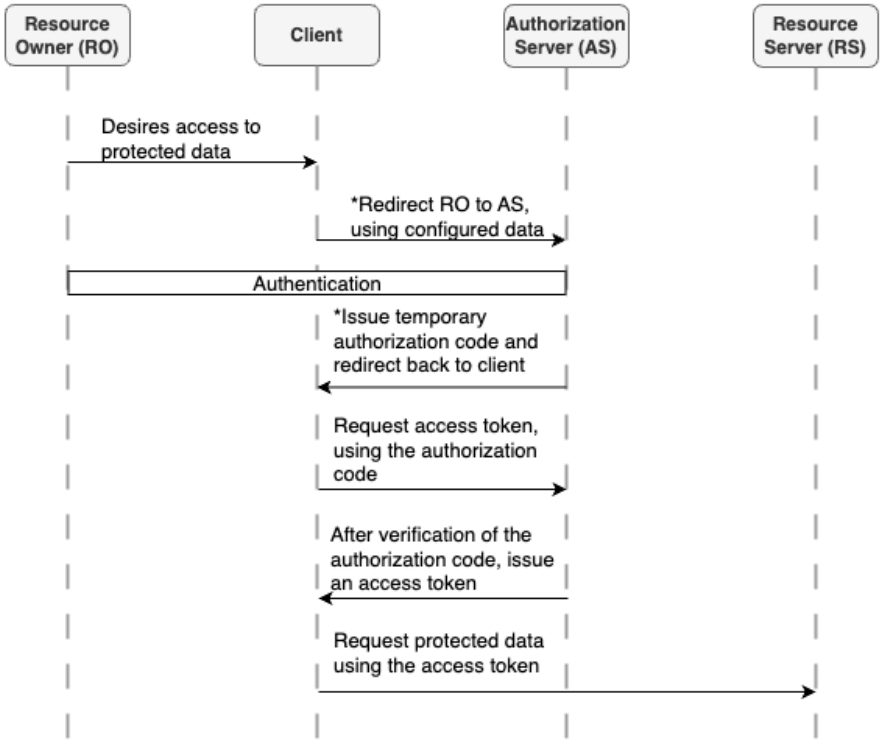

Abbildung 2.1: Authorization Code Grant without any extensions

token injection [Lod+20]. The Implicit Grant is officially deprecated, but still has its relevance, as it is still offered by 37% of common identity providers [PPJ22].

Resource Owner Password Credentials Grant

This grant type is special in the way that the client is providing its authentication credentials for the authorization provider to the resource provider instead, as illustrated in figure 2.3. The resource provider than uses the credentials to retrieve authorization from the authorization provider. This grant type is only feasible for the scenario, that the resource provider is trusted completely [Har12, Sec. 4.3.].

Client Credentials Grant

The client credentials grant must only be used by confidential clients interacting with each other. This means the clients have the ability to securely store a secret, which is only accessible by themselves. A common use-case for this scenario would be machine-to-machine interactions. This grant type is meant for clients to access their own resources as is the case in micro-service architectures. As depicted in figure 2.4 the client authenticates at the authorization server with its client secret and receives an access token, to authenticate at the resource provider. This means that the resource provider does not need to verify client secrets, but instead only needs the capability to verify access tokens. This fact is useful for practical reasons, as a resource provider could reuse the implementation of access tokens for other grant types it is offering. [Har12, Sec. 4.4.]

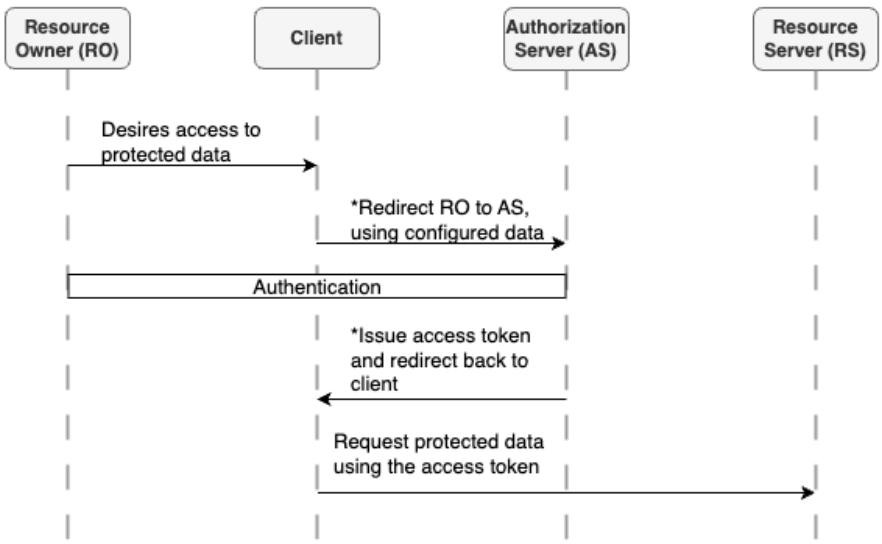

Abbildung 2.2: Implicit Grant

Device Authorization Grant

Introduced in RFC8628 the device authorization grant is meant to be used for devices, that lack a user agent like a web browser or do not offer a convenient way of entering text [Den+19]. In this grant type the client is not mainly interacting through a user agent like a web browser anymore, but instead is using a device authorization endpoint at authorization provider directly to initiate an authorization request. The client, then instructs the user to open a webpage on a secondary device to complete the authorization process using a displayed code for verification of the session. This OAuth flow still requires the involved devices to use HTTP for communication, which in general is not a feasible solution for many IoT-devices. A solution for the popular IoT-protocol CoAP is proposed by Chung et al.

Other grant types

There are several more grant types, which got introduced to the OAuth standard over time, which are listed below, but are out of scope for this work:

- Refresh Token Grant
- JWT Bearer Grant
- UMA Grant
- SAML 2.0 Bearer Grant
- Token Exchange Grant

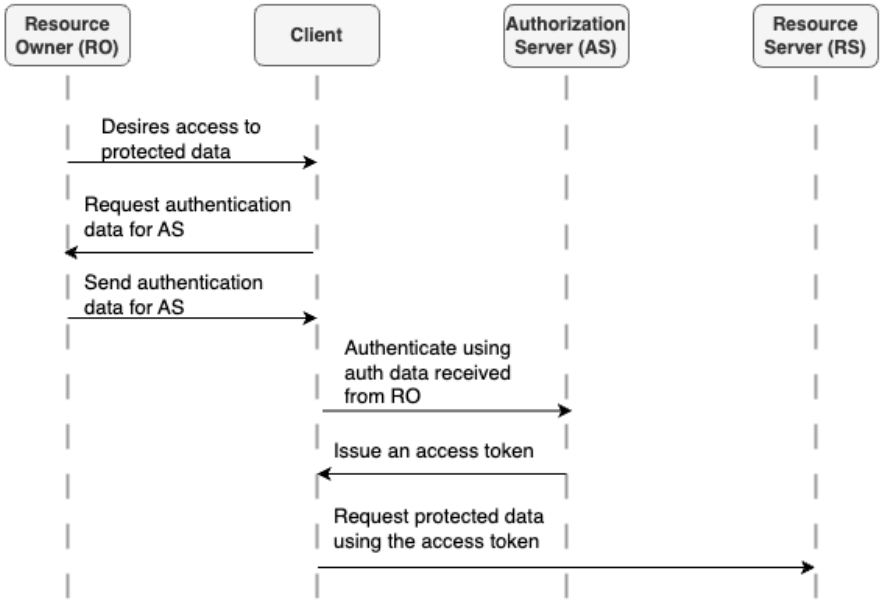

Abbildung 2.3: Resource Owner Password Credentials Grant

2.1.4 Open ID Connect

Open ID Connect (OIDC) is a layer on top of OAuth 2.0, introduced in 2014 by a combination of researchers and companies like Google, Microsoft and Salesforce, which form the OpenID group. The main focus of Open ID Connect is authentication, which means identifying a user but not authorizing the user. OIDC flows use a particular OAuth scope, which is called öpenid", an extra token, the ÏD token", which is a JWT token containing the identifying information about the user, which are called claims. The claims can be used to retrieve more information about the user at the OpenID provider, or the information can be directly part of the initial ID token, so OpenID adds the authentication to the already provided Authorization capability of OAuth. For OIDC, the same grants are used as for OAuth, but in the context of OIDC, they are often referred to as flows [LM16] [Sak+14].

2.1.5 The Future: OAuth 2.1

OAuth 2.1 will be the next version of OAuth. There is currently one draft available at the IETF [HPL23]. The main intent of this draft is to consolidate the various extensions to OAuth introduced in the last years. It simplifies the core document for OAuth 2.0 and omits outdated features because of new browser capabilities, which came over the last years and allow for more secure interactions. Below is a list of significant changes:

- The Implicit Grant gets omitted
- The Resource Owner Password Credentials Grant gets omitted
- PKCE is required for the Authorization Code Grant
- Redirect URIs must be compared with exact string matching, so pattern matching is completely disallowed

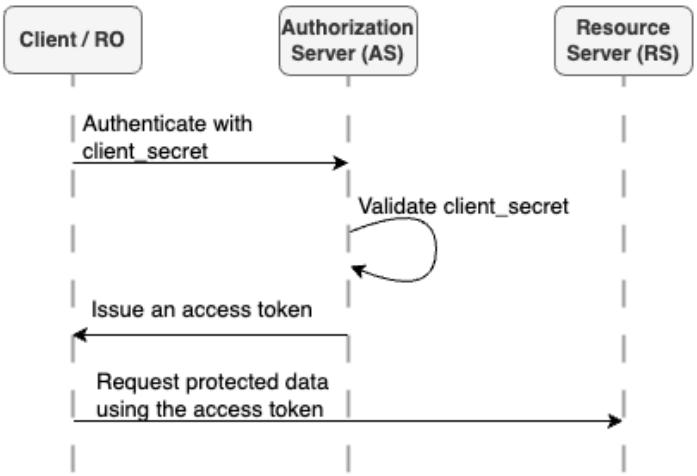

Abbildung 2.4: Client Credentials Grant

- Access tokens must not be transported in a Request-URI in any case (which also makes the Implicit Grant impossible

In conclusion, the draft for OAuth 2.1 picks valuable features of existing standards for OAuth to require them and omits outdated features, intending, on the one hand, to simplify the OAuth landscape and, on the other hand, get rid of insecure features of the past.

2.2 Intrusion Detection System

Fundamental Knowledge: Make clearer that the long definitions use one source

An intrusion is any malicious behavior with the intention to damage or take control of an information system. All primary protection goals of information security, like confidentiality, integrity and availability, can be the target of an intrusion. A method to harden systems against intrusions is using intrusion detection systems (IDS), which monitor all traffic, events or actions of a system to detect when malicious actions are executed so the system's owner, the administrator or the system itself can take immediate action on intrusion attempts. If the intrusion detection system takes action to prevent intrusions, it is called an *Intrusion Detection and Prevention System (IDPS)* [SM10]. There are several taxonomies for intrusion detection system types. For example, [Lia+13] categorized IDS into five categories: Statistics-based, Pattern-based, Rule-based, State-based, and Heuristic-based, referring to the methodology of detection the IDS is using. [Khr+19] build on top of that approach to further specify an overall taxonomy for IDS, which is used in this work to define Intrusion Detection Systems.

2.2.1 Distinction by input data

Generally, when categorizing Intrusion Detection Systems by environment or input data, there are two types to distinguish them. *Host-based Intrusion Detection Systems (HIDS)* and *Network-based Intrusion Detection Systems (NIDS)*.

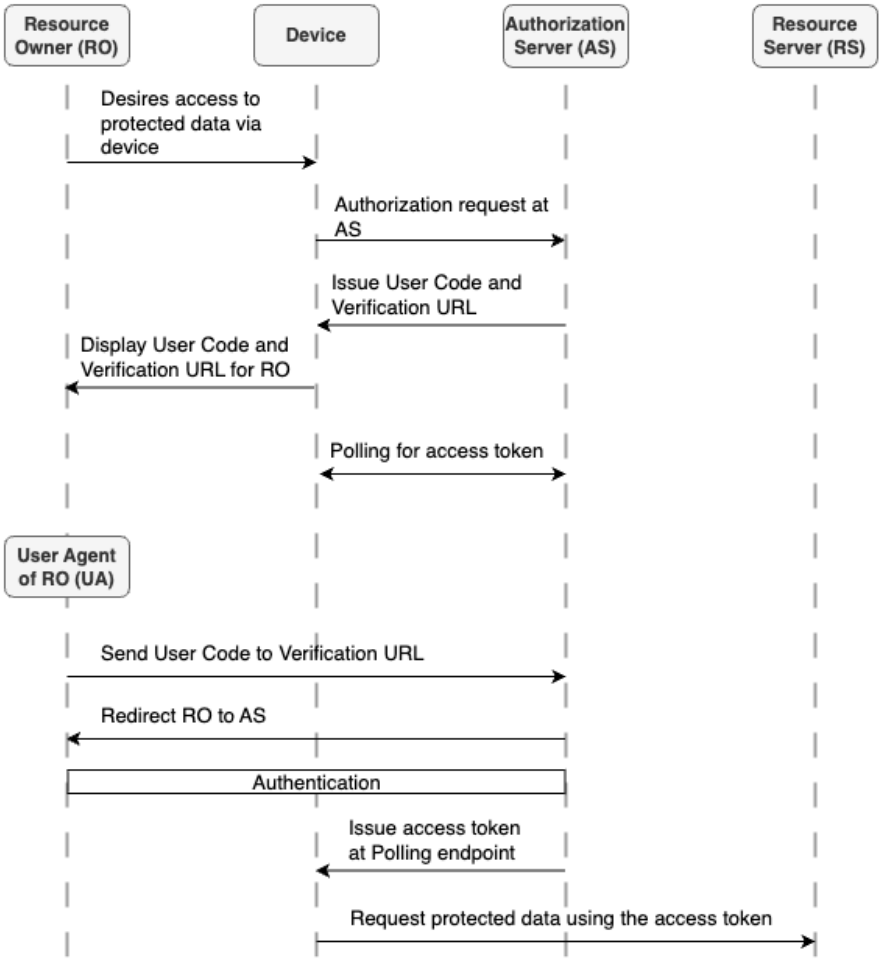

Abbildung 2.5: Device Authorization Grant

Host-based Intrusion Detection System

HIDS monitor all data concerning a host system, like operating system events, host firewall logs or application-specific logs. They can detect specific attacks on the system level without using network logs.

Network Intrusion Detection System (NIDS)

NIDS monitor network traffic, which is acquired by packet capture tools and other network data sources. One challenge of Network Intrusion Detection Systems is often the large amount of data that needs to be analyzed in high-bandwidth networks, which demands high computing capabilities.

2.2.2 Distinction by detection characteristic

When distinguishing types of intrusion detection systems by the way they operate, there are two categories: Signature-based (SIDS) and Anomaly-based (AIDS).

Signature-based Intrusion Detection System

Signature-based IDS utilize databases of fingerprints of already known attacks. These databases are called knowledge databases. A fingerprint could be, for example, a hash value of an executable malware file. A HIDS could compare any file an operating system executes against the knowledge database to detect the specific malware. The same concept also works for NIDS, e.g., when scanning mail attachments transferred via SMTP. The main advantage of SIDS is that they rarely produce false positive alerts when identifying intrusions. Their downside is that they cannot detect unknown attacks.

Anomaly-based Intrusion Detection System

The core concept for Anomaly-based IDS is to differentiate between usual and unusual behaviour. Unusual behaviour is identified as an intrusion. There is a variety of techniques that AIDS can use to achieve the goal of differentiating typical and malicious behaviour. One possibility is creating a statistical model over the data and filtering out events with a low probability. Another approach is to use machine learning techniques. Unsupervised learning methods, like clustering, on the one hand, are very similar to the statistical approach because the goal is again to sort out rare occurrences in small clusters. Supervised learning methods, on the other hand, create a model of usual behaviour in a training phase with labelled data to test any new input of unknown data if it is classified as malicious. An additional way for Anomaly-based IDS is the knowledge-based method. With this method, knowledge is applied to the detection in the form of rules to identify any behaviour that breaks those rules as an intrusion. These rules can stem from knowledge about a network protocol or any other system behaviour.

2.2.3 zeek IDS

Fundemental Knowledge: Write zeek section

2.3 Algorithms

Describe the algorithms used in the experiments

3 Related Work

Write Introduction to Related Work chapter

3.1 Security Analysis of OAuth

[FKS16]

Realted Work: Describe Security Analysis of OAuth

3.2 Machine Learning approach to detect OAuth vulnerabilities

[MP22]

Related Work: Describe approach to detect OAuth vulnerabilities

3.3 Another One

Related Work: Decide on which 1-2 more articles to mention here as-well

4 OAuth Security

Write Introduction to OAuth Security chapter

4.1 Threats and Vulnerabilities

4.1.1 Insufficient Redirect URI Validation [Lod+20] [Wan+19]

Authorization Servers need to whitelist redirection URLs in order to make sure, that an attacker cannot craft a hyperlink, which leads to the victim initiating an OAuth flow and sending the authorization code or token to an attacker-controlled domain. Some authorization servers may allow the usage of patterns in order to allow several domains at once. As well as the absence of any sort of whitelist mechanism even a pattern-matching functionality could lead to security problems. Among the possibility that a user is entering patterns that are too broad and allow the usage of unintended redirect URLs, the attack surface includes issues with the URL parsing implemented by the authorization server as shown by Wang et al. [Wan+19]. They presented several techniques to trick the parser into accepting unintended domain names, like using squared brackets for IPv6 parsing or the *Evil Slash Trick*, where the parser does not treat a forward slash as a path separator, while modern browsers do. Depending on the OAuth grant type in use this vulnerability leads to different possibilities to exploit it.

Authorization Code Flow

- The attacker uses techniques like phishing to make its victim open an attacker-controlled webpage, which initiates an OAuth flow with the vulnerable authorization server.
- The request is crafted with a valid client ID (which is public information), “code” as response type and a malicious redirect URI, which leads to an attacker-controlled server again.
- If the user logs in at the authorization server, the authorization code now gets transmitted to the attacker’s webpage, via the redirect URI.
- The attacker page can now use the received authorization code, to retrieve a token

Implicit Flow

Maybe write down open redirection with the implicit flow here regarding redirect_uri check circumvention

4.1.2 Credential Leakage via Referer Headers

The Referer HTTP header is a potential attack surface. It can be utilized by a malicious actor to capture query parameters, which are sent via the front channel, like the state and the authorization code. The authorization code may be used to redeem an access token before the victim retrieves it and the state parameter oftentimes includes a CSRF token, which could potentially open up vulnerabilities in other parts of the application as explained by Fett et al [FKS16].

OAuth Client

If a client renders third-party content, like advertisements in iframes or images, before redeeming the authorization code for an access token an attacker, who places these advertisements or images can capture the code via the referer header and redeem it for an access token.

Authorization Server

At the authorization server, the state parameter could be leaked via the Referer header, when 3rd party images or advertisements are being rendered on the page. This may be an issue when the state contains a CSRF token as explained by Fett at al [FKS16].

4.1.3 Credential Leakage via History Logs

OAuth potentially transports sensitive data via the request-URI, like the access token, as is the case when the implicit grant is used or if other grant types optionally allow the transportation of access tokens or authorization codes via URI parameters. Therefore, a person accessing the user's browser can extract this sensitive data and try to replay it. The same threat is present when a logging server is present, for example, in a corporate network [Lod+20]. Research about browser history security focuses mainly on accessing information about the victim's history by comparing cache timings if a page was visited [BPM]. This type of threat is irrelevant in the OAuth context because an attacker would need to guess the access token, which the attacker could endeavor outside the browser history as well. However, recent studies on the security of browser extensions show that malicious browser extensions could access the browser history, or data could be leaked by utilizing vulnerable browser extensions [EPS22].

Countermeasures

- Authorization Code - Access Token

4.1.4 Mix-Up Attack

In this attack, at least two authorization providers are involved. The target AP and the attacker AP. The OAuth standard allows the resource provider to interact with multiple authorization providers, one of which could be malicious, so the security for such interactions must also be provided. The attack is feasible for the implicit and authorization code grants and works similarly for both. Another precondition is that the resource owner registers the same redirect URI at both authorization providers, which is typical for Open ID Connect dynamic client registration [HKW23]. With these preconditions present, the attacker now waits until the target initializes an OAuth flow with the attacker AP. The attacker then intercepts the initialization request and exchanges the target AP with its attacker AP. When the target client gets redirected to the attacker AP, the target gets immediately redirected back to the target AP for authentication. At the same time, the client ID in the query parameters of the redirection URL gets replaced with the one registered at the target AP. Suppose the target user authenticates because it did not detect that it intended to authenticate at another AP. In that case, an authorization code gets issued to the client when the authorization code grant used. The client then proceeds to try to redeem an access token at the attacker AP, as the client still thinks that it initiated an OAuth flow with the attacker AP. The attacker can now use the received authorization code to redeem an access token at the target AP. [FKS16]

4.1.5 Authorization Code Injection [PPJ22]

The precondition for an authorization code injection is that an attacker has successfully stolen an authorization code. This can be accomplished in various ways for example by tricking the user into installing a malicious browser extension, using other vulnerabilities in a web app like open redirections, or abusing proxy auto-configuration files [PPJ22].

In the case, that the client is using the authorization code flow the attacker can use the stolen authorization code to fetch an access token before the client does.

Countermeasures

There are several ways to mitigate the risk of authorization code injection. For example, if an authorization code got used twice in the case it got stolen and the attacker, as well as the client, tried to redeem a token, every token that was redeemed with this authorization code should get invalidated. Another method is the usage of PKCE to make sure the OAuth flow is only used, between the two legitimate parties.

4.1.6 Access Token Injection

This kind of attack describes the process of an attacker using a stolen access token in a legitimate authentication flow, to impersonate the client. If the implicit flow is available, the attacker can now start a new flow and simply replace the access token in the authorization servers' response. This will circumvent any CSRF protection, as there is no difference to a non-compromised flow. [Lod+20]

4.1.7 Cross Site Request Forgery

This type of attack, often referred to by its abbreviation “CSRF, is about the attacker executing a request in the name of the user, by tricking the user into executing requests for the attacker including all required authentication, or authorization information. The default OAuth protocol does not include protection mechanisms against this type of attack.

4.1.8 PKCE Downgrade Attacks

If an authorization server is not implemented to require PKCE for all its flows, it is susceptible to being vulnerable to PKCE downgrade attacks [PPJ22]. Even if it is documented otherwise, attackers might try to omit PKCE parameters, as the current OAuth 2.0 standard does not require the usage of the PKCE extension [Har12].

4.1.9 Access Token Leakage at the Ressource Server

In the scenario that clients can dynamically connect to resource servers at runtime, as is the case in mail or banking applications, an attacker could create a malicious resource server and trick the user into sending valid access tokens for the target data to the malicious resource server. It could also be the case that the client application is misconfigured to send access tokens to a dynamically created resource server. Another vector for Access token leakage at the resource server is when the server itself gets compromised, so the attacker receives access tokens by analysing connections to the server itself [Lod+20].

4.1.10 307 Redirect

The OAuth standard does not specify which type of HTTP redirect should be implemented to redirect the user back to the client after the authentication at the authorization provider is successful. As the HTTP 307 redirect reuses the header and the body of the original request [Fie+99], a malicious client could extract the username and password of the initial form submission action because it receives this data as part of the redirection [FKS16].

4.1.11 Client Impersonating Resource Owner

In the scenario that an authorization server allows for multiple grant types, including the client credentials grant and another typical grant like the authorization code grant, the threat of a malicious client impersonating a resource owner can be present in certain implementations. One example is when the authorization server allows for dynamic registration of clients, with the possibility of setting a client ID. A client could set its ID to the value of an identifying value of a resource owner. To build the example further, Open ID Connect uses a token’s subject property to identify a user [Sak+14]. A client could use the subject value as its client ID. Improper implementations of resource servers, which do not distinguish between token types by grant type, could mistake an access token issued to a resource owner with a token issued to a client, which allows the malicious client to access the protected data of the resource owner [Lod+20].

4.1.12 Authorization Server Redirecting to Phishing Site

When the authorization server allows dynamic client registration, an attacker could create a valid client to which the authorization server could redirect. The attacker then crafts a malicious authorization request that will always fail by appending an invalid *scope* value and then will redirect to the phishing site. An example of such a malicious authorization request is depicted in Figure 4.1. As the authorization attempt using this crafted URL is always invalid because of the *scope*, the victim immediately gets redirected back to the site given by the *redirect_uri* parameter, which legally got enlisted through dynamic client registration. This site could be a copy of the valid login page to trick the user into entering its credentials. This way of phishing is very subtle as the domain of the phishing link is valid and known by the victim. The victim would need to identify that the *redirect_uri* query parameter is invalid and realize that it gets redirected to this URL after clicking on the link with the valid domain [Lod+20].

https://valid-site.com/authorize?scope=invalid&redirect_uri=https://
phishing-site.com/login&client_id=client_id_of_malicious_client

Abbildung 4.1: Phishing request

4.1.13 Unvalidated Redirects and Forwards

This type of vulnerability, known as *Unvalidated Redirects and Forwards* (URF) as well as *Open Redirect*, exists when a web application exposes redirection or forward capabilities to untrusted user input, for example, through query parameters. An attacker could generally utilize URF vulnerabilities to craft phishing links that are masked with valid, trustworthy domains [WW15]. Especially in connection with OAuth, an attacker using an existing URF vulnerability in a client can potentially circumvent whitelists for redirection URIs, by masking the redirect to a malicious client with a valid client exposing an open redirect in the query parameter [Lod+20]. Section 4.1.12 describes a different attack aimed at masking a phishing attack utilizing a mechanism specific to OAuth that is similar to an open redirection at the authorization server and therefore a threat, which is introduced by the implementation of OAuth itself.

4.1.14 Clickjacking

As authorization providers authorize applications to access confidential data, they are susceptible to being targeted by clickjacking attacks. Clickjacking attacks trick users into performing clicks on elements on a web page the users did not intend to interact with, e.g., by overlaying invisible iframes. In the case of OAuth, an attacker could create a malicious application and register it at the authorization provider of the target. The attacker also prepares a webpage that tricks the user into clicking on an invisible iframe of the authorization provider. The iframe could contain the grant access step of allowing the malicious application to access the user's confidential data. If the user has an active session at the authorization provider, a single click is enough to fulfill this action [GRC14].

4.2 Countermeasures

4.2.1 PKCE system

As an extension to OAuth defined by RFC 7636, "'Proof Key for Code Exchange" is a technique to mitigate Authorization Code Injection misuse or CSRF attacks [BA15]. Specifically, it extends the authorization code flow

OAuth Security: Finish section about PKCE

4.2.2 Refresh Token Protection

OAuth Security: Write about Refresh Token Protection

4.2.3 Misuse of Stolen Access Tokens as Countermeasure

OAuth Security: Write about Misuse of Stolen Access Tokens

4.3 OAuth Threats by Mitigation Responsibility

Threat	User Agent	Client	Authorization Server	Resource Server
Insufficient Redirect URI Validation		o	x	x
Credential Leakage via Referer Headers	x			
Credential Leakage via History Logs	x	x	o	
Mix-Up Attacks		x	x	
Authorization Code Injection		x	x	
Access Token Injection		x	x	
PKCE Downgrade Attacks			x (Mandatory PKCE)	
Access Token Leakage at the Ressource Server			o (Sender-Constrained Access Tokens)	x
Misuse of Stolen Access Tokens			x (Sender-Constrained Access Tokens)	x
307 Redirect			x	
Client Impersonating Resource Owner			x	
Authorization Server Redirecting to Phishing Site			x	
Attacks on In-Browser Communication Flows		x	x	
Open Redirection		x	x	
Clickjacking			x	
Cross Site Request Forgery		x	x	

Abbildung 4.2: Draft Table (Make LaTeX table, when table is finalized)

Maybe add countermeasures to table

Make LaTeX table with finished overview

5 Intrusion Detection Approach

Maybe integrate this chapter into 6

5.1 k-nearest neighbor

5.2 k-means clustering

5.3 Rule-based techniques

5.3.1 Count auth code usages

5.3.2 Seperate flows and check completion

6 Experimental Analysis

This chapter examines the capabilities of the intrusion detection techniques and algorithms presented in chapter 6.

Initially, section 6.1 describes the experimental environment for dataset generation and analysis. This section is subdivided into three parts to describe the multi-step process that was implemented to generate and analyze datasets for the experiments.

6.1 Environment Setup

An essential piece for this research and its experiments is the dataset for testing intrusion detection methods. At the time of this writing, a dataset of network logs with specific attacks on OAuth does not exist. Hence, the first part of the experimental setup is the dataset generation.

6.1.1 OAuth flow execution and Logging

For this first part, an OAuth environment was implemented, which consists of several subsystems that are needed to execute the OAuth protocol flow. This experimental setup is illustrated in Figure X. It consists of two independent networks. The first network contains the authorization provider and a logging service, and the second network contains the OAuth client service and a logging service. The loggers produce .pcap-files of all activity in their network using "tcpdump".

Experimental Analysis: Create figure of experiment environment

The OAuth authorization framework is practice-oriented. Therefore, the network logs are divided between the auth provider and the client, as in practice, mostly two different parties run these services. The auth provider implements four OAuth grants, which are among the most essential grants, namely Ïmplicit Grant", Äuthorization Code Grant", "Client Credentials Grant", and "Password Grant". It also implements extensions like the "Refresh Token Grantänd the "PKCE-extension. In addition to the Authorization capabilities, the auth provider service also offers a protected resource in the form of an API, which is only accessible by authorized clients. The protected endpoint is '/api/me' and returns the username of the authenticated and authorized user.

The client is a simple static webpage that handles the authorization code flow. It can be used to generate traffic manually, but its primary purpose is to handle redirections as part of the different flows. It is written in HTML, CSS, and Javascript to utilize the Fetch API and the Browser Storage API.

The following service is the attacker server implemented in Python using the "http.server"library, which serves the purpose of a callback handler for attacks that redirect the victim's OAuth flows to the attacker. It also completes OAuth flows with stolen authorization codes or access tokens and generates traffic at the auth provider like this.

6.1.2 Fuzzing

The above-described services create a complete environment for executing valid OAuth flows and attacks on OAuth. A generator service was implemented in Python to utilize this whole setup to produce network logs. The generator uses fuzzing to generate network traffic, including attacks randomly, which then gets logged by the logger services attached to the networks of the auth provider and client. The generator also logs whenever an attack is executed in its process output.

6.1.3 Analysis

The last piece of the experimental setup is the intrusion detection mechanism. This part is again a multi-step process. In the beginning, the network logs of the logger services are statically analyzed by the ßeekÏDS using the -readfileöption of zeek. Zeek then produces so-called ßeek-logs". In order to process the zeek-logs, the ßat"library is used to load the data into Python "pandas"data frames. The reason for this is that "pandasïs a popular Python data analysis library, which already implements several valuable functions, like clustering or model-based algorithms.

7 Conclusion

Literatur

[BA15] J Bradley und N Agarwal. *RFC 7636: Proof Key for Code Exchange by OAuth Public Clients*. 2015.

[Bel+22] Yousra Belfaik u. a. *Single Sign-On Revocation Access*. In: *Advances in Information, Communication and Cybersecurity: Proceedings of ICI2C'21*. Springer. 2022, S. 535–544.

[BPM] Chetan Bansal, Sören Preibusch und Natasa Milic-Frayling. *Cache Timing Attacks Revisited: Efficient and Repeatable Browser History, OS and Network Sniffing*. In: ().

[Den+19] William Denniss u. a. *Oauth 2.0 device authorization grant*. Techn. Ber. 2019.

[EPS22] Benjamin Eriksson, Pablo Picazo-Sanchez und Andrei Sabelfeld. *Hardening the Security Analysis of Browser Extensions*. In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. SAC '22. Virtual Event: Association for Computing Machinery, 2022, S. 1694–1703. ISBN: 9781450387132. DOI: 10.1145/3477314.3507098. URL: https://doi.org/10.1145/3477314.3507098.

[Fie+99] Roy Fielding u. a. *RFC2616: Hypertext Transfer Protocol–HTTP/1.1*. 1999.

[Fie99] Roy Fielding. *Hypertext transfer protocol*. In: *RFC 2616* (1999).

[FKS16] Daniel Fett, Ralf Küsters und Guido Schmitz. *A comprehensive formal security analysis of OAuth 2.0*. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, S. 1204–1215.

[GRC14] Kevin Gibbons, John O Raw und Kevin Curran. *Security evaluation of the OAuth 2.0 framework*. In: *Information Management and Computer Security* 22.3 (2014).

[Ham10] Eran Hammer-Lahav. *Rfc 5849: The oauth 1.0 protocol*. 2010.

[Har12] Dick Hardt. *Rfc 6749: The oauth 2.0 authorization framework*. 2012.

[HKW23] Pedram Hosseyni, Ralf Küsters und Tim Würtele. *Formal security analysis of the OpenID FAPI 2.0 Security Profile with FAPI 2.0 Message Signing, FAPI-CIBA, Dynamic Client Registration and Management: technical report*. In: (2023).

[HPL23] Dick Hardt, Aaron Parecki und Torsten Lodderstedt. *The OAuth 2.1 Authorization Framework*. Internet-Draft draft-ietf-oauth-v2-1-09. Work in Progress. Internet Engineering Task Force, Juli 2023. 90 S. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-v2-1/09/.

[Khr+19] Ansam Khraisat u. a. *Survey of intrusion detection systems: techniques, datasets and challenges*. In: *Cybersecurity* 2.1 (2019), S. 1–22.

[Lia+13] Hung-Jen Liao u. a. *Intrusion detection system: A comprehensive review*. In: *J. Netw. Comput. Appl.* 36 (2013), S. 16–24. URL: https://api.semanticscholar.org/CorpusID:18434328.

[LM16] Wanpeng Li und Chris J Mitchell. *Analysing the Security of Google's implementation of OpenID Connect*. In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer. 2016, S. 357–376.

[Lod+20] Torsten Lodderstedt u. a. *OAuth 2.0 security best current practice*. In: *IETF Web Authorization Protocol, Tech. Rep. draft-ietf-oauth-security-topics-16* (2020).

[MP22] Kindson Munonye und Martinek Péter. *Machine learning approach to vulnerability detection in OAuth 2.0 authentication and authorization flow*. In: *International Journal of Information Security* 21.2 (2022), S. 223–237.

[PPJ22] Pieter Philippaerts, Davy Preuveneers und Wouter Joosen. *OAuch: Exploring Security Compliance in the OAuth 2.0 Ecosystem*. In: *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. 2022, S. 460–481.

[Sak+14] Natsuhiko Sakimura u. a. *Openid connect core 1.0*. In: *The OpenID Foundation* (2014), S3.

[SM10] Karen Scarfone und Peter Mell. *Intrusion detection and prevention systems*. In: *Handbook of Information and Communication Security*. Springer, 2010, S. 177–192.

[Wan+19] Xianbo Wang u. a. *Make redirection evil again: Url parser issues in oauth*. In: *BlackHat Asia* 2019 (2019).

[WW15] Jing Wang und Hongjun Wu. *URFDS: Systematic discovery of Unvalidated Redirects and Forwards in web applications*. In: *2015 IEEE Conference on Communications and Network Security (CNS)*. 2015, S. 697–698. DOI: 10.1109/CNS.2015.7346891.

Eidesstattliche Versicherung

Bitte verwenden Sie hier in jedem Fall die offizielle von der Prüfungsbehörde vorgegebene Formulierung der Selbständigkeitserklärung.

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ggf. streichen: Ich bin damit einverstanden, dass meine Abschlussarbeit in den Bestand der Fachbereichsbibliothek eingestellt wird.

Hamburg, den 06.01.2023

_____

Florian Nehmer

Thema: Privacy Enhancing Technologies zum Schutz von Kommunikationsbeziehungen

Bearbeiter: Eva Musterfrau, Heinz Mustermann

Datum: 29. November 2023

Muster der Literaturliste

Literaturliste

Todo list