Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

Muster des Deckblatts für Abschlussarbeiten

Entwurf Version 17. Dezember 2023

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

In the evolving landscape of internet services, the OAuth (Open Authorization) framework is an ever-growing popular tool that allows users to integrate their existing accounts holding personal data seamlessly into different online services. As OAuth offers diverse possibilities for various use cases to handle authorization, the standard has grown in complexity over time. There are formal analyses of the security of OAuth, which prove the security of the standard itself [FKS16], but with this large complexity comes great responsibility when implementing the protocol. A study by Philippaerts et al. conducted in 2022 shows that 97 of 100 popular OAuth providers leave at least one known OAuth threat unmitigated, with an average of four unmitigated threats per provider [PPJ22]. This study shows that many OAuth implementations are still flawed, and it did not even include the implementation of millions of OAuth clients, which potentially could also have flawed implementations. Adding to that picture of the current OAuth threat landscape, the introduction of new browser features, as demonstrated by the number one of the top ten web hacking techniques of 2022 published by the company PortSwigger [Ket22], revealed even newer and more niche attack vectors on OAuth as well. This concerning situation in the threat landscape of OAuth is not hopeless in any way, more than that, this condition should be the reason to be addressed appropriately. One way to achieve that is to detect intrusion attempts through OAuth flaws as early as possible to allow OAuth providers to react adequately and reduce damages. This can be achieved by utilizing anomaly detection techniques, even identifying unknown attack vectors, which is especially useful in an ever-changing threat landscape.

1.2 Research Goals

This research includes two primary goals. The first goal is to provide an overview of the current threat landscape of the OAuth 2.0 protocol, including potential countermeasures by suggesting a classification.

Based on the knowledge gained by evaluating the threat landscape, the second goal is to test the effectiveness of common anomaly-based intrusion detection techniques on the application layer data of the protocol. To achieve this goal, two clustering algorithms using the protocol data encoded into word embeddings are implemented and compared to each other by their effectiveness.

1.3 Outline

While this chapter served to introduce the issues with the current threat landscape of OAuth and the motivation and approach to address them, the subsequent chapters are structured as follows:

Chapter 2 provides fundamental knowledge about the OAuth authorization framework, including its different modes of operation and its future outlook. It also introduces the theory of intrusion detection systems based on recent taxonomies. The chapter ends by describing the different algorithms utilized to implement the anomaly-based intrusion detection approach applied in this work.

Chapter 3 presents related research in the realm of OAuth security in general and intrusion detection approaches utilizing Word2Vec embeddings and clustering algorithms as this work applies them.

Chapter 4 continues by laying out the different threats the implementation of OAuth bears and presents the current state of countermeasures to be applied. The chapter finishes with a suggestion for a classification of the various threats in the context of their countermeasures.

Chapter 5 starts by giving a detailed description of the implementation of the experiments, including the OAuth environment, the dataset generation, and the algorithmic intrusion detection method. It continues by presenting and comparing the effectiveness of the different methods as the results of the experiments.

Chapter 6 ends this work with a résumé of this work and an outlook for future work.

2 | Fundamental Knowledge

The OAuth authorization framework is complex and extensive as it has been refined and extended through several standards over the last 11 years. Securing applications using the OAuth protocol, therefore, leads to various considerations. Consequently, section 1 introduces the essential aspects and vocabulary of the OAuth authorization framework in the web context to form an understanding of the security-relevant parts of the protocol.

As one of the main goals of this thesis is to experiment with techniques to detect attacks on services using OAuth, section 2 provides an exploration of intrusion detection systems and the different ways to categorize them.

In the last part of this chapter, section 3 concludes with a description and depiction of the different algorithms used in the experiments.

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.

Improve all graphics resolutions
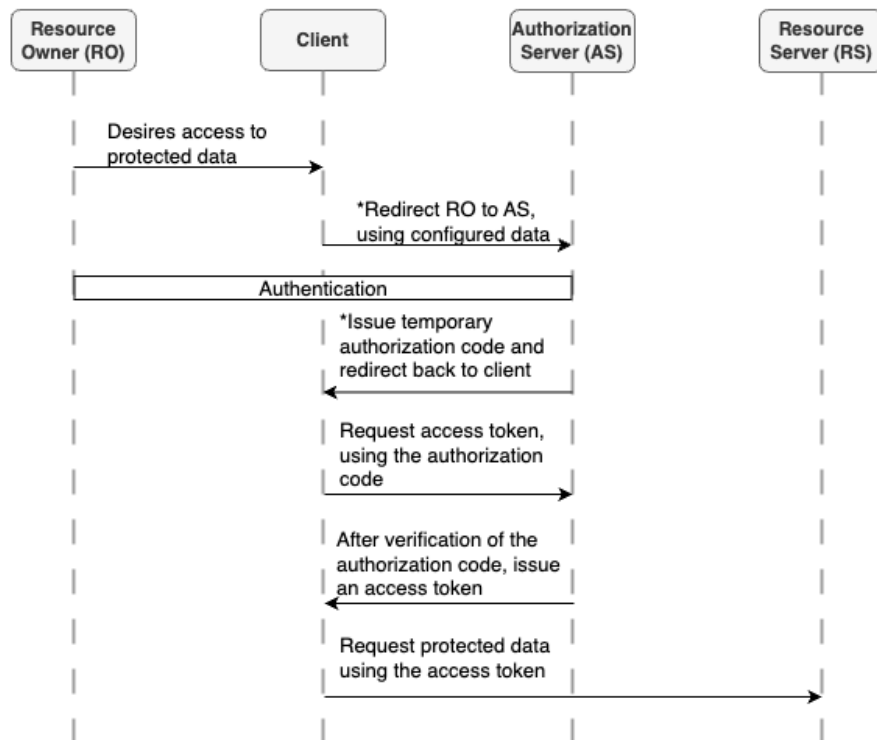
Abbildung 2.1: Authorization Code Grant without any extensions

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 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

Abbildung 2.2: Implicit Grant

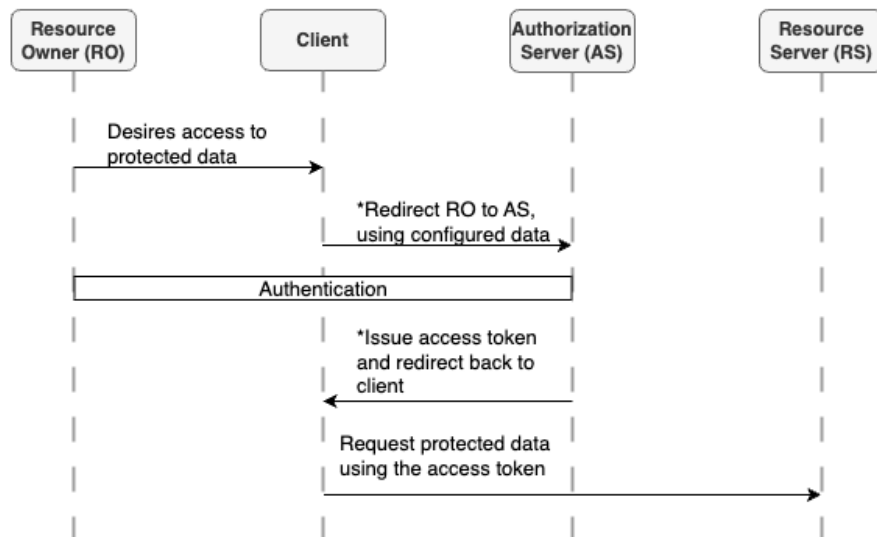themselves. A common use-case for this scenario would be machine-to-machine interactions. This grant type is meant for clients to access their own resources as is the case in micro-service architectures. As depicted in figure 2.4 the client authenticates at the authorization server with its client secret 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.]

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

11
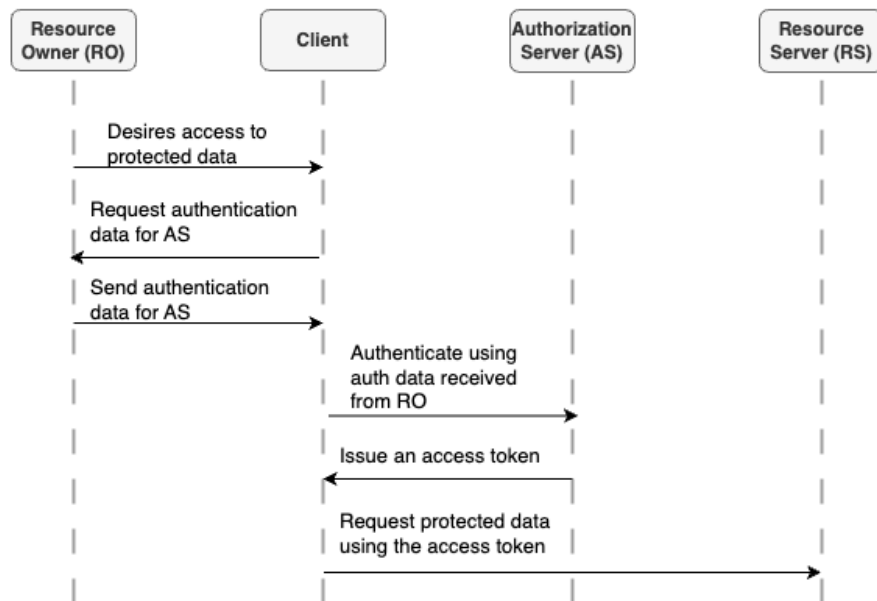
Abbildung 2.3: Resource Owner Password Credentials Grant

- UMA Grant

- SAML 2.0 Bearer Grant

- Token Exchange 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 [HPL23a]. 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

Abbildung 2.4: Client Credentials Grant

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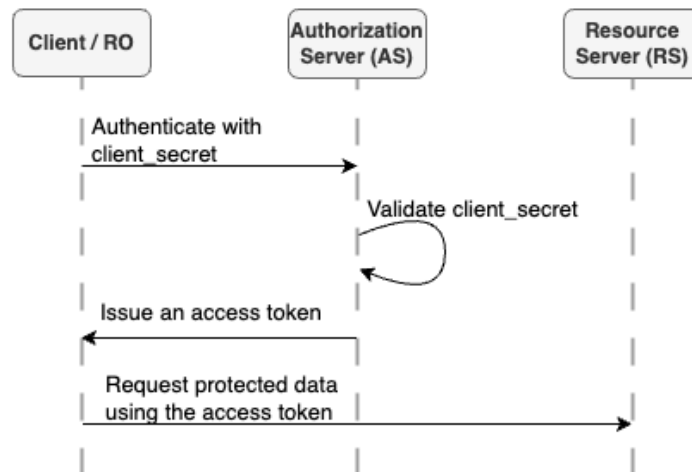
- Access tokens must not be transported in a Request-URI in any case (which also makes the Implicit Grant impossible

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

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.

Fundamental Knowledge: Make clearer that the long definitions use one source
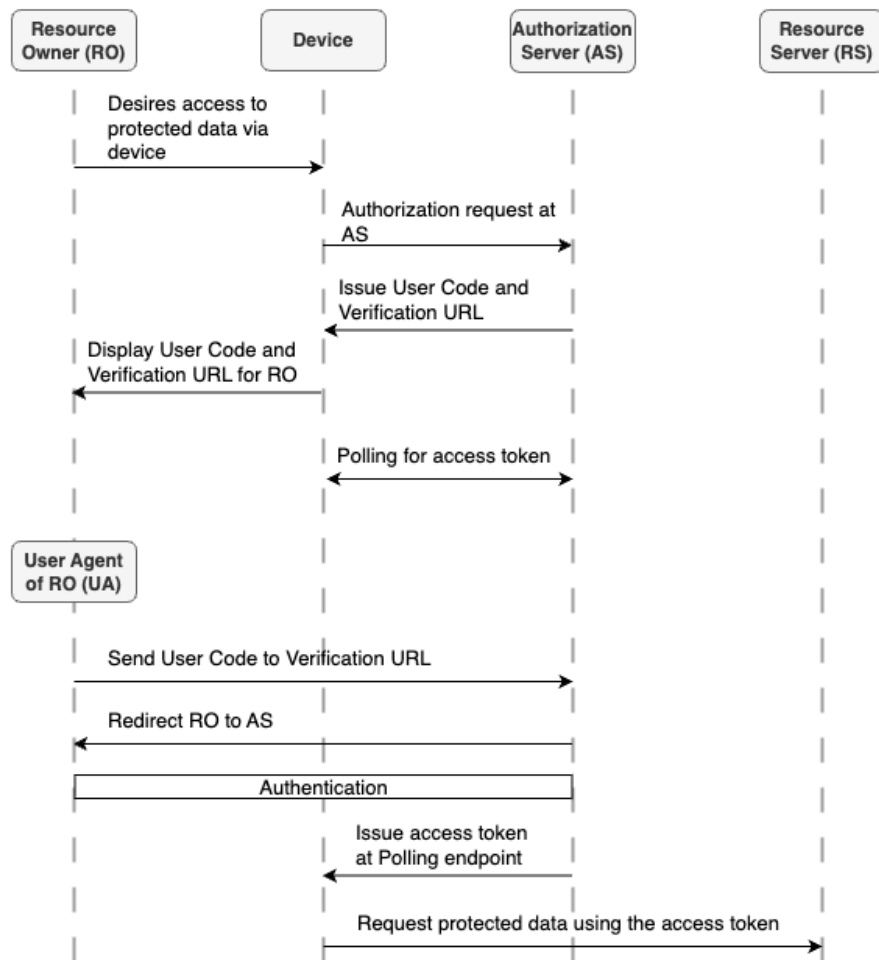
Abbildung 2.5: Device Authorization Grant

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)*.

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

Zeek formerly known as *bro*, as a reference to George Orwell's 1984 as a "reminder that monitoring comes hand in hand with the potential for privacy violations" [Pro23] is an open source network intrusion detection system. The first version of *bro* was released in 1995 by the Lawrence Berkeley National Laboratory. In 2018, the project was renamed to *zeek*. The system is often used for network security monitoring to detect and analyze suspicious behaviour in network traffic. Its architecture consists of two major components, as displayed in Figure. The first component is the *event engine*, which refines the incoming network traffic into higher-level events without doing any analysis. It processes the traffic through different subcomponents, identifying protocols, file types, and, e.g. TCP connections at the link layer. The second component is the *policy script interpreter*, which runs different event handlers to

analyze the events. These event handlers are built in their own scripting language, but adaptors to public programming languages also exist and are applied in this work.
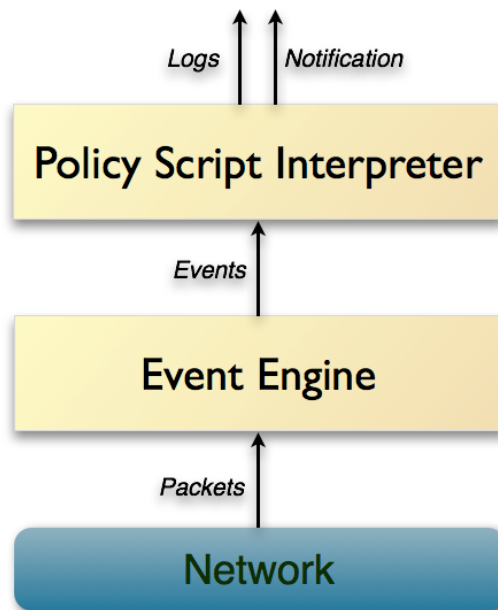
Abbildung 2.6: Architecture of zeek [Pro23]

2.3 Algorithms

Write intro to the algorithms

2.3.1 Word2Vec

Word2Vec introduced in 2013 by Mikolov et al. is a technique in natural language processing to convert words into embeddings. An embedding is a vector carrying a semantic to represent a word as a numerical value to enable the usage of words in algorithms requiring numerical inputs. In the case of *Word2Vec*, the idea is that words hold similarities based on their surrounding words in a text corpus. This concept leads to words, which often stand close together in a text, receiving vectors that point in a similar direction. There are two models which are employed by *word2vec*, the *Continous Bag of words (CBOW)* model and the *Skip-Gram* model. Both models have in common that they apply supervised learning techniques by training a neural network to predict relationships in text corpora. In both cases, the goal is to use the weights of the hidden layer after training as word embeddings instead of utilizing the neural networks for any predictions [Mik+13] [Ron14].

Continous Bag of Words (CBOW)

The continuous bag of words model trains a neural network that predicts the word in the middle based on its surrounding words. An important hyperparameter for CBOW and Skip-Gram is the window size, which describes how many words surrounding a word are considered context.

During training, the network receives combined one-hot-encoded vectors of surrounding words (e.g., averaging or summing them) in a tuple with the corresponding target word. When the training is completed at inference, the model gets context words as input and outputs a vector of probabilities for the predicted target word. But as mentioned above, the goal of word2vec is to extract word embeddings, which are the weights of the hidden layer of the neural network.

Skip-Gram

The skip-gram model uses the reverse strategy compared to the CBOW model. Instead of predicting a word by its surroundings, it trains a neural network that tries to predict the surroundings of a word. At training, the neural network receives word tuples of single context words and their corresponding target words one-hot-encoded. After fitting the network during inference, the model receives a one-hot-encoded word as input and produces a probability distribution over the vocabulary, which stands for the probability of a word being a surrounding word for the input. Again, in the case of word2vec, this supervised learning approach is only applied to retrieve the weights of the hidden layer as word embeddings, as word2vec is used for non-labelled data. In their research, Mikolov et al. state that skip-gram is feasible for small data sets and even considers rare occurrences of words well. In contrast, CBOW is faster in training with large datasets and better at capturing the semantics of high-frequency words.

2.3.2 k-Means Clustering

In the field of data analysis, finding patterns and anomalies in data sets is a ubiquitous challenge. Grouping data into different clusters can be an approach to reveal useful features or meanings from the data. One common technique for clustering is k-means clustering. There are different flavors of k-means algorithms, but one of the most common k-means algorithms and the one applied in this work is Lloyd's k-means algorithm [WH07].

Lloyd's k-Means clustering

The general goal of the algorithm is to group data by similar features into a fixed amount of clusters. The algorithm achieves this by finding k *centroids* $\mu_1, \mu_2...\mu_k \in \mathbb{R}$. The clusters are then based on the centroids so that every point x of the n-dimensional dataset $x_1...x_m$ gets assigned to the centroid with the lowest Euclidean distance. In order to find the centroids, the algorithm's approach is an iterative process described in figure 2.3.2. In the beginning, k (defined by the number of clusters) random centroids are chosen. Then, the clusters are created by calculating the minimum Euclidean distance from every point of the dataset to the centroids: $\arg\min_j \|x_i - c_j\|^2$. In the next step, the centroids are recalculated by finding the points in the center of the clusters: $\mu_j = \frac{1}{|\mu_j|} \sum_{x_i \in \mu_j} x_i$. The two steps of creating the clusters based on the centroids and calculating the center point of the new clusters are repeated until the centroids converge or a definition for a fixed amount of iterations is given [Pie13] [Llo82].

Adjust with stanford formulas

<div align="center">Algorithm 1: Lloyd's K-Means Algorithm</div>

1: **Input:** Data set $X = x_1 \ldots x_m$, number of clusters k
2: **Output:** Cluster centroids $\{\mu_1, \mu_2, \ldots, \mu_k\}$
3: Randomly choose k centroids $\mu_1 \ldots \mu_k$
4: **repeat**
5:     **for** each data point x_i **do**
6:         Assign x_i to the cluster with the nearest centroid: $\mu_i = \arg\min_j \|x_i - \mu_j\|^2$
7:     **end for**
8:     **for** each cluster j **do**
9:         Calculate new centroid μ_j as the center of all points assigned to cluster j for every cluster: $\mu_j = \frac{1}{|\mu_j|} \sum_{x_i \in \mu_j} x_i$
10:     **end for**
11: **until** New centroids don't change anymore or a fixed number of iterations was given

---

2.3.3 Self-Organizing Maps

<div style="border:1px solid orange; color:orange; display:inline-block">Write about Self-Organizing Maps</div>

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

3 | Related Work

This chapter presents important research in the realm of OAuth security and intrusion detection using similar techniques as proposed in this work. As this thesis first builds a strong foundation in OAuth threads and vulnerabilities, Section 3.1 starts by presenting an important publication in the realm of theoretical analysis of OAuth by Fett et al..

Section

3.1 Theoretical Security Analysis of OAuth

Security research on the OAuth standard has been conducted since its beginnings. Impactful research that unveiled security issues in the definition of OAuth 2.0 is the work by Fett et al. [FKS16], who constructed a theoretical *Dolev Yao*-style model called the FKS model and extended that model with OAuth2 capabilities. On the one hand, the FKS model mimics web standards, like HTTP/1.1 and HTML5. On the other hand, it simulates entities like browsers, including the concept of windows, documents, iframes, complex navigation rules, DNS servers, web servers, and web and network attackers. The FKS model puts entities as processes into practice, while the communication through standards is modelled via events between the processes. For OAuth 2.0, the researchers' goal is to reveal security flaws in the most secure implementation of the standard regarding all publicly known security considerations while modelling the four grant types: *implicit grant*, *authorization code grant*, *resource owner password credentials grant*, and *client credentials grant*. The central methodology of Fett et al. is to prove the security of the OAuth standard using their formal language, in which the standard is modelled, to find the flaws breaking the security proof. Following this methodology, the researchers uncovered four new vulnerabilities in the OAuth standard, with the impact that the OAuth working group of IETF added these vulnerabilities and their countermeasures to the official document for security considerations for OAuth. These four vulnerabilities are later elaborated in this work in sections .

Add the reference to sections

Add the table from the research

3.2 Analysis of OAuth implementations

Through research like the one mentioned in Section 3.1 by Fett et al. and the continuous work by the OAuth working group of the IETF, the OAuth standard gets enriched with several suggestions for improving the security of OAuth implementations. However, because OAuth is such a complex protocol, implementation flaws still arise, even concerning already-known threats. A recent study by Philippaerts et al. implemented an audit tool called "OAuch" to test the coverage of countermeasures against known threats for OAuth. It revealed that 97 of 100 popular identity providers (OAuth and Open ID Connect) leave at least one common threat completely unmitigated. The researchers also found that, on average, OAuth identity providers, in particular, did not implement 34% of the security specifications suggested by the OAuth working group

[PPJ22]. This work again shows the importance of awareness of the threat landscape when implementing and operating an OAuth provider or an OAuth client.

3.3 Word embeddings and clustering for intrusion detection

Write about the related work by [ZZS17] and [NM14]

4 | OAuth Security

This chapter presents known threats and vulnerabilities when using the OAuth authorization framework. Section 4.1 elaborates on every threat and vulnerability separately. The list is mainly based on the current draft for security best practices for OAuth from the OAuth task force at IETF and is enriched with various examples, further explanations, and a categorization of types of threats.

Section 4.2 starts with a classification of OAuth threats and vulnerabilities by mitigation responsibility of the involved parties in the protocol. It then continues with a description of the specific countermeasures and mitigation strategies and ends with an overview of which threat can be mitigated by which countermeasure.

4.1 Threats and Vulnerabilities

4.1.1 Insufficient Redirect URI Validation

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 [Lod+20].

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

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 third-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].

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.

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 Mandatory PKCE

As an extension to OAuth defined by RFC 7636, *Proof Key for Code Exchange (PKCE)* is a technique to mitigate several OAuth threats [BA15]. The main problem PKCE solves for the authorization code grant is verifying that only the original client who started an OAuth flow receives the access token, so a stolen authorization code cannot just get exchanged with an access token by the entity who stole the code. Figure 4.2 shows the way PKCE works [Sir20]:

1. Initially, before the client redirects the resource owner's user agent to the authorization server, it generates a random string of length 43 at minimum, called the *code_verifier*. It then calculates the hash value of the code_verifier and encodes it with base64 without padding. The hash algorithm in use should be one which is currently regarded as secure and known by the authorization server. The resulting value is called the *code_challenge*

2. At redirection to the authorization server, the client appends the code_challenge and the hashing algorithm, which it has used to create the code_challenge to the redirection URL as query parameters.

3. After successful authentication, the authorization server creates a record of code_challenge and the corresponding authorization code it generates and sends back to the user agent. Alternatively, it may append the code_challenge to the authorization code and create the record this way.

4. When exchanging the authorization code for an access token, the client sends the code and the code_verifier to the authorization server.

5. The client compares the code_verifier with the corresponding code_challenge of the authorization code by calculating the unpadded base64 encoded hash value of the provided code_verifier. Only the client who initially requested protected resources could possess the random value that matches the code_challenge.

The PKCE mechanism mitigates several threats and vulnerabilities, as shown in Table X. It mitigates credential leakage via referer headers (T2), as this attack aims in the case of the authorization grant to extract authorization codes after a redirect. Even with a stolen authorization code, an attacker does not know the code_verifier and thus can not receive an access token. For the same reason, any kind of authorization code injection (T5), where the precondition is a stolen authorization code, would only be possible by knowing the code_verifier. Protection against cross-site request forgery (T7) is also provided since an attacker could not trick a client into appending confidential data to the resources owned by the attacker, as the victimś client does not send the attackerś code_verifier to the authorization server. Lastly, protection against PKCE downgrade attacks (T8) is ensured when the authorization server implements PKCE as mandatory.

4.2.2 Random value state

Another method to protect against Cross-Site request forgery is by taking advantage of the state parameter the OAuth standard offers. When the client initiates an authorization request, it generates a random value of sufficient length, attaches it to the current session and sets it as the state parameter for the request. After authentication, the authorization server has to attach the previously received state parameter to the redirection URI as a query parameter. The client can now validate the received state value with the one it attached to its session earlier to ensure the redirection stems from the authorization flow it initiated [FOC15]. When PKCE for the authorization code grant is already implemented, this measure does not add any benefit regarding CSRF protection [BA15]. In conclusion, the usages of state's main difference to PKCE is that the client knows that the received authorization code is connected to its initial authorization request before redeeming an access token with the code_verifier, where the authorization server relates the client's requests to each other with the code_challenge and the code_verifier.

4.2.3 Invalidation of access tokens

The OAuth framework requires an authorization code to be redeemed only once for an access token. Therefore, an authorization code needs to be invalidated after its first usage. However, this
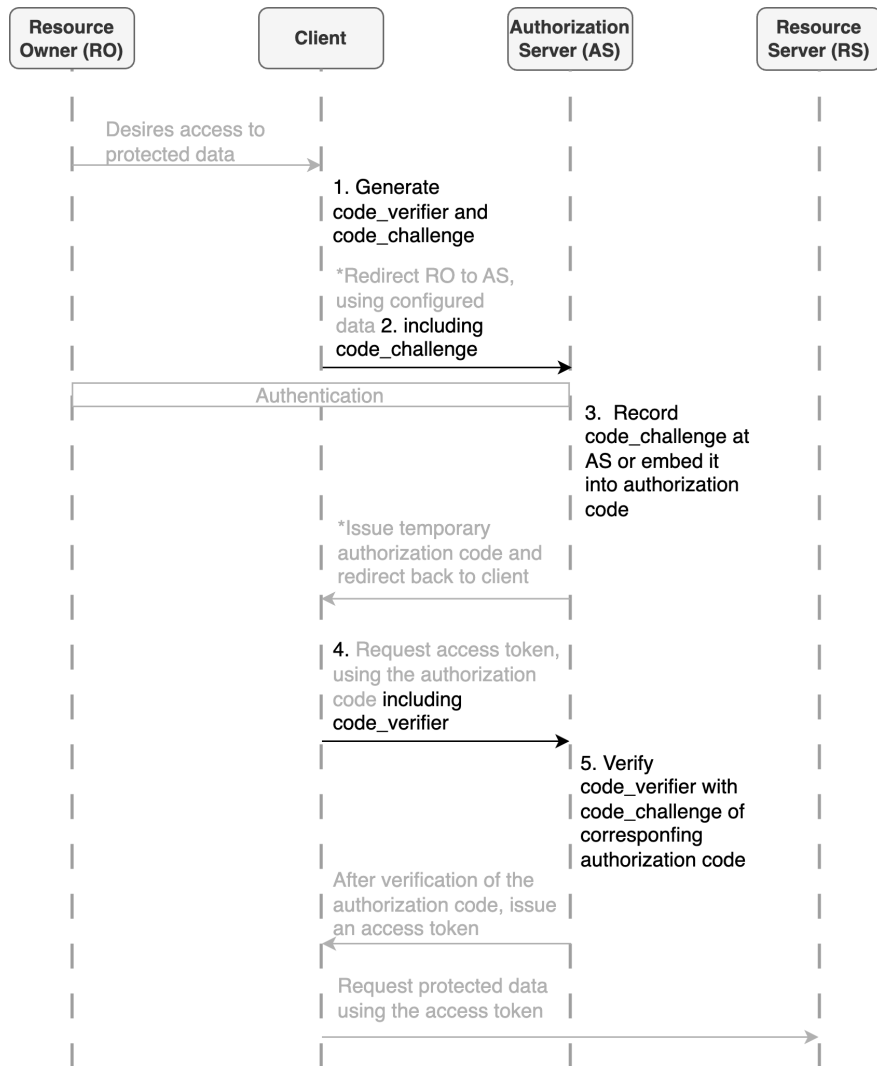
Abbildung 4.2: Authorization Code Grant with extension for Proof Key for Code Exchange

does not stop an attacker from receiving an access token with a stolen authorization code before the valid user does it. The original OAuth standard recommends invalidating a previous access token if there was an attempt to receive an access token at least twice for the same authorization code [Har12]. Invalidation of access tokens if an authorization code gets redeemed twice helps mitigate the threats of credential leakage via referer headers (T2) and credential leakage via browser history (T3) [Lod+20], as for these threats, exposed authorization codes were already redeemed in most cases.

4.2.4 Simple string comparision

A technique to completely circumvent the threat of insufficient redirect URI validation (T1) is to disallow whitelisting redirect URIs using regular expressions. Instead, exact string comparison should be the way to allow redirections [Lod+20].

4.2.5 Avoid usage of grant types

Threats like credential leakage via referer headers (T2) or browser history (T3) can be avoided when the implicit grant is not used at all, as the access token would not be in the URI of a request. Access token injection (T6) using the implicit grant makes it easy for any attacker to circumvent CSRF protection through the state parameter (see Section 4.2.2) since, for the client, it does not make a difference what access token is used for the state check [Lod+20]. For these reasons, the OAuth task force removed the implicit grant in the draft for OAuth 2.1 [HPL23b]. In the past the main benefit of the implicit flow was that it was possible to implement it for clients that did not utilize cross-origin resource sharing (CORS). CORS was finalized in 2014 by the W3C and superseded by the fetch standard afterwards [Kes14]. As the OAuth standard dates back to 2012 [Har12], the implicit grant could be implemented for applications that did not support CORS yet.

4.2.6 Constrained access tokens

This countermeasure aims to harden the protocol against the misuse of stolen access tokens if they are leaked, for example, at the resource server (T9) or through credential leakage via the browser history (T3) and or referer headers (T2). It reaches this goal by narrowing the access permissions of the access token in two ways [Lod+20]:

- *Sender-constrained access token*: The token contains cryptographical material to identify the client who redeems the access token, which opens the possibility of only allowing specific clients.

- *Audience-constrained access token*: Access tokens are constrained to particular resource servers. The resource owner executes the audience configuration at the resource server, and the resource server performs the audience verification. If the access token, in addition, bears user-identifying data, different audiences can be defined at the resource server level to further restrain access to the protected data.

4.2.7 Issuer identification

Per default, the OAuth authorization grants do not include mechanisms for clients to identify an authorization server, which sends them an authorization response through redirection. This circumstance leads to threats like Mix-Up attacks (T4), when multiple authorization servers are involved. Issuer identification has been introduced to tackle this issue with RFC9207 in 2022. The issuer identifier *iss* is a parameter which must be sent to the client in every response, even if it is an error response. The *iss* parameter contains the URL, which points to the metadata of the authorization server. This metadata has to include an "issuer" property, which contains a value identical to the value of the *iss* property. The client, which supports issuer identification, has to store the identifier locally when initiating an interaction with an authorization server. The client then must compare the stored value with the one it receives from the authorization response with a simple string comparison. The client is also responsible for ensuring that every authorization server it interacts with holds a unique issuer identifier [MF22].

4.2.8 Implementation details

There are several countermeasures that one can categorize as implementation details, as they are applied through small decisions during client or authorization server implementation. The following is a brief list of these measures as suggested by Lodderstedt et al. [Lod+20] and the classification of which involved party is responsible for implementing the measure.

- *303 redirect*: As mentioned in 4.1.10, the OAuth standard does not mandate the type of HTTP redirect to use for authorization servers. To circumvent issues with improper redirect handling leading to security issues, the authorization server should use 303 redirects.

- *Client_ID not choosable by user*: To mitigate threats that arise through the possibility of a client choosing its own client ID, as described in 4.1.11, the client ID should always be chosen at random by the authorization server, where the clients are getting registered.

- *No redirect before authentication at authorization provider*: Some threats arise because authorization servers might be implemented in a way that they are redirecting the user agent in error cases, even without complying with the redirect URI whitelists in some cases. This behaviour could lead to advanced phishing attacks described in 4.1.4. Therefore, authorization servers should be implemented in a way that they only redirect the user agent if the authentication of the resource owner has been successful.

- *No access token in uri*: The OAuth standard allows for access tokens to be transported in the request URI from the client to the authorization server when accessing protected data. This option leads to threats concerning leakage of the access token through browser history 4.1.3 or referer headers 4.1.2. Therefore, an authorization server should implement the transporting of the access token through a request body as mandatory.

- *Avoid third-party content on pages involved with OAuth*: Malicious advertisements in iframes or attacker-generated hyperlinks on pages an OAuth flow redirects to could lead to leakage of access tokens or authorization codes through referer headers 4.1.2. Therefore, authorization servers and clients must ensure that no third-party content is allowed for the redirection endpoint at the client and authentication page at the authorization server.

- *Appropriate Referer Policy*: On top of the measure described above about rendering of third-party content, to mitigate credential leakage via referer headers, an appropriate referer header policy should be implemented like the "Referrer-Policy: no-referrer" header in authorization requests or as a meta tag in HTML documents.

4.2.9 General web security countermeasures

After laying out several mitigations for attack vectors of OAuth, there are still general web security threats, which are especially important for OAuth, like clickjacking 4.1.14, open redirections 4.1.13 or CSRF 4.1.7. The reason why these common web security vulnerabilities are critical in the case of OAuth has been laid out in their specific sections. The countermeasures for those types of vulnerabilities, however, are very context-specific and are out of the scope of this work. Hence, in further portrayals of threats and countermeasures in this work, they are described as countermeasures against clickjacking, open redirect and CSRF, but do not include specifics.

4.3 Classification of OAuth threats and vulnerabilities

The OAuth authorization framework tries to solve a lot of practical authorization use cases at once and, therefore, offers a very flexible and diverse set of definitions for such use cases. Hence, it follows that the threat space and attack vectors are also complex and diverse, as laid out in the previous two sections **??** and **??**. To facilitate a general, more tangible overview of the threat situation of OAuth, this work presents a brief taxonomy based on the perspective of the involved parties.

4.3.1 Encoding of threats, vulnerabilities and countermeasures

To improve the visibility in tables for the following depictions the OAuth threats and vulnerabilities are encoded with the following scheme:

- TV1: Insufficient Redirect URI Validation
- TV2: Credential Leakage via Referer Headers
- TV3: Credential Leakage via Browser History
- TV4: Mix-Up Attacks
- TV5: Authorization Code Injection
- TV6: Access Token Injection
- TV7: Cross-Site Request Forgery
- TV8: PKCE Downgrade Attack
- TV9: Access Token Leakage at the Resource Server
- TV10: 307 Redirect
- TV11: Client Impersonating Resource Owner
- TV12: Authorization Server Redirecting to Phishing Site
- TV13: Unvalidated Redirects and Forwards
- TV14: Clickjacking

for the same reason, the countermeasures are encoded as well using the following scheme:

- C1: Mandatory PKCE
- C2: Random value state
- C3: Invalidation of access tokens
- C4: Invalidation of state value
- C5: Simple String comparison
- C6: Avoid usage of grant types
- C7: Sender Constraint Access Token

- C8: Audience Constrained Access Token

- C9: Issuer identification

- C10: 303 Redirect

- C11: Not choosable client_ID

- C12: No redirect before authentication

- C13: No access token in URI

- C14: No third-party content on pages involved with OAuth

- C15: Appropriate Referer Policy

- C16: Open redirection countermeasures

- C17: Clickjacking countermeasures

Non-encoded versions of the tables are added to the appendix and referenced from the encoded versions of the tables.

4.3.2 Threats and their countermeasures

The first classification displayed in figure 4.1 is visualizing, which countermeasure mitigates aspects of which threat or vulnerability.

	Maybe like this?	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17
TV1					x											x	
TV2	x		x	x	x									x	x		
TV3			x		x								x				
TV4								x									
TV5	x	x															
TV6						x											
TV7	x	x															
TV8	x																
TV9							x	x									
TV10									x								
TV11											x						
TV12												x					
TV13																x	
TV14																	x

Tabelle 4.1: CXT

Put vertical names countermeasures

Create caption for table

4.3.3 Threats by mitigation responsibility

Add the table for threats by mitigation responsibility

5 | Experimental Analysis

This chapter examines the capabilities of the intrusion detection techniques and algorithms presented in chapter 5.

Initially, section 5.1 describes generally the structure of 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.

5.1 Implementation of the data generation environment

An essential piece for this research and its experiments is a complete testing environment for OAuth as at the time of this writing a dataset of network logs with specific attacks on OAuth does not exist. To generate a dataset to analyze it, an experimental environment was implemented containing multiple components as illustrated in figure 5.1. It consists of three main parts, the first part being the main OAuth services, which are the authorization server, the resource server, and the client, to make OAuth network traffic in general possible. The second part is the dataset generation part, which is done through fuzzing requests in the OAuth environment, which produces logs in the form of ".pcap"-files through the logger services. The third and last part is the analysis part, which is initialized through the preprocessing of zeek, which produces several log files from which the "http.log" files are getting processed in the analyzer component. The analyzer component executes the implemented algorithms to detect anomalies.
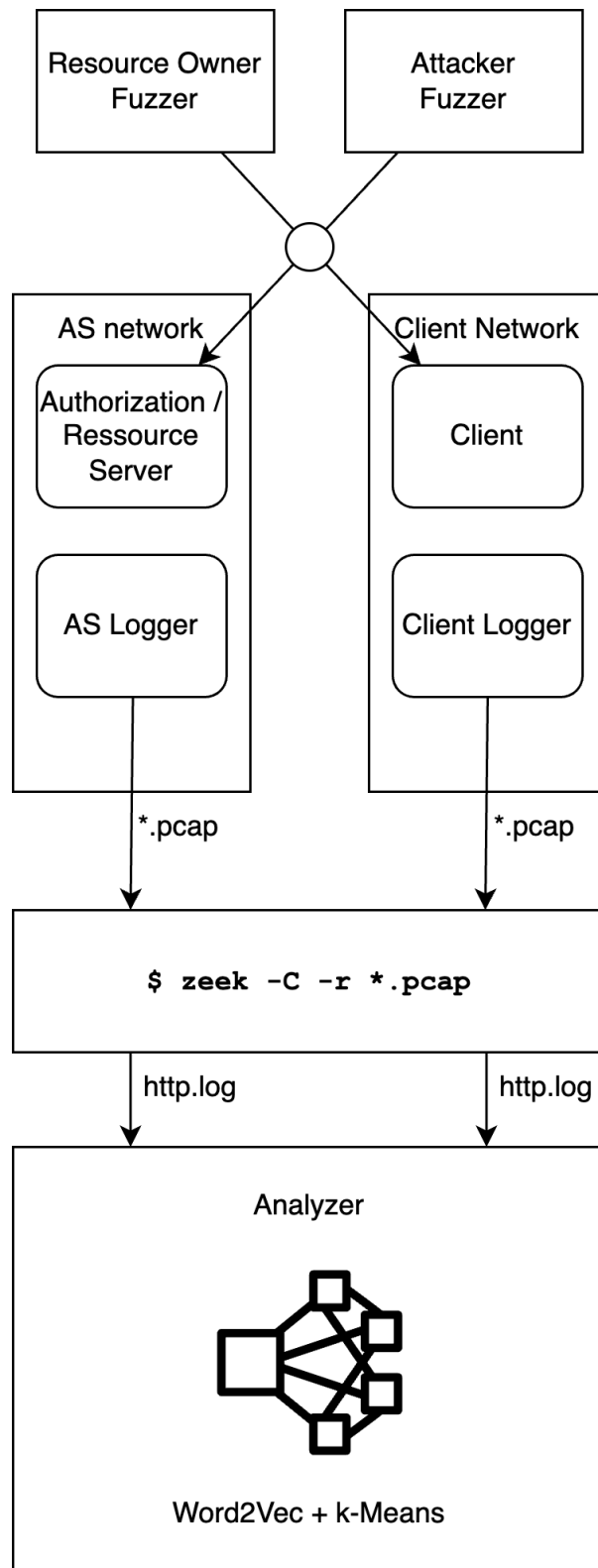
Abbildung 5.1: Overview of the environment for the experiments

Change k-Means to clustering in graphic

5.1.1 OAuth setup

For this first part, an OAuth environment was implemented, which consists of several subsystems that are needed to execute the OAuth protocol flow. It consists of two independent networks. The first network contains the authorization provider, resource server 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*. 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.

Authorization Server and Resource Server

The implementation of the authorization server and resource server is forked from the *authlib* project version 1.2.1 [Ltd23]. More specifically the "Flask OAuth Providers" component is utilized for the authorization and resource server. To make the authorization process more vulnerable the mandatory PKCE feature was deactivated. Having the possibility to make adjustments like this, was the reason for forking the project, instead of importing it as a library. The authorization server implementation of *authlib* offers four OAuth grants out of the box, which are among the most essential grants, namely "Implicit Grant", "Authorization Code Grant", "Client Credentials Grant", and "Password Grant". It also implements extensions like the "Refresh Token Grant" and the "PKCE"-extension. Besides the authorization capabilities, the authlib implementation also offers a protected resource in the form of an API, which is only accessible by authorized clients. The protected endpoint it offers returns the username of the currently authorized user.

Client

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. An implementation of redirect handling is shown in figure 5.1. The client is checking if the code query parameter is present in the currently active URL (line 6) when the page loads. If it is present it extracts it and retrieves an access token from the authorization server.

```javascript
 1  // fetch access token on page load, if the code query param is present
 2  window.onload = function() {
 3      const params = new URLSearchParams(window.location.search);
 4      const client_id = "<pre-configured client ID>"
 5
 6      if (params.has("code")) {
 7          const form = new FormData();
 8          form.append('grant_type', 'authorization_code');
 9          form.append('scope', '<some scope>');
10          form.append('code', params.get("code"));
11          form.append('client_id', client_id);
12          fetch('<token endpoint of authorization server>', {
13              method: 'POST',
14              body: form
15          })
16      }
```

```
17 │ }
```
Listing 5.1: Example implementation of authorization code handling at the client

5.1.2 Network setup and logging

All services are hosted using containers in a docker environment. Technically these services are part of the same network at the docker host they are run on, but the logging services build overlay networks on the services they are observing. As visualized in figure 5.1 there exist two logging services, one for the authorization and resource server and one for the client network. When the experiment environment is active, the logging services stream all network events to a .pcap file, which is accessible on the docker host machine through a volume. All communication is executed without TLS to be able to analyze all packets. In a real-world scenario this would not be the case, but mechanisms like proxies, who are acting as man-in-the middle would be able to inspect the network traffic unencrypted in that case as well.

```
1 │ $ ethtool -K eth0 rx off tx off tso off;tcpdump -i eth0 -w
  │     /log/"$filename_prefix"-log-$(date +"%Y%m%d_%H-%M-%S").pcap
```
Listing 5.2: Logger process

5.1.3 Attacks

For the attacks on OAuth several pre-conditions were established to make the OAuth environment vulnerable. PKCE as mentioned earlier has been made optional. There is not any CSRF protection in place, utilizing the state parameter. This leads to the attacks being harder to detect as omitting these parameters as an attacker, will not make a difference anymore compared to the usual traffic. Another assumption is that an attacker is at all times able to circumvent any whitelist for redirect URIs. This is simulated by allowing the URI of the attacker server in the securely implemented whitelist of *authlib*. In addition, it is assumed that an attacker, who crafts a malicious link can make the victim use the malicious link, through phishing or similar techniques. Another precondition is that the victim is logged in at all times and if it is not logged in it will immediately complete the login process. To facilitate the attacks an attacker callback server has been implemented as well. Also, every HTTP request involved in an attack, even the ones where the victim is visiting a maliciously crafted URL is labeled with a special header *X-Is-Attack* with the name of the attack as its value.

Attacker callback server

The attacker callback server is a service implemented using the Python "http.server" library, which serves the purpose of a callback handler for attacks that redirect the victim's OAuth flows to the attacker. This means if an authorization code or access token ends up being redirected to this callback server it immediately completes OAuth flows with the stolen authorization codes or access tokens and generates traffic at the auth orization server like this. An example implementation for such a callback handler is shown in figure 5.3. The "do_GET" method (line 2) is executed, whenever a GET request reaches the attacker server. If this request contains an

authorization code parameter (line 7), the server tries to redeem it at the authorization server (line 16). The server then tries to access protected data at the resource server (line 21).

```python
class OAuthCallback(BaseHTTPRequestHandler):
    def do_GET(self):
        parsed_url = urlparse(self.path)
        query_params = parse_qs(parsed_url.query)

        # Abort if there is no 'code' parameter present.
        assert query_params["code"] is not None

        # Fetch access token with code
        auth_code=query_params["code"]
        data = {"grant_type": "authorization_code",
                "code": auth_code,
                "redirect_uri": REDIRECT_URI,
                "client_id": CLIENT_ID,
                "client_secret": ""}
        res = r.post(f"{AUTHORIZATION_SERVER_URL}/oauth/token",
            data=data)
        access_token_data = res.json()

        # Utilize stolen access token
        headers = {"Authorization": f"Bearer
            {access_token_data['access_token']}"}
        res = r.get(f"{RESOURCE_SERVER_URL}/api/me", headers=headers)
        print("whoami:", res.content)
```

Listing 5.3: Example implementation of an attacker server, which handles redirections

Arbitrary redirect URI

The first attack implemented is making the victim start an OAuth flow using the authorization code grant, but redirecting it to the attacker callback server after authorization. This is done by simply exchanging the value of *redirect_uri* parameter to the location of the attacker callback server. The callback server then uses the received authorization to redeem an access token at the authorization server as displayed and already explained in figure 5.3. The attacker then uses the access token to fetch protected resources. This is a very basic attack and assumes that there is not any kind of whitelist in place like it is simulated in the experiment environment. However, there are more sophisticated versions of this attack utilizing the e.g. *Evil Slash* [Wan+19] trick mentioned in Section 4.1.1.

Arbitrary redirect URI using "Evil Slash"

The evil slash attack is based on the fact that the parsing of URLs of browsers might mismatch the way the parser for a whitelist parses a given URL. There are two examples given by the research of Wang et al. that are related to handling forward and backward slashes as path separators. Suppose either the whitelist parser or the browser URL parser treats some slash as a path separator while the other parser does not. In that case, this behaviour can be exploited to circumvent the whitelist of the authorization server. These two versions were implemented in the experiments by crafting a URL, which relies on this type of vulnerability.

5.1.4 Network data generation

The above-described services create a complete environment for executing valid OAuth flows and attacks on OAuth. To generate the data set for the analysis, a generator script was implemented in Python to utilize the whole setup to produce network logs. The approach for the generation is to mostly produce random valid traffic in the network. Meanwhile, at random some amount of attacks are executed as well, with a probability of 5%, to simulate subtle attacks on the authorization. As the logging services are always listening, they produce network log files, which are then extracted after a reasonable amount of time.

5.2 Implementation of analysis algorithms

The last remaining part of the experimental setup is the intrusion detection mechanism. Based on the type of attacks implemented the decision was made, to analyze the logs produced by the authorization server, because the attack traffic appears in the network of this entity. The input for this part of the experiments is therefore the network log file produced by the logger of the authorization server overlay network. This file is passed to the data preprocessing step initiated by *zeek*. The approach for the next step after the preprocessing, the detection step is based on previous studies [CS18] [ZZS17], which had success in utilizing word2vec word embeddings to detect malicious network traffic even in small datasets. Therefore the same idea was chosen for this work to encode textual data, to enable clustering methods of the logs to reveal anomalies in the network data.

5.2.1 Data preprocessing

The first step of processing the data is to run it through the *zeek IDS* to produce "zeek-logs". Zeek splits the data into different processed log files depending on the protocol of the packets. Since this research focuses on attacks on a specific protocol in the application layer, all logs, except for the http-log, get omitted. Another important detail is, that while producing the zeek-logs a "zeek-script" is loaded to preserve the HTTP headers in the HTTP log of zeek, as the headers include the labeling for the attacks and are not included by default in the zeek http-logs. The next steps of processing and the entire analysis from now on are executed using Python and popular data science libraries, which are mostly available for this programming language. Zeek supports the *zat* library for Python, which allows for converting the zeek log into a data frame for the popular data-science library *pandas*. After converting the zeek logs to a data frame a column gets added called "is_attack". The column holds a flag for the row in the log if it is involved in an attack. The values for "is_attack" get generated, by checking if the "client_header" row includes a header called "X-Is-Attack". Afterwards, the "X-Is-Attack"-header gets cleared from the header columns, as they are not a natural part of the traffic. The resulting matrix contains 34 columns, which carry different kinds of data like the URI of the request, the source and destination IPs, the HTTP method used, etc (compare Table 5.1 for a full list of columns).

uid	origin	orig_fuids
id.orig_h	request_body_len	orig_filenames
id.orig_p	response_body_len	orig_mime_types
id.resp_h	status_code	resp_fuids
id.resp_p	status_msg	resp_filenames
trans_depth	info_code	resp_mime_types
method	info_msg	client_header_names
host	tags	client_header_values
uri	username	server_header_names
referrer	password	server_header_values
version	proxied	is_attack
user_agent		

Tabelle 5.1: List of all features, which represent a column in the network data matrix after the first pre-processing step.

5.2.2 Encoding of URI data using word embeddings

The attacks in this experiment (see Section 5.1.3) are based on tampering with the application layer data by manipulating the HTTP request-line and its query parameters. This data, therefore, is crucial for achieving the goal of detecting anomalies in the network logs. Hence, the previously prepared data frame columns, which hold the relevant data, are the *uri* and the *method* columns. Listing 5.4 now shows the first necessary step for implementing the Word2Vec embeddings, the tokenization. Line two of the listing shows an important decision for the tokenization approach. The URI string is split into multiple parts by isolating every query parameter and separating every value from its query parameter key.

```
1    # Create token input list of all 'uri' and 'method' values for
         the word2vec model
2    tokenized_uri = selected_features_df['uri'].apply(lambda x:
         re.split('[/?&=]', x))
3    tokenized_method = selected_features_df['method'].apply(lambda
         x: [x])
4    all_tokens = tokenized_method + tokenized_uri
```

Listing 5.4: Tokenization of the features for word embeddings

With the now acquired list of token combinations, a Word2Vec model is trained using the *gensim.models* library [Řeh21]. The model's hyperparameters get configured by setting the parameters of the constructor of the Word2Vec class. In this case, shown in Listing 5.5 in line 1, the *min_count* variable is set to *1* because, especially for anomaly detection, it makes sense to include one-time occurrences of words for the model's training, which would be omitted otherwise. The *window_size* defines what distance of words is still considered as the context of a given word. Finding the optimal value for this hyperparameter is the subject of the experiments and is discussed later in the results. The *vector_size* parameter determines the size of the dimension of the word embeddings produced by the model. Higher values produce more nuanced embeddings but also require more computational effort. Similar previously cited research by Zhou et al. [ZZS17] uses a vector size of 300, which is why this number was chosen for the experiments of this work as well. The last parameter, the *workers* parameter, describes the number of CPU cores to be involved in the training in parallel. With the previously generated

tokens and the chosen hyperparameters, a Word2Vec model gets trained, and the embeddings are created and saved to a new column called *embeddings* (see lines 4-9).

```
1           # Create word2vec model
2           word2vec_model = Word2Vec(all_tokens, vector_size=300,
                window=7, min_count=1, workers=4)
3
4           # Utilize the Word2Vec model to apply embeddings to 'uri' and
                'method' values
5           selected_features_df['uri_embedding'] =
                tokenized_uri.apply(lambda x: sum(word2vec_model.wv[t] for
                t in x))
6           selected_features_df['method_embedding'] =
                tokenized_method.apply(lambda x: sum(word2vec_model.wv[t]
                for t in x))
7
8           # Combine embeddings into a single feature for clustering
9           selected_features_df['embedding'] =
                selected_features_df.apply(lambda row:
                row['uri_embedding'] + row['method_embedding'], axis=1)
```

Listing 5.5: Creation of Word2Vec model and word embeddings

5.2.3 Clustering

Two clustering algorithms for anomaly detection are implemented to test the detection rate of attacks in the network logs based on previously generated word embeddings. The first algorithm is the *k-Means clustering algorithm*, and the second is self-organizing maps. For classifying the clusters, a similarly simple approach to that used in the work of Nalavade et al. [NM14] is chosen by defining a threshold for the cluster size, after which a cluster is labeled as an attack cluster.

Write about the implementation of k-Means and Self-Organizing Maps

k-Means

Self-organizing maps

5.3 Results

Write text for the results

- Talk about size and properties of dataset

- Visualize generated dataset

- Talk about window size of Word2Vec

- Visualize Word embeddings

- Add better visualizations for clusters

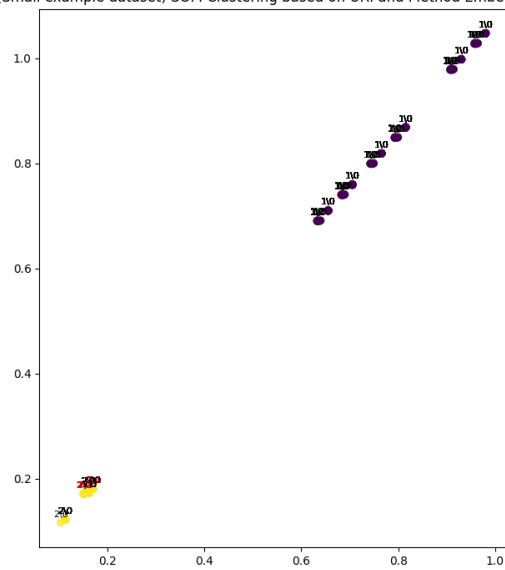- Compare the found Yield and Accuracy

40

Abbildung 5.2: K-Means

Abbildung 5.3: SOM

Use other visualization with greyscale in mind.

6 | 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: ().

[CS18]   Rafael San Miguel Carrasco und Miguel-Angel Sicilia. *Unsupervised intrusion detection through skip-gram models of network behavior*. In: *Computers & Security* 78 (2018), S. 187–197.

[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.

[FOC15]   Eugene Ferry, John O Raw und Kevin Curran. *Security evaluation of the OAuth 2.0 framework*. In: *Information & Computer Security* 23.1 (2015), S. 73–101.

[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).

[HPL23a]   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/.

[HPL23b] 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/.

[Kes14] van Kesteren et al. *Cross-origin resource sharing*. Techn. Ber. World Wide Web Consortium, Jan. 2014. URL: https://www.w3.org/TR/2020/SPSD-cors-20200602/.

[Ket22] James Kettle. *Top 10 web hacking techniques of 2022*. 2022. URL: https://portswigger.net/research/top-10-web-hacking-techniques-of-2022 (besucht am 10.12.2023).

[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.

[Llo82] S. Lloyd. *Least squares quantization in PCM*. In: *IEEE Transactions on Information Theory* 28.2 (1982), S. 129–137. DOI: 10.1109/TIT.1982.1056489.

[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).

[Ltd23] Hsiaoming Ltd. *Authlib: Python Authentication*. 2023. URL: https://docs.authlib.org/en/latest/ (besucht am 09.09.2023).

[MF22] K Meyer zu Selhausen und D Fett. *RFC 9207: OAuth 2.0 Authorization Server Issuer Identification*. 2022.

[Mik+13] Tomas Mikolov u.a. *Efficient estimation of word representations in vector space*. In: *arXiv preprint arXiv:1301.3781* (2013).

[NM14] Kamini Nalavade und Bhushan Meshram. *Evaluation of K-Means Clustering for Effective Intrusion Detection and Prevention in Massive Network Traffic Data*. In: *International Journal of Computer Applications* 96 (Juni 2014), S. 9–14. DOI: 10.5120/16804-6526.

[Pie13] Chris Piech. *K Means*. 2013. URL: https://stanford.edu/~cpiech/cs221/handouts/kmeans.html (besucht am 15.10.2023).

[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.

[Pro23] The Zeek Project. *About zeek*. 2023. URL: https://docs.zeek.org/en/master/about.html (besucht am 09.12.2023).

[Řeh21] Radim Řehůřek. *gensim: topic modelling for humans*. 2021. URL: https://radimrehurek.com/gensim/ (besucht am 09.09.2023).

[Ron14]   Xin Rong. *word2vec parameter learning explained*. In: *arXiv preprint arXiv:1411.2738* (2014).

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

[Sir20]   Prabath Siriwardena. *OAuth 2.0 Security*. In: *Advanced API Security: OAuth 2.0 and Beyond*. Berkeley, CA: Apress, 2020, S. 287–304. ISBN: 978-1-4842-2050-4. DOI: 10.1007/978-1-4842-2050-4_14. URL: https://doi.org/10.1007/978-1-4842-2050-4_14.

[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).

[WH07]   Gregory A. Wilkin und Xiuzhen Huang. *K-Means Clustering Algorithms: Implementation and Comparison*. In: *Second International Multi-Symposiums on Computer and Computational Sciences (IMSCCS 2007)*. 2007, S. 133–136. DOI: 10.1109/IMSCCS.2007.51.

[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.

[ZZS17]   Xiaoyan Zhuo, Jialing Zhang und Seung Woo Son. *Network intrusion detection using word embeddings*. In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, S. 4686–4695.

A | Non encoded tables

Fix sideways table

	Mandatory PKCE	Random value state	Invalidation of access token	Invalidate state value	Simple Str
Insufficient Redirect URI Validation					x
Credential Leakage via Referer Headers	x		x	x	
Credential Leakage via Browser History			x		
Mix-Up Attacks					
Authorization Code Injection	x	x			
Access Token Injection					
Cross-Site Request Forgery	x	x			
PKCE Downgrade Attack	x				
Access Token Leakage at the Resource Server					
307 Redirect					
Client Impersonating Resource Owner					
Authorization Server Redirecting to Phishing Site					
Unvalidated Redirects and Forwards					
Clickjacking					

Eidesstattliche Versicherung

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: 17. Dezember 2023

Muster der Literaturliste

Literaturliste

Todo list