Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

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.

Research task

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.

Abstract

OAuth 2.0 is an authorization framework that has rapidly grown in popularity over the last decade. Whenever internet users click a “Login with...” button on a webpage, they implicitly engage in the OAuth protocol and allow a webpage to access their sometimes confidential data somewhere stored at another service. Therefore, authorization providers and their consuming clients must provide a robust and secure implementation of the OAuth protocol, as well as network security measures to protect their users’ data from attackers. To aid in this responsibility, this thesis presents an overview and classification of common threats and countermeasures when implementing the OAuth 2.0 protocol. This overview shows that flaws in OAuth leading to misuse of the redirection flow of the protocol are among the most impactful threats, and the most sophisticated countermeasures have been established to mitigate them. Based on these observations, this work presents and implements an intrusion detection approach to identify such attacks in a self-generated dataset. The approach utilizes Word2Vec embeddings to encode the HTTP application data of the network traffic into a numerical representation based on their context semantics. These embeddings then get clustered using the k-means and the self-organizing map algorithm to detect anomalies in the network data. The implemented approach resulted in a detection accuracy of 99,3%, with a precision of 63% and a yield of 100%. The results show that the approach generally can sort out the scarce OAuth traffic but has room for improvement in detecting subtle differences inside the OAuth network traffic itself.

Contents

1 Introduction

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 aim to improve the security of the standard itself [**fett2016comprehensive**] by formally proving the security of the standard to unveil flaws. However, the implementation of this complex protocol still could lead to oversights, which could cause vulnerabilities in practice.

A recent 2022 study by Philippaerts et al. shows that 97 of 100 popular OAuth providers leave at least one commonly known OAuth threat unmitigated, with an average of four unmitigated threats per provider [**philippaerts2022oauch**]. This study shows that many OAuth implementations are still flawed, and it did not even include the realization of many thousands 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 security features leads to new emerging threats using other areas of browser feature sets. As demonstrated by the number one of the top ten web hacking techniques of 2022 published by the company PortSwigger [**kettle2022**], newer and more niche attack vectors like the misuse of cross-origin in-browser communication flows could be utilized to leak authorization data as well.

This current situation in the threat landscape of OAuth is not hopeless in any way, more than that, this condition should be the reason that it gets addressed appropriately. Besides the efforts to update the standard to incorporate new security measures, one way to achieve more trust in operating OAuth services 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 intrusion detection techniques, even identifying unknown attack vectors, which is especially useful in an ever-changing threat landscape.

1.1 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.

Building upon the insights gained from 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.

1.2 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 suggestions for classifications 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 the effectiveness of the methods applied as the results of the experiments. It ends with a discussion of the results in the experimental environment.

Chapter 6 ends with a summary of this work and an outlook on future possible research topics.

2 | Fundamental Knowledge

The OAuth 2.0 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 2.0 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 later in this work.

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 [**hammer2010rfc**]. 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 [**denniss2019oauth**] or documents, which describe several security considerations when implementing the protocol in practice [**lodderstedt2020oauth**]. 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 2.1 shows a complete list of all OAuth 2.0. related RFCs released to this date.

RFC	Title
RFC 6749	The OAuth 2.0 Authorization Framework
RFC 6750	The OAuth 2.0 Authorization Framework: Bearer Token Usage
RFC 6755	An IETF URN Sub-Namespace for OAuth
RFC 6819	OAuth 2.0 Threat Model and Security Considerations
RFC 7009	OAuth 2.0 Token Revocation
RFC 7519	JSON Web Token (JWT)
RFC 7521	Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants
RFC 7522	Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants
RFC 7523	JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants
RFC 7591	OAuth 2.0 Dynamic Client Registration Protocol
RFC 7592	OAuth 2.0 Dynamic Client Registration Management Protocol
RFC 7636	Proof Key for Code Exchange by OAuth Public Clients
RFC 7662	OAuth 2.0 Token Introspection
RFC 7800	Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)
RFC 8176	Authentication Method Reference Values
RFC 8252	OAuth 2.0 for Native Apps
RFC 8414	OAuth 2.0 Authorization Server Metadata
RFC 8628	OAuth 2.0 Device Authorization Grant
RFC 8693	OAuth 2.0 Token Exchange
RFC 8705	OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens
RFC 8707	Resource Indicators for OAuth 2.0
RFC 8725	JSON Web Token Best Current Practices
RFC 9068	JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens
RFC 9101	The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)
RFC 9126	OAuth 2.0 Pushed Authorization Requests
RFC 9207	OAuth 2.0 Authorization Server Issuer Identification
RFC 9278	JWK Thumbprint URI
RFC 9396	OAuth 2.0 Rich Authorization Requests
RFC 9449	OAuth 2.0 Demonstrating Proof of Possession (DPoP)
RFC 9470	OAuth 2.0 Step Up Authentication Challenge Protocol

Table 2.1: Overview of all currently existing OAuth 2.0 related RFCs

2.1.1 Involved parties

The OAuth protocol is a very diverse and complex authorization framework, so it makes sense to narrow it down to its core features. Starting with the involved parties in the protocol, meaning the actors, which could be systems involved in interactions or single entities providing authentication data. In general, there are four parties participating 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 access 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 or authorization codes to the client after successful authentication of the resource owner.

Depending on the protocol mode, these four parties exchange different messages in variable ways. In general, these messages can be categorized into two types of messages explained in the following section.

2.1.2 Front-channel and Back-channel messages

Regarding security considerations messages of the OAuth framework can be classified 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* [**fielding1999hypertext**] 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 with multiple connections [**belfaik2022single**]. Front-channel messages introduce more threats as the data transported in the URI could potentially appear in history logs and similar mechanisms. Therefore the preferred way to transport confidential data is via back-channel messages using only one TCP connection between a trusted sender and receiver. Different protocol modes have been developed over time to achieve an adequate OAuth flow for different circumstances.

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. In the following sections, the four most essential grant types are explained.

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 [**philippaerts2022oauch**]. 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 an application, which is allowed to require access to specified protected resources. The redirect URI is the location the user agent is redirected to after authenticating at the authorization server. This in most cases is the URL of the client application. The state can hold any string value and is most commonly used for CSRF tokens, to provide session integrity. After receiving this data from the client, the authorization server now asks the resource owner for authentication through the user agent. 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.

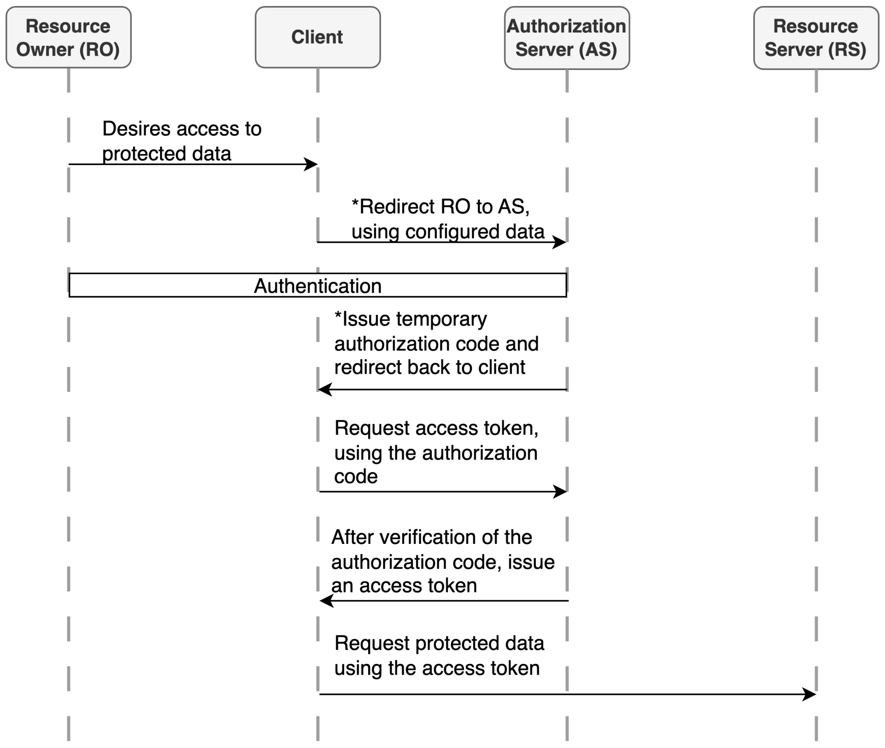

Resource Owner (RO)
Client
Authorization Server (AS)
Resource Server (RS)
Desires access to protected data
*Redirect RO to AS, using configured data
Authentication
*Issue temporary authorization code and redirect back to client
Request access token, using the authorization code
After verification of the authorization code, issue an access token
Request protected data using the access token

Figure 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 [**lodderstedt2020oauth**]. The Implicit Grant is officially deprecated but still has its relevance, as it is still offered by 37% of common identity providers [**philippaerts2022oauch**].

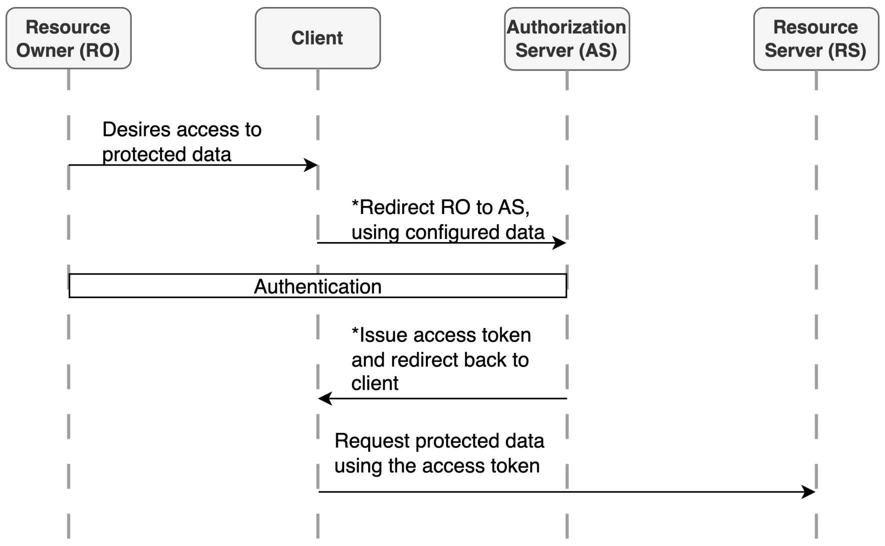

Resource Owner (RO)
Client
Authorization Server (AS)
Resource Server (RS)
Desires access to protected data
*Redirect RO to AS, using configured data
Authentication
*Issue access token and redirect back to client
Request protected data using the access token

Figure 2.2: Implicit Grant

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 then 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 [**hardt2012rfc**].

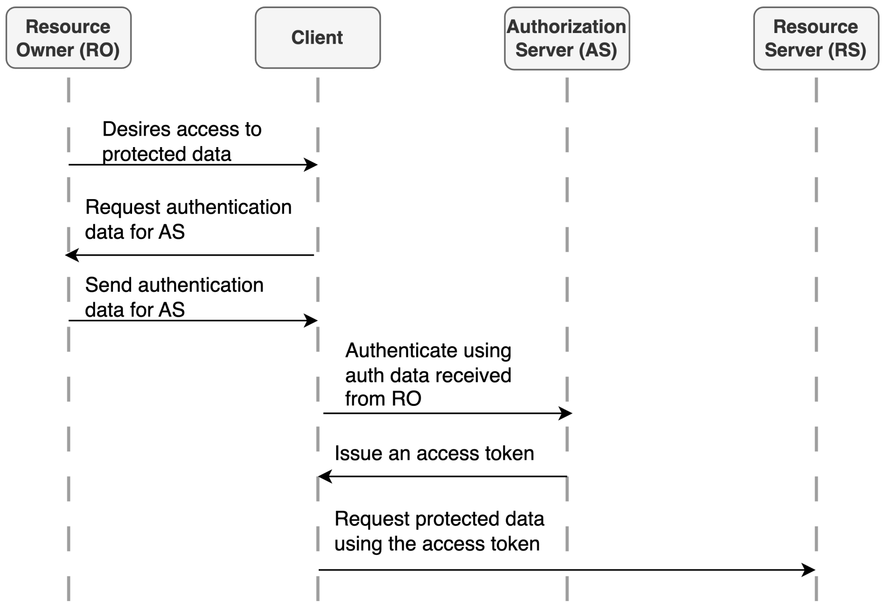

Resource Owner (RO)
Client
Authorization Server (AS)
Resource Server (RS)
Desires access to protected data
Request authentication data for AS
Send authentication data for AS
Authenticate using auth data received from RO
Issue an access token
Request protected data using the access token

Figure 2.3: Resource Owner Password Credentials Grant

Client Credentials Grant

The client credentials grant is designed to 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, which is verified by the possession of the client secret. An example of such a scenario is 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. [**hardt2012rfc**]

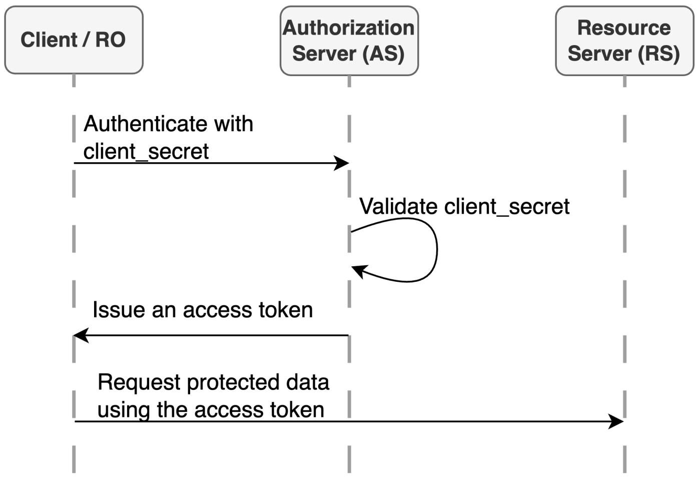

Client / RO
Authorization Server (AS)
Resource Server (RS)
Authenticate with client_secret
Validate client_secret
Issue an access token
Request protected data using the access token

Figure 2.4: Client Credentials Grant

Device Authorization Grant

Introduced in RFC8628 the device authorization grant, illustrated in Figure 2.5 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 [**denniss2019oauth**]. 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.

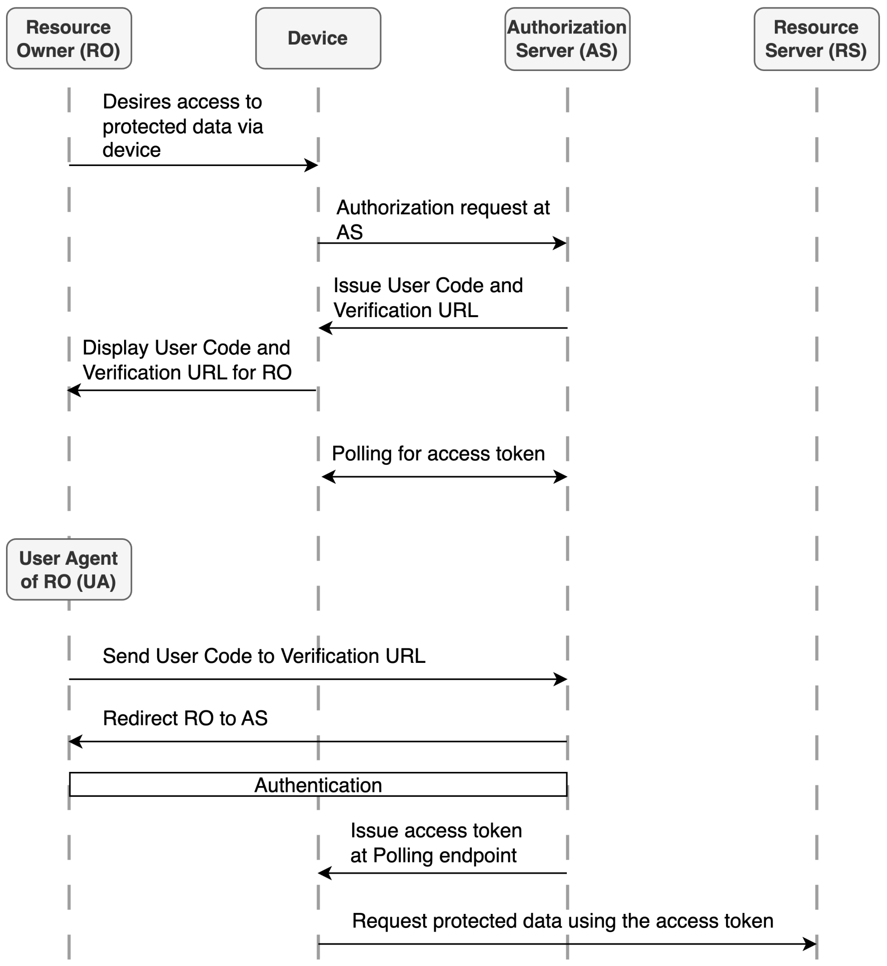

Resource Owner (RO)
Device
Authorization Server (AS)
Resource Server (RS)
Desires access to protected data via device
Authorization request at AS
Issue User Code and Verification URL
Display User Code and Verification URL for RO
Polling for access token
User Agent of RO (UA)
Send User Code to Verification URL
Redirect RO to AS
Authentication
Issue access token at Polling endpoint
Request protected data using the access token

Figure 2.5: Device Authorization Grant

Other grant types

For the purpose of thoroughness, it is to mention, that there are several more grant types, which got introduced to the OAuth standard over time. These additional grant types are listed below, but are out of scope for this work and are not getting explained further:

1. Refresh Token Grant
2. JWT Bearer Grant
3. UMA Grant
4. SAML 2.0 Bearer Grant
5. 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. The authorization of the identified user is a secondary purpose. OIDC flows use a particular OAuth scope (an additional message, besides *state*, *client_id*, and similar), which is called *openid*. OIDC also utilizes an extra token, the *ID* 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 [**li2016analysing**] [**sakimura2014openid**].

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 [**ietf-oauth-v2-1-09**]. 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 were introduced over the last years and allow for more secure interactions. Below is a list of significant changes:

1. The Implicit Grant gets omitted
2. The Resource Owner Password Credentials Grant gets omitted
3. PKCE is required for the Authorization Code Grant
4. Redirect URIs must be compared with exact string matching, so pattern matching is completely disallowed
5. 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 utilizing 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)* [**scarfone2010intrusion**]. There are several taxonomies for intrusion detection system types. For example, Liao et al. [**Liao2013IntrusionDS**] 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. [**khraisat2019survey**] build on top of that approach to further specify an overall taxonomy for IDS. The definition of Intrusion detection systems is based on the taxonomy summarized by these researchers.

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 the usage of 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). These categories can then be combined with the above categories, which are distinguished by input data, depending on the context.

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 behavior. Unusual behavior is identified as an intrusion. There is a variety of techniques that AIDS can use to achieve the goal of differentiating typical and malicious behavior. 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 behavior in a training phase with labeled 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 behavior that breaks those rules as an intrusion. These rules can stem from knowledge about a network protocol or any other system behavior.

2.2.3 zeek IDS

One example of an implementation of an intrusion detection system is called *zeek*.*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" [**zeek2023**] 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 behavior in network traffic. Its architecture consists of two major components, as displayed in Figure 2.6. 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 adapters to popular programming languages also exist and are applied in this work.

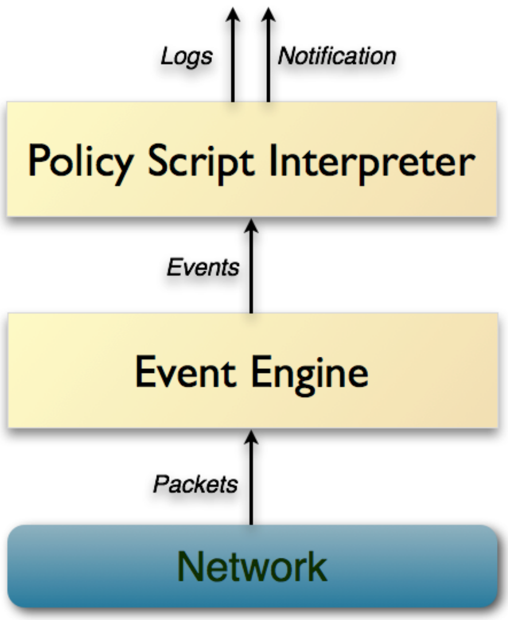


Figure 2.6: Architecture of zeek [**zeek2023**]

2.3 Algorithms

The algorithms applied in the experimental part of this work serve different purposes in the experiment chain described in Chapter 5. These algorithms are essential for following along with the experimental approach. Therefore, this section starts by describing the technique for encoding the textual network data to numerical data. It follows up by illustrating the inner workings of two clustering algorithms applied to the encoded data at the end of the experiments.

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. Two models are employed by *word2vec*, the *Continuous 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 [**mikolov2013efficient**] [**rong2014word2vec**].

Continuous 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-labeled, unsupervised 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 [**wilkin2007kmeans**].

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* $\mu_1, \mu_2 ... \mu_k \in \mathbb{R}$, which are cluster defining centers. The clusters are based on the centroids in the way 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 Algorithm 1. 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: $\arg\min_j \|x_i - \mu_j\|^2$. 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 [**piech2013kmeans**] [**lloyd1982kmeans**]. In the end, every point of the dataset belongs to the cluster centroid with the minimum Euclidean distance.

Algorithm 1: Lloyd's K-Means Algorithm

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

2.3.3 Self-Organizing Maps

A Self-Organizing Map (SOM) is an unsupervised machine-learning algorithm that aims to reduce the dimensions of high-dimensional data. SOMs were introduced in 1990 by the Finnish engineer and senior member of the IEEE Teuvo Kohonen. As described in the initial publication, Self-Organizing Maps not only help with dimension reduction but also can discover semantic relationships in sentences [**kohonen1990self**]. This aspect, in particular, is one reason the algorithm is applied in fields other than data visualization as well, like natural language processing and intrusion detection [**qu2021survey**]. The following explanation is based on the original publication by Kohonen. At its core, a Self-Organizing Map is based on an artificial neural network that is trained using competitive learning. The network's input layer usually has many more dimensions than the output layer, which aligns with the algorithm's goal to reduce the dimensions of data. Then, there is an inner layer called the Kohonen Layer, a grid of neurons with the desired output's dimensions. This layer of neurons gets initialized at random in the beginning. The training process now works as follows:

1. An input vector of the sample high-dimensional data gets randomly chosen. The neuron in the Kohonen layer, whose weight is most similar to the input vector, gets chosen by calculating the Euclidean distance:

   $$D(w,x) = \sqrt{\sum_{i=1}^{N} (w_i - x_i)^2}$$

   for weight vector w, input vector x for N dimensions. The weight vector, which is chosen this way, is called the *winner* or the *Best Matching Unit (BMU)*.

2. A neighborhood function gets applied to the BMU, influencing the weights of neighboring neurons. The neighborhood function is most often a Gaussian decay function:

   $$h(r,t) = \exp\left(-\frac{2\sigma^2(t)}{||r||^2}\right)$$

where $||r||^2$ is the Euclidean distance between the BMU and the neighbor, t is the time (or iteration amount), which has passed and $\sigma^2(t)$ is the neighborhood radius, which decreases over time. This formula leads neurons near the BMU to be updated more heavily in the early stages of the training.

3. The weights of the Kohonen layer are updated for multiple iterations to resemble the input vectors in the end. The output is the weights of the Kohonen layer, which have been reduced to the desired lower dimension.

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 threats, Section 3.1 starts by showcasing important publications in the realm of theoretical security of the OAuth 2.0 standard. It then proceeds to present a recent study on practical implementations of OAuth, which was already mentioned briefly in the introduction

Section 3.2 continues with examples of related work, which utilize similar techniques to the ones applied in this work, but in different contexts or based on different dataset approaches.

3.1 OAuth Security Research

In the area of OAuth security, a lot of research focuses on analyzing implementations of the OAuth standard in practice. Surveys scanning public OAuth providers for vulnerabilities exist [**philippaerts2022oauch**], as well as analysis of OAuth flows using finite state machines [**munonye2022machine**]. However, there are publications by one research group, that focus on the OAuth standard itself, which is presented in Section 3.1.1, because it had an important impact on the OAuth 2.0 standard itself.

3.1.1 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. [**fett2016comprehensive**], 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 modeled 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 modeling 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 modeled, to find the flaws breaking the security proof. Following this methodology, the researchers uncovered four new threats in the OAuth standard, with the impact that the OAuth working group of IETF added these threats and their countermeasures to the official document for security best practices for OAuth [**lodderstedt2020oauth**].

3.1.2 Analysis of OAuth implementations

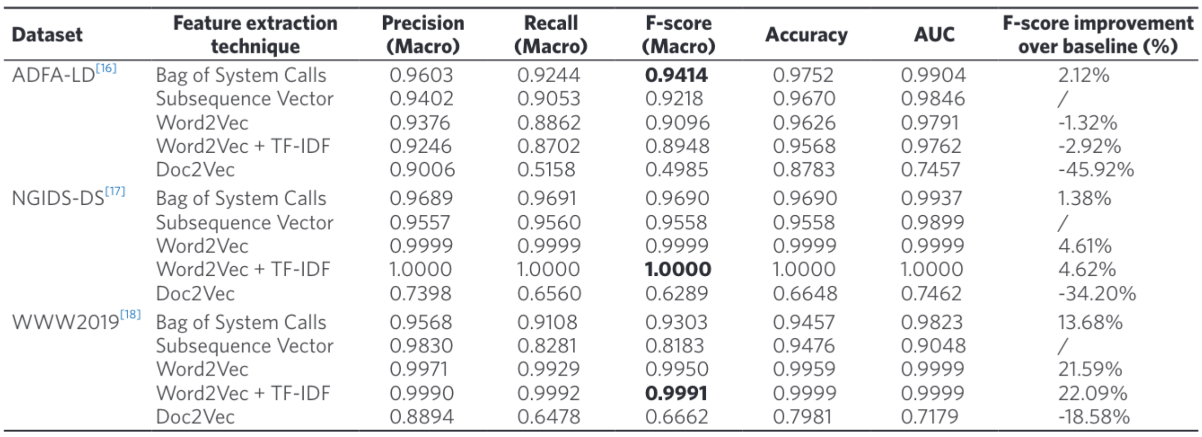
Through research like the one mentioned in Section 3.1.1 by Fett et al. and the continuous work by the OAuth working group of the IETF, the OAuth standard gets enriched with several suggestions for improving the security of OAuth implementations. However, because OAuth is such a complex protocol, implementation 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 [**philippaerts2022oauch**]. This work again shows the importance of awareness of the threat landscape when implementing and operating an OAuth provider or an OAuth client.

3.2 Anomaly-based Intrusion Detection

In the realm of intrusion detection, anomaly-based intrusion detection is a popular research topic, as this type of detection bears one important advantage. Anomaly-based intrusion detection possibly detects unknown attacks in network traffic. Over the last two decades, research in this area has been conducted on benchmark datasets like the *KDD'99* dataset [**kdd1999**] and updated versions of it. Even though this work uses a self-simulated dataset, because of the lack of existing OAuth intrusion network data, techniques applied in such research can be transferred to the scenario of this work. Relevant research for these techniques is showcased in the following sections.

3.2.1 Word embeddings for feature extraction

One issue to overcome, when working with data logs is to extract relevant features and convert them into a form, that is digestible for machine learning algorithms. Especially textual data, which may hold semantic meaning poses a challenge in that regard. The research by Corizzo et al. [**corizzo2020feature**] focuses exactly on tackling this issue, by proposing methods, where raw system calls get encoded by different NLP algorithms. The researchers tested their methods on three different datasets and concluded that an approach combining Word2Vec and TF-IDF performs the best. As visualized in Figure 3.1 even just applying Word2Vec performed in the same area of results as the combination with TF-IDF. Even though the work by Corrizzo et al. concludes that the results are not vast enough, it shows that there is potential in this technique to carry out intrusion detection tasks.

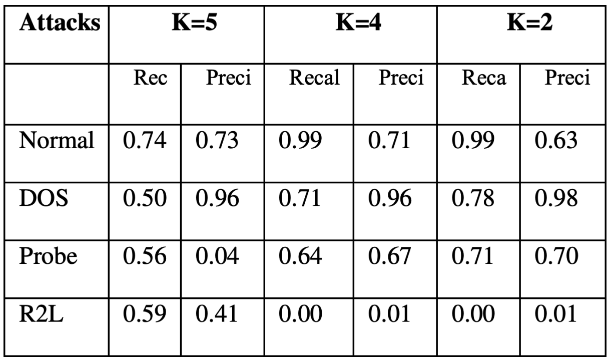


Dataset	Feature extraction technique	Precision (Macro)	Recall (Macro)	F-score (Macro)	Accuracy	AUC	F-score improvement over baseline (%)
ADFA-LD[16]	Bag of System Calls	0.9603	0.9244	**0.9414**	0.9752	0.9904	2.12%
	Subsequence Vector	0.9402	0.9053	0.9218	0.9670	0.9846	/
	Word2Vec	0.9376	0.8862	0.9096	0.9626	0.9791	-1.32%
	Word2Vec + TF-IDF	0.9246	0.8702	0.8948	0.9568	0.9762	-2.92%
	Doc2Vec	0.9006	0.5158	0.4985	0.8783	0.7457	-45.92%
NGIDS-DS[17]	Bag of System Calls	0.9689	0.9691	0.9690	0.9690	0.9937	1.38%
	Subsequence Vector	0.9557	0.9560	0.9558	0.9558	0.9899	/
	Word2Vec	0.9999	0.9999	0.9999	0.9999	0.9999	4.61%
	Word2Vec + TF-IDF	1.0000	1.0000	**1.0000**	1.0000	1.0000	4.62%
	Doc2Vec	0.7398	0.6560	0.6289	0.6648	0.7462	-34.20%
WWW2019[18]	Bag of System Calls	0.9568	0.9108	0.9303	0.9457	0.9823	13.68%
	Subsequence Vector	0.9830	0.8281	0.8183	0.9476	0.9048	/
	Word2Vec	0.9971	0.9929	0.9950	0.9959	0.9999	21.59%
	Word2Vec + TF-IDF	0.9990	0.9992	**0.9991**	0.9999	0.9999	22.09%
	Doc2Vec	0.8894	0.6478	0.6662	0.7981	0.7179	-18.58%

Figure 3.1: Results of applying different NLP techniques for feature extraction on three different datasets of Host-based intrusion data [**corizzo2020feature**]

3.2.2 K-Means and intrusion detection

Clustering algorithms, like k-means, have their place in the anomaly-based intrusion detection research landscape for several years. Earlier research focuses on solely applying these algorithms to filter out anomalous traffic by calculating cluster centroids using k-means and measuring the distance from every datapoint to every centroid to determine outliers for anomaly detection [**munz2007traffic**]. The research by Nalavade et al. [**nalavade2014**] is an example of applying the k-means algorithm for network intrusion detection again using the famous *KDD'99* dataset [**kdd1999**]. The researchers preprocess the data by determining features, that hold more relevance than others and increasing their weights for the clustering, by altering their distance metric during clustering. The results depicted in Figure X show that Navalde et al. had some success detecting certain types of attacks like Denial of Service.



Attacks	K=5		K=4		K=2	
	Rec	Preci	Recal	Preci	Reca	Preci
Normal	0.74	0.73	0.99	0.71	0.99	0.63
DOS	0.50	0.96	0.71	0.96	0.78	0.98
Probe	0.56	0.04	0.64	0.67	0.71	0.70
R2L	0.59	0.41	0.00	0.01	0.00	0.01

Figure 3.2: Results by Navalde et al. applying k-means for different cluster sizes *K* to the KDD'99 dataset after pre-processing the data for intrusion detection [**nalavade2014**]

3.2.3 Self-organizing Maps and intrusion detection

Besides k-means other clustering approaches are frequently researched for intrusion detection. One of these other clustering algorithms is the Self-Organizing Maps algorithm. A survey by Qu et al. [**qu2021survey**] provides a taxonomy of different SOM approaches for network intrusion (see Figure 3.3). The researchers present generally three main categories, where static SOMs are not able to adapt dynamically to new input data, whereas dynamic SOMs have this capability. The hybrid SOMs are a combination of SOMs with any other technique. One example of such a hybrid SOM is the combination of SOM and k-means, which improves the overall performance in intrusion detection on the KDD‘99 dataset (see Figure 3.4) based on the results of different studies gathered by the survey. Figure 3.4 also shows that the results for k-means without SOM are significantly better than the results presented in section 3.2.2. This further emphasizes that the preprocessing of the data is a very important step for intrusion detection.

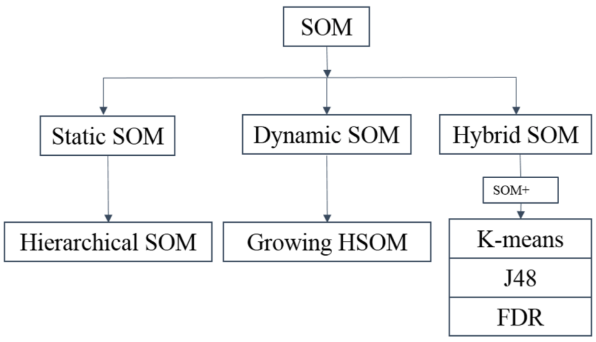
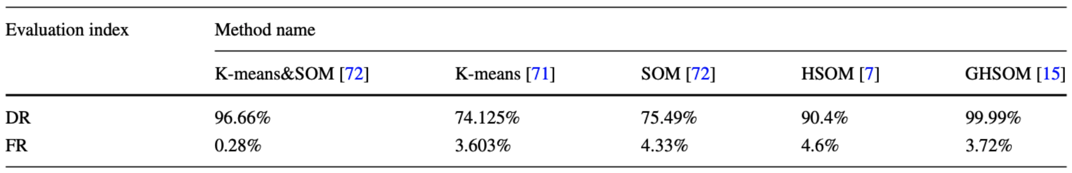


Figure 3.3: Taxonomy of different approaches for implementing Self-Organizing Maps [**qu2021survey**]



Evaluation index	Method name				
	K-means&SOM [72]	K-means [71]	SOM [72]	HSOM [7]	GHSOM [15]
DR	96.66%	74.125%	75.49%	90.4%	99.99%
FR	0.28%	3.603%	4.33%	4.6%	3.72%

Figure 3.4: Results of different research, applying SOM methods for intrusion detection on the KDD‘99 dataset (*DR=detection rate, FR=false positive rate*) [**qu2021survey**]

4 | OAuth Threat Landscape

This chapter presents an overview of the currently known threat landscape when using the OAuth framework. Section 4.1 elaborates on every threat separately. The list is mainly based on the current draft for security best practices for OAuth from the OAuth task force at IETF [**lodderstedt2020oauth**] and is enriched with examples and further explanations.

Section 4.2 showcases different countermeasures, which can be applied to mitigate the presented threats.

Finally Section 4.3 presents two schemas to classify the OAuth threats and countermeasures.

4.1 Threats

Over the 11 years of the existence of OAuth 2.0, there has been always an effort to learn about threats and vulnerabilities, when applying the protocol. When the standard was introduced in 2012 browsers did not have the feature set they have today. This implies on the one hand, that new browser features could lead to new threats and on the other hand that new browser features could help mitigate older threats, which the standard of course could not foresee when it was created. An elaborate source for getting informing about the status of the OAuth threat landscape is the ongoing draft by the OAuth working group of IETF about the best current practice in OAuth security [**lodderstedt2020oauth**]. As mentioned in Section 3.1.1 the draft is influenced by state-of-the-art research about OAuth security in a collaboration effort. This is the reason this section is mostly based on this draft and enriched with examples of other research work at appropriate places. Also every threat explained in this section receives an encoding in the form of *T<number>* to reference them in later tables. These encodings can be found at the end of every paragraph.

4.1.1 Insufficient Redirect URI Validation (T1)

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. [**wang2019make**]. 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 [**lodderstedt2020oauth**].

Authorization Code Flow

Using the example of the authorization code flow the threat of insufficient redirect URI validation can be exploited by crafting arbitrary redirect URIs in the following way:

1. 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.
2. 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.
3. If the user logs in at the authorization server, the authorization code now gets transmitted to the attacker’s webpage, via the redirect URI.
4. The attacker page can now use the received authorization code, to retrieve a token

4.1.2 Credential Leakage via Referer Headers (T2)

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 [**fett2016comprehensive**].

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 [**fett2016comprehensive**].

4.1.3 Credential Leakage via History Logs (T3)

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 [**lodderstedt2020oauth**]. Research about browser history security focuses mainly on accessing information about the victim’s history by comparing cache timings if a page was visited [**bansalcache**]. 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 [**eriksson2022**].

4.1.4 Mix-Up Attack (T4)

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 [**hosseyni2023formal**]. 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. [**fett2016comprehensive**]

4.1.5 Authorization Code Injection (T5)

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 [**philippaerts2022oauch**].

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 (T6)

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. [**lodderstedt2020oauth**]

4.1.7 Cross Site Request Forgery (T7)

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 (T8)

If an authorization server is not implemented to require PKCE for all its flows, it is susceptible to being vulnerable to PKCE downgrade attacks [**philippaerts2022oauch**]. 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 [**hardt2012rfc**].

4.1.9 Access Token Leakage at the Ressource Server (T9)

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 [**lodderstedt2020oauth**].

4.1.10 307 Redirect (T10)

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 [**fielding1999rfc2616**], 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 [**fett2016comprehensive**].

4.1.11 Client Impersonating Resource Owner (T11)

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 [**sakimura2014openid**]. 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 [**lodderstedt2020oauth**].

4.1.12 Authorization Server Redirecting to Phishing Site (T12)

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 [**lodderstedt2020oauth**].

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

Figure 4.1: Phishing request

4.1.13 Unvalidated Redirects and Forwards (T13)

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 [**wang2015urfds**]. 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 [**lodderstedt2020oauth**]. 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 (T14)

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 [**gibbons2014security**].

4.2 Countermeasures

Addressing the issue of the ever-evolving threat landscape of OAuth several countermeasures have been established in a head-to-head race with the exploitations of vulnerable OAuth implementations. The countermeasures profit from the same circumstance of ongoing browser development as the threats. Countermeasures today can leverage new protection features, which were not present in the past. These countermeasures are also well documented by the OAuth working group of IETF [**lodderstedt2020oauth**]. In this section, these common countermeasures are briefly explained. Again they receive an encoding at the end of their section's headline for later reference.

4.2.1 Mandatory PKCE (C1)

As an extension to OAuth defined by RFC 7636, *Proof Key for Code Exchange (PKCE)* is a technique to mitigate several OAuth threats [**bradley2015rfc**]. 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 [**siriwardena_oauth_2020**]:

1. Initially, before the client redirects the resource owners̋ 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.

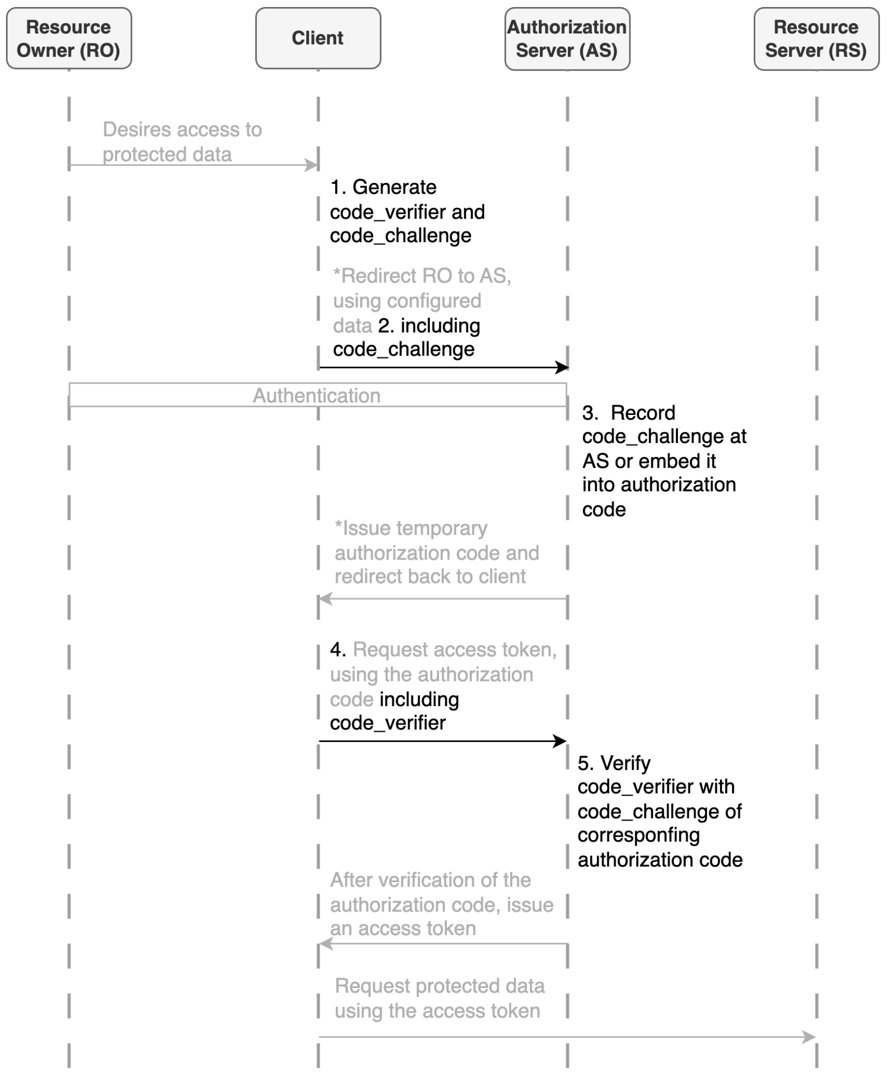

Resource Owner (RO)
Client
Authorization Server (AS)
Resource Server (RS)
Desires access to protected data
1. Generate code_verifier and code_challenge
*Redirect RO to AS, using configured data 2. including code_challenge
Authentication
3. Record code_challenge at AS or embed it into authorization code
*Issue temporary authorization code and redirect back to client
4. Request access token, using the authorization code including code_verifier
5. Verify code_verifier with code_challenge of corresponfing authorization code
After verification of the authorization code, issue an access token
Request protected data using the access token

Figure 4.2: Authorization Code Grant with extension for Proof Key for Code Exchange

4.2.2 Random value state (C2)

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 [**ferry2015security**]. When PKCE for the authorization code grant is already implemented, this measure does not add any benefit regarding CSRF protection [**bradley2015rfc**]. 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 (C3)

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 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 [**hardt2012rfc**]. 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) [**lodderstedt2020oauth**], as for these threats, exposed authorization codes were already redeemed in most cases.

Auth code injection? Check table Mitigation responsibility

4.2.4 Simple string comparision (C4)

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 [**lodderstedt2020oauth**].

4.2.5 Avoid usage of grant types (C5)

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 [**lodderstedt2020oauth**]. For these reasons, the OAuth task force removed the implicit grant in the draft for OAuth 2.1 [**hardt2023rfc**]. 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 [**vanKesteren2014**]. As the OAuth standard dates back to 2012 [**hardt2012rfc**], the implicit grant could be implemented for applications that did not support CORS yet.

4.2.6 Constrained access tokens

Make clear misuse of stolen access token

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 [**lodderstedt2020oauth**]:

- *Sender-constrained access token (C6)*: 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 (C7)*: 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 (C8)

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 [**meyer2022rfc**].

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. [**lodderstedt2020oauth**] and the classification of which involved party is responsible for implementing the measure.

- *303 redirect (C9)*: 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 (C10)*: 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 (C11)*: 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 (C12)*: 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, even when the authorization code grant is in use. 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 (C13)*: 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 (C14)*: 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 (C16), open redirect (C15) 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 4.1 and 4.2. 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 Threats and their countermeasures

The first classification displayed in Table 4.1 is visualizing, which countermeasure mitigates aspects of which threat when using OAuth. The table shows that most countermeasures are specific to single threats. However, there are two countermeasures, which mitigate at least three threats. These countermeasures are mandatory PKCE and avoiding the usage of specific grant types. Looking at the problems these two countermeasures solve, one can identify the two main weaknesses of the OAuth 2.0 standard. The first one being grant types like the implicit grant, which communicate confidential messages via the front channel. The second weakness being the session integrity of the protocol, as multiple connections are involved via redirects. Both these countermeasures will be enforced by the OAuth 2.1 standard [**hardt2023rfc**], which makes conforming implementations potentially a lot more secure in the future.

	Mandatory PKCE	**Random value state**	**Invalidation of access tokens**	**Simple string comparision**	**Avoid usage of grant types**	**Sender-constrained access token**	**Audience-constrained access token**	**Issuer identification**	**303 redirect**	**Client_ID not choosable by user**	**No redirect before authentication at AS**	**No access token in URI**	**Avoid third-party content**	**Appropriate referer policy**	**Open redirect countermeasure**	**Clickjacking countermeasure**
T1				x											x	
T2	x		x		x								x	x		
T3			x		x							x				
T4								x								
T5	x	x														
T6					x											
T7	x	x														
T8	x															
T9						x	x									
T10									x							
T11										x						
T12											x					
T13															x	
T14																x

Table 4.1: Classification of which countermeasure mitigates which threat to OAuth. *The encoding T1-T14 corresponds to the threats described in Section 4.1.*

4.3.2 Threats by mitigation responsibility

Another way to put the threat landscape into perspective is by categorizing the different threats by the entities, which are responsible for mitigation. Depicted in Table 4.2 it is visible, that most threats need to be mitigated by the client and the authorization server together. This is not surprising as one goal of OAuth is to provide a modular authorization workflow, so the resource server for example only needs to provide a token verification ability for the protocol. Important to note is that most countermeasures are enforced by the authorization server as the client is more or less a consumer of the authorization service. If an authorization server decides to enforce countermeasures like PKCE (C1) a client has to adapt to this requirement. Therefore even if many of the countermeasures need to be realized by client and authorization server together, in practice the authorization server bears the main responsibility for fulfilling mitigation efforts. An additional aspect of the table to consider is that there are still threat aspects, that only the client can handle. The general web security threats like unvalidated redirects and forwards (open redirects) and clickjacking are threats that could be chained together with other OAuth threats for exploitation. These are examples of dangers the authorization server cannot influence.

Threat	*Client*	*Authorization Server*	*Resource Server*
Insufficient Redirect URI Validation		C4	
Credential Leakage via Referer Headers	C1, C5, C12, C13, C14	C1, C3, C5, C6, C7, C12, C13, C14	
Credential Leakage via History Logs	C5, C12	C3, C5, C12	
Mix-Up Attacks	C8	C8	
Authorization Code Injection	C1	C1, C3	
Access Token Injection	C5	C5	
Cross-site Request Forgery	C1, C2	C1, C2	
PKCE Downgrade Attacks	C1	C1	
Access Token Leakage at the Ressource Server	C6, C7	C6, C7	C6, C7
307 Redirect		C9	
Client impersonating resource owner		C10	
Authorization Server Redirecting to Phishing Site		C11	
Unvalidated Redirects and Forwards	C15		
Clickjacking	C16	C16	

Table 4.2: Classification of threats by mitigation responsibility. *The encoding C1-C16 corresponds to the countermeasures described in Section 4.2.*

4.3.3 Threats by properties in common

Another way for classification proposed by this work is to sort the different threats by common properties to determine threat categories and possibly identify more impactful groups of threats. One possible categorization is portrayed in Figure X:

1. *Credential leakage and session hijacking*: Threats in common that through different web technologies, like HTTP, HTML, and browser features such as history logs and certain implementation flaws, the credentials of a user, or an active session could get leaked to an adversary. These threats must be mitigated by different mechanisms these web technologies offer, like strict content security policies, adequate referer header policies and generally careful implementation decisions.

2. *Redirection tampering*: Threats that are characterized by the possibility that an adversary tampers with the redirection flow of an OAuth grant. These threats are often bound to flaws in the implementation of OAuth by the client and the authorization server, combined with phishing. These threats are among the most impactful threats as they allow for an

attacker to completely impersonate the victim, by just crafting a malicious URL and making the victim click on it.

3. *Breaking of protocol integrity*: Threats for mechanisms to ensure session integrity like CSRF tokens in the state parameter or PKCE. These measures could have a flawed implementation or could be optional leading to the possibility an adversary could circumvent them.
4. *Credential Injection*: If credentials are already leaked, one category of threats is the threat of the injection of these credentials into the protocol. Certain measures can be introduced to even mitigate the injection at this point, by one-time tokens connected with invalidation of used tokens, or short timespans for redeeming access tokens for authorization codes.

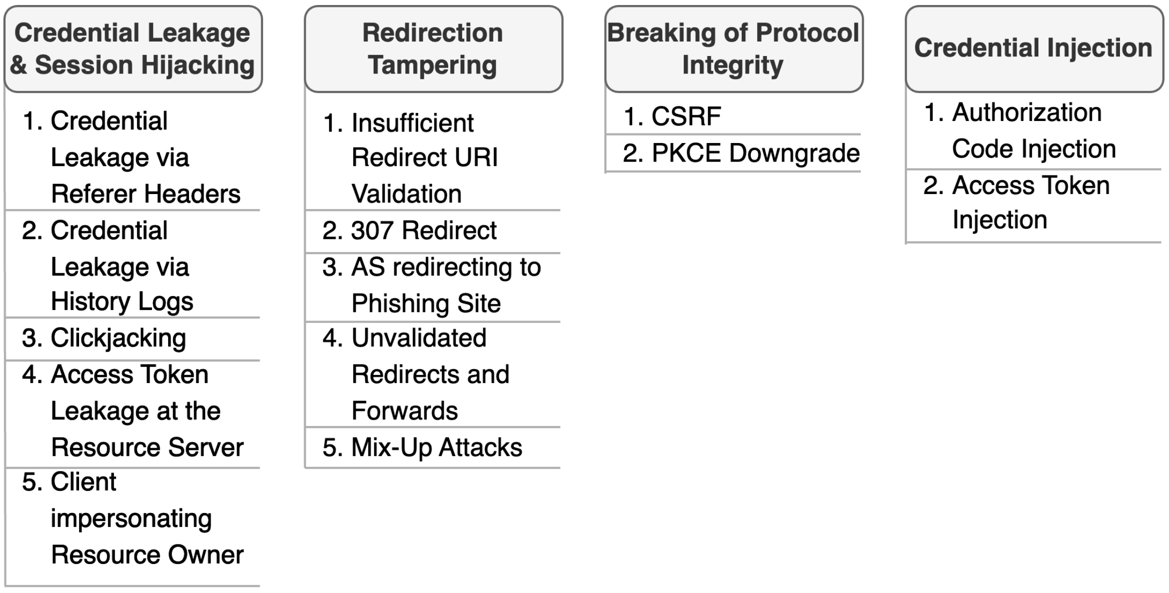

Credential Leakage & Session Hijacking
1. Credential Leakage via Referer Headers
2. Credential Leakage via History Logs
3. Clickjacking
4. Access Token Leakage at the Resource Server
5. Client impersonating Resource Owner
Redirection Tampering
1. Insufficient Redirect URI Validation
2. 307 Redirect
3. AS redirecting to Phishing Site
4. Unvalidated Redirects and Forwards
5. Mix-Up Attacks
Breaking of Protocol Integrity
1. CSRF
2. PKCE Downgrade
Credential Injection
1. Authorization Code Injection
2. Access Token Injection

Figure 4.3: Threats categorized by properties in common

Following along with this categorization, the *Redirection Tampering* category was chosen for further analysis in the intrusion detection experiment of this work, as the potential vulnerabilities that are involved with this threat category are directly tied to flaws in OAuth implementations instead of problems with other web technologies. In addition, these threats are potentially highly impactful with minimal effort from the adversary.

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.

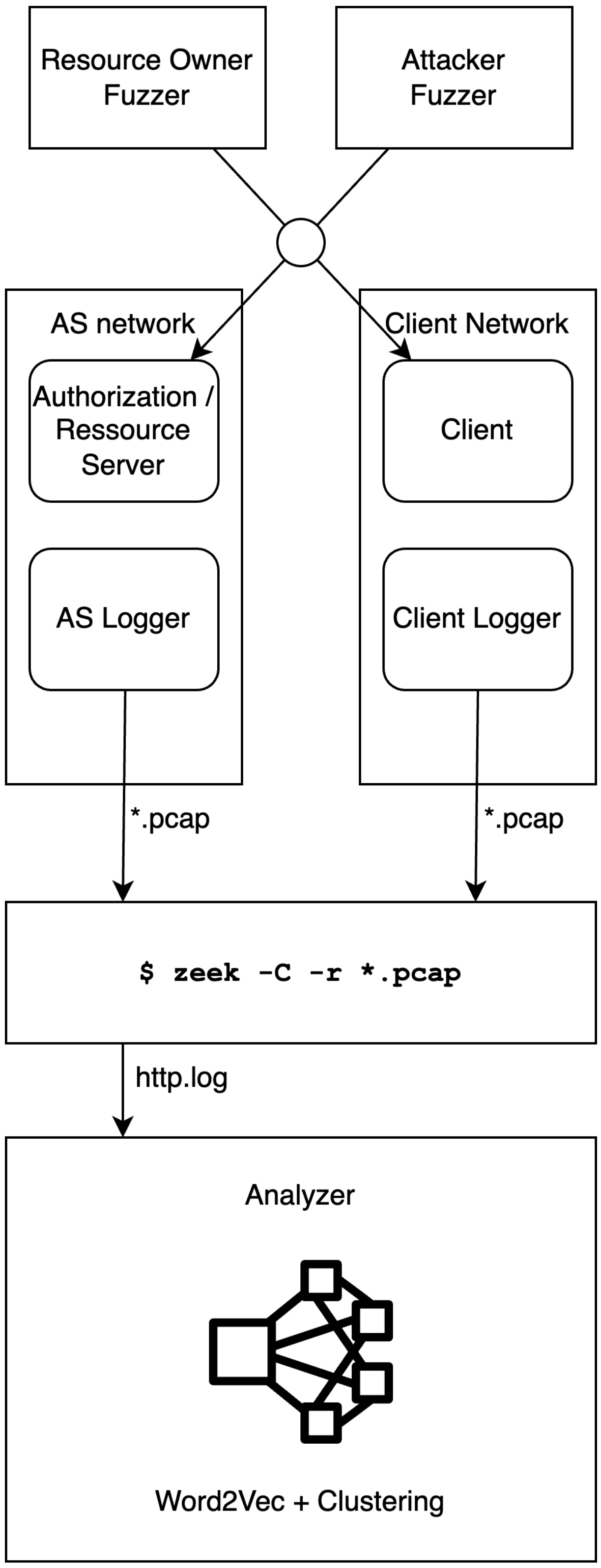


Figure 5.1: Overview of the environment for the experiments

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 [**authlib2023**]. 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
// fetch access token on page load, if the code query param is present
window.onload = function() {
    const params = new URLSearchParams(window.location.search);
    const client_id = "<pre-configured client ID>"

    if (params.has("code")) {
        const form = new FormData();
        form.append('grant_type', 'authorization_code');
        form.append('scope', '<some scope>');
        form.append('code', params.get("code"));
        form.append('client_id', client_id);
        fetch('<token endpoint of authorization server>', {
            method: 'POST',
            body: form
        })
    }
}
```

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.

```
$ 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).

```
 1  class OAuthCallback(BaseHTTPRequestHandler):
 2      def do_GET(self):
 3          parsed_url = urlparse(self.path)
 4          query_params = parse_qs(parsed_url.query)
 5
 6          # Abort if there is no 'code' parameter present.
 7          assert query_params["code"] is not None
 8
 9          # Fetch access token with code
10          auth_code=query_params["code"]
11          data = {"grant_type": "authorization_code",
12                  "code": auth_code,
13                  "redirect_uri": REDIRECT_URI,
14                  "client_id": CLIENT_ID,
15                  "client_secret": ""}
16          res = r.post(f"{AUTHORIZATION_SERVER_URL}/oauth/token",
                data=data)
17          access_token_data = res.json()
18
19          # Utilize stolen access token
20          headers = {"Authorization": f"Bearer
                {access_token_data['access_token']}"}
21          res = r.get(f"{RESOURCE_SERVER_URL}/api/me", headers=headers)
22          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* [**wang2019make**] 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 approaches

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 [**carrasco2018unsupervised**] [**zhuo2017network**], 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

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		

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

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).

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 [**gensim2021**]. 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. [**zhuo2017network**] 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. [**nalavade2014**] is chosen by defining a threshold for the cluster size, after which a cluster is labeled as an attack cluster. The performance metrics are described in detail in Section 5.2.4.

k-means

The first approach chosen to cluster the records is the k-means clustering algorithm, a popular clustering method in the intrusion detection research space, as presented in Section 3.2.2 and Figure 3.4. The *scikit-learn* [**scikitlearn2011**] library was utilized for the k-means clustering approach in the experiment environment as it implements Lloyd's k-mean algorithm compatible with the *pandas* data frames. Using the prepared embeddings, the implementation of k-means boils down to two lines of code as depicted in Listing 5.6. Line 1 shows the initialization of the clustering algorithm given the number of clusters it should calculate and an optional *random_state* parameter, which gets utilized to initialize the pseudo-random number generator, which is responsible for calculating the randomness for the first set of cluster centroids. Setting this parameter makes it possible to reproduce the experiment results precisely. Line 2 shows the execution of the algorithm on the previously initialized *kmeans* object. A new column gets added to the data frame, that includes the clustering features, which holds the result of the clustering for every record.

```
1           kmeans = KMeans(n_clusters=5, random_state=10)
2           zeek_log_df['cluster'] =
               kmeans.fit_predict(selected_features_df['embedding'].tolist())
```

Listing 5.6: Application of k-means clustering on the previously calculated embeddings

Self-organizing maps

The second approach implemented for clustering is the self-organizing maps algorithm. Again, a *pandas* and *numpy* compatible library was chosen for integrating this approach into the experiment environment. The selected *MiniSom* library [**vettigliminisom**] offers a minimalistic implementation of self-organizing maps. With the help of this library, the implementation showcased in Listing 5.7 comes down to the initialization of the SOM algorithm in Line 1, where multiple parameters are provided:

1. *x* and *y*: The number of neurons of the SOM grid initialized at the x-axis and the y-axis.
2. *input_len*: The dimensionality of the input vectors, given by the dimensionality of the word embedding vectors.
3. *sigma*: The radius of influence in the neighborhood, when a BMU gets updated with the neighborhood function as explained in Section 2.3.3.
4. *learning_rate*: The order of magnitude for the update of the weights during the learning process. A higher value results in more impact at the training, but only initially, as the update function is a decay function.
5. *random_seed*: The seed for the pseudo-random number generator, which is responsible for the randomness when picking the next input vector for every training epoch. Setting this parameter leads to reproducible experiment results.

Line 2 applies the algorithm utilizing the previously initialized *som* class to train the weights for the SOM grid. The parameter *num_iterations* is provided to set the number of training iterations. The results are saved by overwriting the *embedding*-column. Finally, Line 4 assigns clusters to every record by using the x-coordinate of the BMU representing a cluster and writing it to a new *cluster*-column.

```
        som = MiniSom(x=3, y=3, input_len=300, sigma=0.8,
           learning_rate=0.2, random_seed=10)
        som_weights =
           som.train_batch(selected_features_df['embedding'].tolist(),
           num_iteration=100)

        zeek_log_df['cluster'] = [som.winner(x)[0] for x in
           selected_features_df['embedding'].tolist()]
```

Listing 5.7: Application of the SOM clustering approach on the previously calculated embeddings

5.2.4 Performance metrics

Several standard performance metrics are employed on the clustering results to evaluate the effectiveness of the intrusion detection approaches. The key performance metrics considered in this work are *yield*, *accuracy* and *precision*. These key metrics are derived from four more general measures:

1. *True Positive (TP)*: An attack record correctly identified as an attack.
2. *True Negative (TN)*: A non-attack record correctly identified as a non-attack-record.
3. *False Positive (FP)*: A non-attack record incorrectly identified as an attack record.
4. *False Negative (FN)*: An attack record incorrectly identified as a non-attack record.

Based on these measures, the yield, accuracy, and precision are calculated in the following way:

1. $\text{Yield} = \frac{\text{TP}}{\text{Total amount of attack records in dataset}}$
2. $\text{Accuracy} = \frac{\text{TP + TN}}{\text{Total amount of records in dataset}}$
3. $\text{Precision} = \frac{\text{TP}}{\text{TP + FP}}$

The yield evaluates how many records are identified as an attack out of all attack records. The accuracy gives an overview of the overall performance of the intrusion detection approach by factoring in the true positives and true negatives. The precision shows how many records detected as attacks actually are attack records. This measure, in particular, bears much relevance for practical use cases as every record labeled as an attack record could mean that work resources must be invested into handling them.

5.3 Results

This section describes the results of the dataset generation and intrusion detection approaches conducted in this work.

5.3.1 Dataset generation

The resulting dataset, as described in Section 5.2.1, consists of records representing the HTTP network traffic over a fixed period of time. For this analysis, a dataset containing 3627 records is generated containing 44 records, which are requests involved in the attacks on the OAuth protocol. Thus, the resulting attack rate in this generated dataset amounts to 1,12%. Table 5.2 shows an example of an attack record, while Table 5.3 shows examples of non-attack records.

Method	URI	IsAttack
GET	/oauth/authorize ?response_type=code &client_id=A30qfzW7fbvhAN4OtIq4ZNFR &redirect_uri=https://evil-server.com	1

Table 5.2: Representation of a record of the dataset containing a request involved in an attack

Method	URI	IsAttack
GET	/?next=http://localhost:5123/oauth/authorize ?response_type=code &client_id=A30qfzW7fbvhAN4OtIq4ZNFR &redirect_uri=http://localhost:8080/index.html	0
GET	/	0
POST	/	0
POST	create_client	0

Table 5.3: Representation of records of the dataset containing requests involved in usual traffic

5.3.2 Word2Vec embeddings

The tuning of the main hyperparameter for the word2vec embeddings, namely the window size, led to the result that experimenting with it did not influence the performance of the intrusion detection for both clustering algorithms in this environment of generated data. This circumstance could result from the short repetitive strings given to the algorithm as input, so extending the window size to include more than one neighboring word does not make a difference. The result of the embeddings is showcased in Figure 5.2 normalized to three dimensions. It is visible that one large group of records exists with several small outlier groups with a relatively large distance to the cluster. This observation shows that the encoding using word2vec embeddings profoundly impacts the results, as there are already visually distinguishable clusters of records, which need to be identified by the clustering algorithms in the next step.

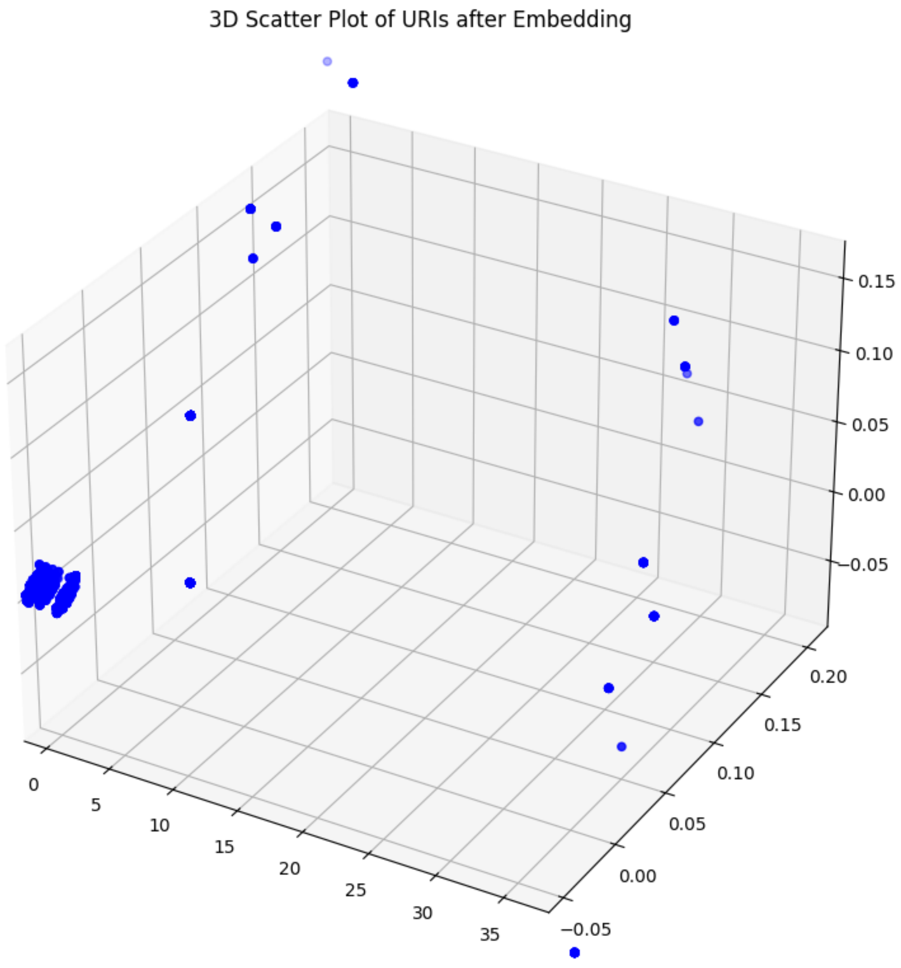


Figure 5.2: Embedding Vectors normalized to three dimensions. Every point in the plot represents one record of the dataset. The word2vec embedding approach leads to a clear group and some sets of outliers.

5.3.3 Overall anomaly detection results

The overall results after clustering are showcased in Table 5.4. Both clustering algorithms achieve the same results with a high accuracy of >99% but, in relation, only attain a low precision score after the word2vec embedding. The main observation which leads to these results is that the implemented approaches perform well in filtering out traffic not involved in OAuth but do not achieve the same quality of detection in more subtle differences in the OAuth traffic itself. Also, achieving the same results for both clustering algorithms could mean that the main impact for anomaly detection in this scenario lies in the encoding step with word2vec.

These observations are discussed further in Section 5.3.4 after the following sections explain how the clustering algorithms achieve their results.

	k-means	**Self-Organizing Maps**
Accuracy	0.993	0.993
Yield	1.0	1.0
Precision	0.637	0.637

Table 5.4: Results of Intrusion Detection using Word2Vec and Clustering algorithms

Anomaly detection with k-means

The first observation on the k-means clustering results is the behaviour when setting different cluster sizes for the algorithm for an anomaly threshold of 5%. The results show that after slowly increasing the number of clusters k-means calculates, a decay of accuracy and precision appears after 7 clusters as depicted in Table 5.5. The reason is that k-means starts to split larger groups of records into clusters small enough to be detected as anomalous at some point and many of these larger groups are non-attack traffic. As depicted in Figure 5.3, when only two clusters are calculated, a large group of non-attack traffic records exist near the origin, and some smaller outlier groups, including attack traffic, appear further away from the origin. Figure 5.4 shows that k-means starts to split the large cluster of non-attack traffic near the origin into smaller clusters when the cluster number is set to 6. Experimenting with the threshold on a higher number of clusters also leads to the highest achievable precision of 0,637%, for example, when setting the number of clusters to 24 and the threshold to 0,39%.

k-means Clusters	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**17**	**20**	**24**
Anomalous Clusters	1	2	2	2	3	4	5	10	12	18
Accuracy	0,99	0,99	0,99	0,99	0,99	0,99	0,97	0,92	0,89	0.71
Precision	0,63	0,63	0,63	0,63	0,63	0,63	0,34	0,13	0,11	0,05

Table 5.5: Setting a higher number of k-means clusters to calculate leads to decay in accuracy and precision starting at 8 clusters with an anomaly threshold of 5%

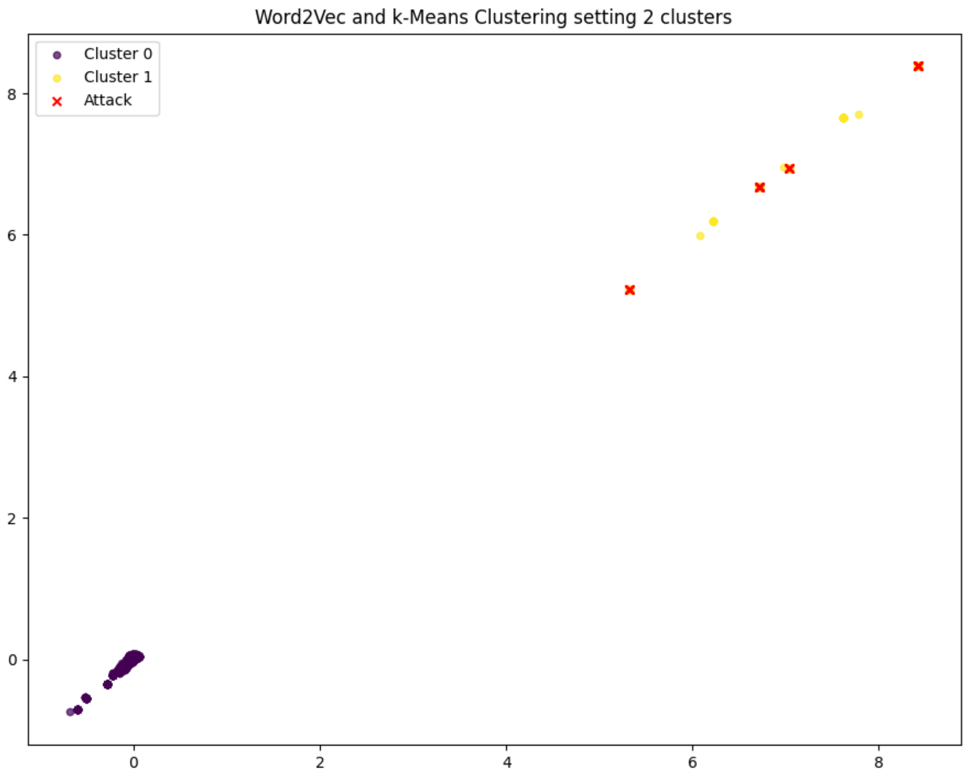


Figure 5.3: Word2Vec and k-means setting 2 clusters

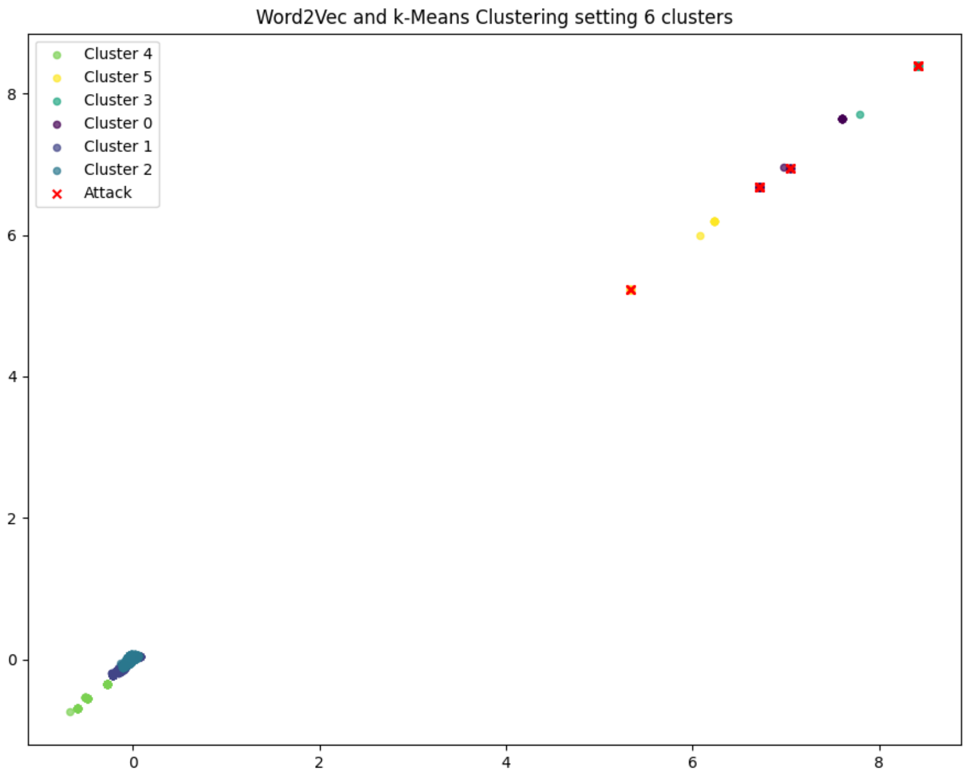


Figure 5.4: Word2Vec and k-means setting 6 clusters

Anomaly detection with self-organizing maps

The self-organizing maps clustering approach leads to the same results as the k-means clustering algorithm. Figure 5.5 shows that the algorithm creates three clusters, one of which is an attack cluster. In the figure, it is also visible that SOM is splitting the main group of records near the origin, a non-attack traffic group, into two clusters that are large enough to be considered non-anomalous. Tuning the hyperparameters only leads to the result that fewer iterations of training are needed to achieve the same best accuracy and precision.

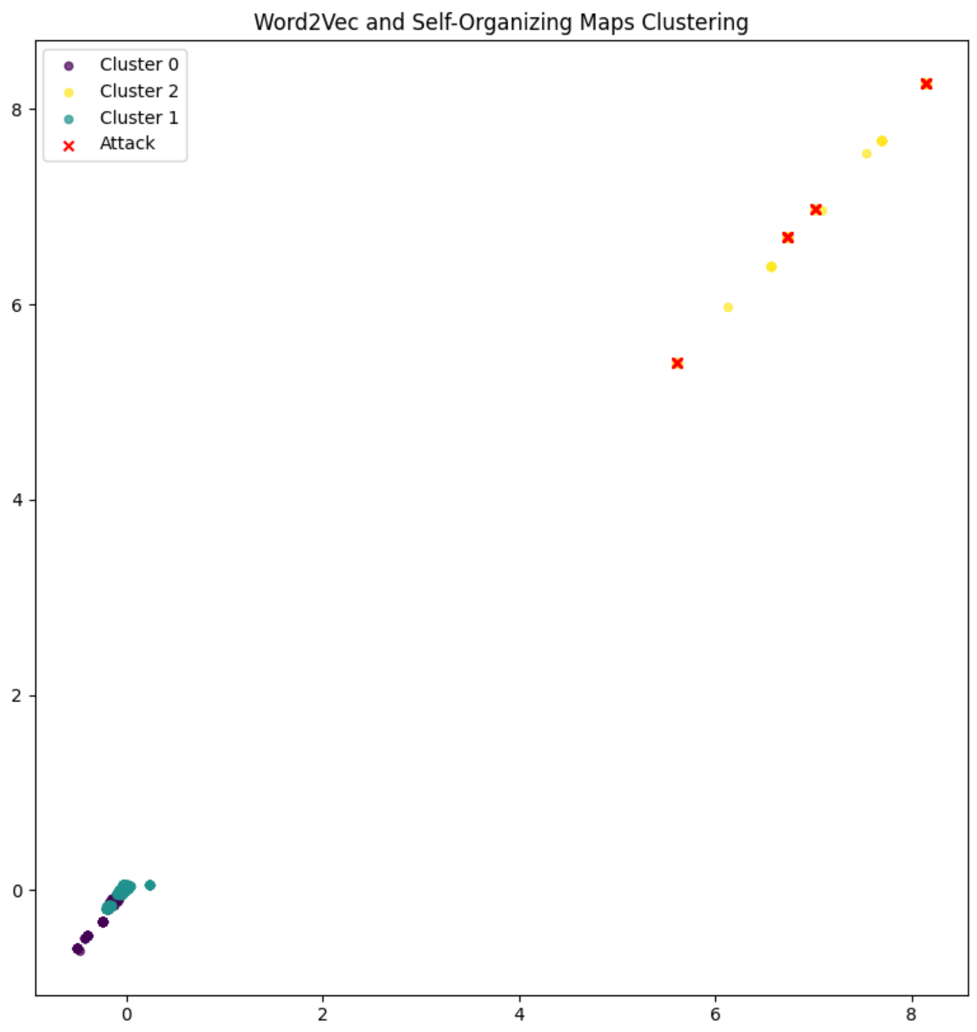


Figure 5.5: Word2Vec and Self-Organizing Maps

5.3.4 Discussion of methodology and results

The results clearly show that the embedding approach greatly impacts the results in the given scenario. The scenario itself, however, is artificial and created through simulating traffic. This approach is arguably too sparse, but a dataset containing specific attacks on the OAuth protocol was not available at the time of this writing. Hence, this model for intrusion detection was created for this work. The dataset simulates real-world HTTP packet data using valid implementations of the OAuth protocol. It does not model metadata of the network traffic, as it does not model real-world timings and flow data of different actors and attackers accessing a network. Consequently, this research focuses on analyzing the semantics of the involved HTTP application data instead of the states and transitions of the protocol flow.

The attack type integrated into the dataset was chosen because attacks on OAuth manipulating the redirect flow are highly impactful concerning the takeover of accounts and extracting confidential data. Attacks involving multiple auth providers, like Mix-Up attacks, were not included as detecting these kinds of attacks would require sophisticated data about the timings and transitions of the network data, which is not provided by the modelled dataset and could only realistically be provided by a real-world network, which was not available for the scope of this work.

The encoding of the application data, based on previous research by Corizzo et al. [**corizzo2020feature**], is a highly promising approach for extracting features for network data, which is why it was applied in this work. However, the low diversity of attacks outside the OAuth traffic influences the performance of this approach in this modeled data. Several classifier methods were tested to improve the anomaly detection performance before choosing the clustering approach for this work. Supervised learning classifiers like support vector machines were tested but not further researched as the narrow model of network data quickly led to overfitting and a classifier that would have a 100% precision rate inside the modeled data. Consequently, researching clustering methods with word embeddings was the most promising approach to detecting anomalies, meaning detecting even unknown attacks outside the modeled dataset.

As protocols for the internet like OAuth 2.0 grow in complexity over time, the approach of detecting anomalies in the application data of the protocol gains importance as well since some types of attacks exploit subtle logical implementation flaws, like the redirect manipulation attacks implemented in this work. Therefore, this research bears relevance for future methods in detecting anomalies in application data, even though the dataset is artificially generated.

6 Conclusion

The chapter concludes the thesis by summarising the content presented in the previous chapters in Section 6.1 and providing an outlook to future research in the area of anomaly intrusion detection in application layer data in Section 6.2.

6.1 Summary

This thesis presented an overview of the OAuth threat landscape and showcased a methodology for anomaly intrusion detection analyzing semantics in HTTP application data on the example of OAuth protocol attacks.

The work within this thesis was motivated by the profound impact vulnerabilities in authorization systems could bear and recent analysis that shows that for many authorization providers, known mitigation measures still needed to be implemented.

Initially, a baseline of fundamental knowledge has been established in Chapter 2. It included a broad overview of the OAuth 2.0 protocol, its different protocol modes, and an outlook on the future of OAuth 2.1. Furthermore, the chapter had an insight into intrusion detection and the involved algorithms for the approach implemented in this work for the detection of attacks on OAuth.

To further support the initial motivation with data, Chapter 3 3 presented related research in the realm of OAuth security research and the intrusion detection techniques applied in this work's experiments.

In Chapter 4, the work proceeded to present the main known threats when utilizing the OAuth 2.0 authorization framework and the countermeasures to mitigate them. It put these threats into different perspectives to further emphasize the responsibilities of authorization providers and clients implementing the protocol and to work out the main weaknesses of the protocol.

Building on top of the knowledge gained on the main threats for the OAuth 2.0 protocol, this work presented and implemented an anomaly intrusion detection approach to help handle possible attacks an authorization provider could face. This implementation also included a complete network environment to generate OAuth interactions and attacks on OAuth to record network logs of these events. Chapter 5 began by explaining the full implementation of the environment and the methodology employed in this work for anomaly intrusion detection. This intrusion detection approach was based on identifying semantics in HTTP application data using word2vec embeddings, which convert textual data, which commonly stands together into vectors that point in similar directions. Two clustering approaches were applied to determine the anomalies in these word2vec representations. The methods used led to the observation that they both depend on the quality of the embeddings, which the word2vec algorithm creates beforehand, as both clustering algorithms produced the same results. Since the implemented approach was efficient in singling out sparse OAuth traffic but was not able to determine subtle anomalies in the OAuth traffic itself to detect the attacks, it is clear that this approach needs further research in certain aspects formulated in the next section.

6.2 Future Work

The main contribution to this area of research in the future would be the creation of a real-world dataset that includes attacks on OAuth. There are several methods to achieve that kind of goal. One would be that staged attacks are executed and logged in a real network. Even an attempt of an attack could produce valuable data. The ideal scenario, however, would be acquiring data from a real-world network where verifiably flaws in the OAuth implementation were exploited to gain access to confidential data. This data could then be utilized to test the approach presented by this work in a real-world scenario.

Another area where the research could be continued is testing further encoding methods of HTTP application data for feature extraction. The results produced by the Word2Vec embeddings left room for improvement. Hence, other textual data encoding methods may also be researched. On top of that, further classification methods could be employed, like supervised learning approaches on more diverse datasets, including OAuth attacks.

Ultimately, the challenge is to design sophisticated intrusion detection methods that identify subtle anomalies in protocols that grow increasingly complex.

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

Todo list