CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Security Analysis of Attack Surfaces on the Grant Negotiation and Authorization Protocol

Master's thesis in Computer science and engineering

Åke Axeland
Omar Oueidat

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Security Analysis of Attack Surfaces on the Grant Negotiation and Authorization Protocol

Åke Axeland
Omar Oueidat

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Security Analysis of Attack Surfaces on the Grant Negotiation and Authorization Protocol
Åke Axeland
Omar Oueidat

Supervisor: Pablo Picazo-Sanchez, Department of Computer Science and Engineering
Advisor: Tobias Ahnoff & Martin Ahlstedt & Kasper Karlsson, Omegapoint
Examiner: Koen Claessen, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Security Analysis of Attack Surfaces on the Grant Negotiation and Authorization Protocol
Åke Axeland
Omar Oueidat
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Accessibility is a booming practice, with applications incorporating easy authentication and authorization increasing. OAuth 2.0 is a framework created to easily integrate resourceful platforms with a client application, giving users the opportunity to access their resources in different means while only storing them in one place. Due to resources often being confidential or private the security of such frameworks is imperative. GNAP is a new protocol inspired by OAuth 2.0, created with the intention to uphold security standards of modern application usage. This thesis tests GNAP and its robustness against legacy attacks targeting OAuth 2.0. The tests consist of vulnerable redirect URI attacks, access code hijacking, CSRF, and AS mix-up attacks. Results show that due to GNAP's cryptographic-based design, attacks that utilize data manipulation or additional input are not possible in the environment created for the thesis. However, given the less secured client instance in the protocol, AS mix-up attacks are possible in a niche environment given the assumptions made in the thesis.

Keywords: OAuth 2.0, OAuth 2.1, GNAP, authentication, authorization, security

Acknowledgments

We would like to express our gratitude to our supervisor Pablo Picazo-Sanches for all the guidance and help in writing this thesis, the help was invaluable and hugely impacted the way this thesis turned out. We would also want to thank our technical supervisors at Omegapoint, Tobias Ahnoff, Martin Ahlstedt, and Kasper Karlsson, who always were there to answer our questions about the protocols as well as having meaningful discussions with us about the thesis. We also want to thank our examiner Koen Claesson for being cooperative and helpful in pushing us in the right direction to a nuanced and innovative project. Finally, we would like to direct a thank you to our opponents Gustav Pettersson and Lina Lagerquist Sergel for the feedback on our report, due to this we were able to refine the thesis to be better understood by our audience.

Åke Axeland & Omar Oueidat, Gothenburg, June 2021

Contents

List of Figures

List of abbreviations

- OAuth: Open Authentication
- API: Application Programming Interface
- IETF: Internet Engingeering Task Force
- GNAP: Grant Negotiation and Authorization Protocol
- CSRF: Cross-Site Request Forgery
- HTTP: Hyper Text Transfer Protocol
- RO: Resource Owner
- RS: Resource Server
- AS: Authorization Server
- BCP: Best Current Practice
- RFC: Request For Comments
- URI: Uniform Resource Identifier
- HTML: Hyper Text Markup Language
- CSS: Cascading Style Sheets
- TLS: Transport Layer Security
- JSON: JavaScript Object Notation
- JWT: JSON Web Token
- JWE: JSON Web Encryption
- JWS: JSON Web Signatures
- JAR: JWT Secured Authorization Request
- PAR: Pushed Authorization Request
- RAR: Rich Authorization Request
- PKCE: Proof Key for Code Exchange
- IS4: IdentityServer4
- HAS: Honest Authorization Server
- AAS: Attacking Authorization Server
- Pk: Public Key
- FAPI: Financial API

1
Introduction

"Login with Google" or "Login with Facebook" is a feature recognizable in modern applications requiring authentication. This feature conveniently allows the user to skip registering to service, and developers can choose to skip creating their own login service. OAuth 2.0 is a framework that creates the opportunity to have these authorization methods and is widely adopted in modern applications [1]. OAuth 2.0 is an authorization framework that was published in 2012 with the goal of a secure delegation process that respects the principle of least privilege [2, 3]. It is currently the most adopted authorization framework, becoming more of a standard for authorization [1].

Utilizing OAuth 2.0 allows for generating partial access to data. Having partial access to data is more appropriate when, for example, granting access to a specific drawer in a file cabinet rather than granting access to the cabinet. This is similar to how people typically use OAuth 2.0 to grant websites access to their Facebook or Gmail accounts to use pictures or contacts while keeping other data out of reach [4]. OAuth 2.0 does not give the application your login credentials, i.e. the full access, but rather makes sure the application is authorized to access specific data from the API, i.e. the drawer [5].

While keeping a social media account protected from foreign applications may be smart in the case of privacy, OAuth 2.0 is also used in applications to access for example financial APIs where security is more important due to the ramifications of a breach [6, 7]. The security standard required is something that the framework can achieve, but it is imperative that the implementation is done correctly [8, 9].
There have been discussions of how to remedy the mandatory practice requirements for OAuth [10]. As of October 2020, the Internet Engineering Task Force (IETF) produced a draft for a new protocol called the Grant Negotiation and Authorization Protocol (GNAP). The intention with GNAP is to fill the shortcomings OAuth 2.0 currently has, as well as to separate the protocol from the developer so that their responsibility for a secure application decreases.

OAuth 2.0 has become an industry standard and best practice for current systems. This thesis is a collaboration with Omegapoint. They perform security reviews, which include but are not limited to implementations of systems on top of OAuth 2.0, and give advice to customers on best practices. Any changes and additions to the standard body of OAuth 2.0 will affect the tech industry in general and Omegapoint in particular. Omegapoint wants to maintain a cutting edge in state-of-the-art

cybersecurity and to monitor the progress of GNAP to see how it might affect the security of systems built on OAuth 2.0 today.

Authorization vs Authentication There is a very clear distinction between what the original idea of OAuth 2.0 was and what it can be interpreted as today. *Authorization* is when an entity wants partial access to data, it can provide proof that it has the right to access that data, thus gaining authorization. Compared to *Authentication* which essentially means providing enough information to an entity to verify that you are the actual user, e.g. when you log in to a service.

1.1 Problem Statement

This thesis will review new and upcoming protocol, GNAP, and more precisely its robustness against legacy attacks. It will also highlight how GNAP handles the vulnerabilities from OAuth 2.0 differently than OAuth 2.0's successor, OAuth 2.1.

Limitations In the following, we explain the main limitations this thesis has.

- **Countermeasures:** We do not aim to come up with new attacks, neither propose fixes to the protocol for any vulnerabilities that are potentially discovered. The thesis primarily focuses on the findings from the attack coverage of GNAP and how the protocol covers the attack or what the possible consequences are. It is infeasible to propose solutions since they will require testing for completeness.
- **Focus on protocol design:** The focus of the attack surfaces will be those that target the protocol design. For example, in OAuth 2.0 there exist optional parameters which when not included, or in some cases included, makes the framework vulnerable to various threats. This is clearly a design flaw, while faulty implementation or malpractice is not and will therefore not be covered by the thesis.
- **Fifth draft:** The latest draft published as of writing the thesis is version five, thus the research reflects the findings against this iteration of the protocol.

Ethical Considerations Any vulnerabilities found during our research will be reported to the IETF.

1.2 Related Work

Fett et al. [11] did a comprehensive analysis of OAuth 2.0, where they followed the recommendations given by OAuth 2.0, as well as the best current practices to create state-of-the-art applications for their tests. Their paper also described the discovery of two attacks previously unknown to the OAuth 2.0 team, one of which was tried

on GNAP in this thesis. Our thesis uses their structured description of the attacks, applying assumptions to the protocol in order to produce the attacks.

Yang and Manohoran [12] studied the security vulnerabilities of OAuth 2.0 with the use of an attacker model. In their paper, they describe potential leaks of authorization codes during transmissions. Through replay attacks, impersonation attacks, and CSRF attacks they proved these attacks may lead to compromising the protected resource from OAuth 2.0. Our thesis follows the same method of creating basic applications for quick testing given the specification we define in the methodology.

Additional work has been conducted by Li et al. [13], which discusses the user security when using OAuth as an authentication method. A scanner called OAuthGuard was used to detect vulnerabilities when using OAuth to sign in with their Google accounts. The results showed 67 sites out of 137 that have at least one security vulnerability.

Shehab et al. [14] discuss the implications of security issues in mobile applications that utilize OAuth which is also interesting given the increasing use with the ever-growing mobile usage.

2
Background

Before the creation of OAuth, the industry used a basic authentication pattern which was made famous by HTTP Basic Authentication in order to authorize a client's access to a protected resource, which tightly knitted authorization and authentication of applications [6]. In this pattern, the user was asked to present her credentials which were then submitted into an HTTP form that the client could use to log into a service holding the protected resource. The client stored the user's credentials so the cumbersome task of presenting credentials every time can be skipped [6]. However, this authorization pattern exposes the user to numerous vulnerabilities, the most severe threat is the difficulty revoking access to resources since the credentials are stored in the client [15]. This was especially the case for third-party websites which is a very common use case for OAuth today [16].

An equally bad vulnerability is the zero-sum game in giving a client the user credentials [17]. By providing clients with user credentials the clients then have the same privileges as the user. Therefore the only way to restrict the privilege is to reduce the privilege of the user, which reduces functionality for the user application. Companies like Facebook noticed this and started to implement their own authorization schema [6]. The newly created authorization schema was plentiful and abundant, which put a strain on application developers trying to implement authorization services. This problem screamed for standardization [6].

The focus of the thesis is on the successors of OAuth 1. There are some basic key features and terminology that OAuth 1 introduced, which the successors use as well.

- **Resource Owner (RO):** The entity which will grant or deny access to the resources it retains.
- **Resource Server (RS):** This is the service that holds access to the RO's protected data. E.g. Facebook, which has people's pictures and contacts.
- **Protected Resource:** The resource owner's personal data, pictures, and contacts from Facebook are protected resources.
- **Client:** It is important to differentiate the user from the client. In OAuth, the *client* is the *client application* who wants access to the protected resource. For example, LinkedIn might want access to the user's Facebook contacts in order to present them as potential contacts on their platform.
- **Authorization Server (AS):** The server with which the client communicates in order to eventually be presented with an *access token* that is used for accessing the protected resource.
- **Scopes:** Scopes are what permissions the client wants authorization for from

the resource server. For example, if the resource server is the user's Twitter account a scope can be "read tweets".
- **Flows:** The flows are schemas of how the data are transmitted between the user, client, AS, and RS.

Over the years of usage, several areas of OAuth 1 presented themselves as difficult to work with since it either limited the API excessively or was too challenging to implement. Consequently, companies called for a newer, more flexible framework which over the years formed itself into the currently popular OAuth 2.0 [18].

As previously stated, OAuth 2.0 is defined by its different flows in which data are transmitted. A simplified figure of how an abstract OAuth 2.0 flow works is presented in Figure 2.1. The figure represents an RO using a client application to gain information from another application (1), the RS, by allowing it to get hold of a token from the AS (2) and (3). This token can then be passed and verified by the RS which authorizes access for the client (4) and (5). Hence, the client can access resources from other applications without possessing user credentials.

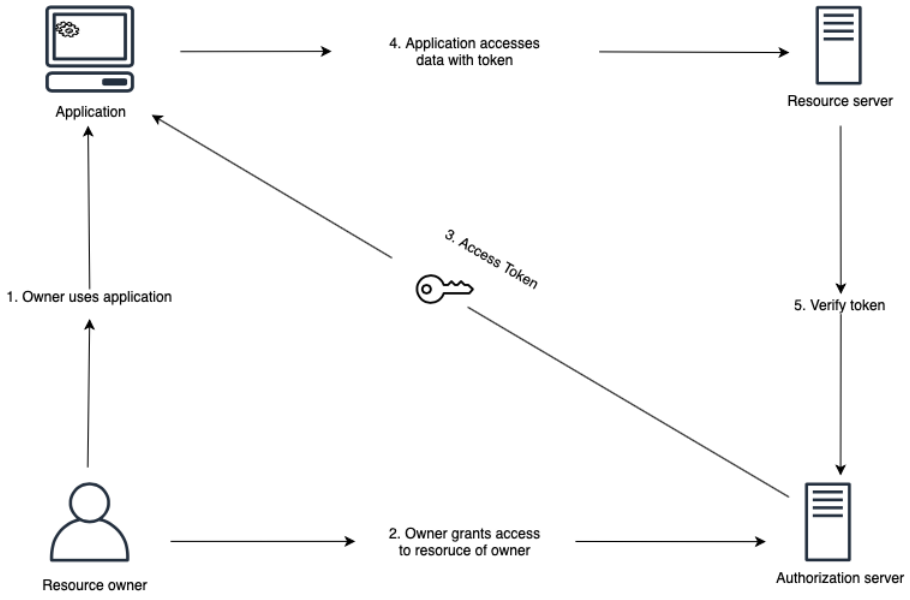


Figure 2.1: A high level diagram that shows the flow of information in OAuth 2.0

OAuth 2.0 is more of a general framework, i.e. it is made for authorization but is applied differently depending on the use cases. This makes it lack the details necessary to enforce higher security demands [9, 19]. Therefore, the framework has been updated with extensions and Best Current Practice (BCP) documentations to uphold security standards for different niche situations, e.g. if the client is a single page application or if the client is the owner of the resource [9, 19]. Since the dependency on varying extensions grew plentiful, it became difficult to keep track of when to implement them and which were relevant for the application's purpose [9]. To solve this, a new framework built upon OAuth 2.0 was made named OAuth 2.1 [20].

OAuth 2.1 is an attempt to clean the OAuth 2.0 framework and make it more consistent to implement [20]. This iteration is defined with a more narrow definition of the aforementioned framework (e.g. the different flows in which data are transmitted has been narrowed down from four to two) [21]. In a short summary, this version of the framework excludes less secure flows, formalizes clearer guidelines, and enforces BCPs [20].

OAuth 2.1 incorporates some of the extensions used to make OAuth 2.0 more secure. However, the issue with the extensions is that the interaction among them is complicated, and making them work with legacy applications is difficult [9]. Additionally, there are plenty of BCP implementations regarding which service OAuth is to handle [9]. Formal security analysis on the protocol has shown that when the best practices are followed during implementation, the protocol is deemed secure [11]. Moreover, the practices have grown extensively, making it more difficult to follow them, and they vary for the intended purposes the framework is used in [9].

Regardless of how secure a protocol is, if it is not possible for developers to enforce the security principles required when implementing the protocol, vulnerabilities might exist [22]. Therefore, instead of developing the old framework further, as in the case of OAuth 2.1, there is an incentive for a brand new framework that is more manageable and consistent, inspired on OAuth 2.1. The IETF recently started building the Grant Negotiation and Authorization Protocol, a new protocol intended to replace OAuth 2.0. This project is in its infancy where the first draft was posted as late as October 2020 and is continuously updated [23].

Summary
Authorization between parties with the use of OAuth 2.0 is a feature utilized widely nowadays [24]. Implementing OAuth 2.0 should be easy and leave little room for errors. The current standards inadvertently made it difficult to implement and consequently a new protocol is being developed. However, the new protocol is currently missing external security reviews or penetration testing which is of utmost importance for a protocol that aims to make authorization more secure.

3
Authorization Frameworks

This chapter presents the necessary technical background about OAuth 2.0, OAuth 2.1, and GNAP. The chapter uses the Request for Comments (RFC) publications for respective protocols. These publications serve as the principal technical development and standards-setting bodies for the protocols.

3.1 OAuth 2.0

OAuth 2.0 is defined by its different flows for different use cases. All flows have the same high-level idea but are implemented in different ways depending on the characteristics of the application. The most commonly used flow is called Authorization Code Flow and is depicted in Figure 3.1. Authorization Code Flow works in the following way:

1. OAuth 2.0 is initiated when a user instructs the client to fetch some protected resource from the resource server.
2. The client responds to the user with (i) client id used for verification in the authorization server, (ii) redirection URI (i.e. the route where the data from the resource server are to be sent), and (iii) a state parameter which can be used to store information, often the state of the protocol.
3. The user then sends the data received from the client to the authentication endpoint in the AS, where client id and redirection URI are verified.
4. If the client id and redirection URI are verified correctly the authorization server asks for the user's credentials.
5. The user then passes her credentials to the authorization server.
6. Then the credentials are verified and if accepted, the authorization server generates a nonce called *access code* and sends it back to the user along with the state.
7. The information from the authorization server is then redirected to the redirection URI of the client.
8. Now the access code can be used by the client in an attempt to get hold of an access token. Along with the code, the client id and redirection URI are sent.
9. If all the parameters are correctly verified by the authorization server, the client is authorized access to the protected resource by being handed an *access token*.
10. The client now has hold of the key to the resource and can query the RS for it.

11. At stage 11 in Figure 3.1, the RS passes the resource to the client if the access token was valid.

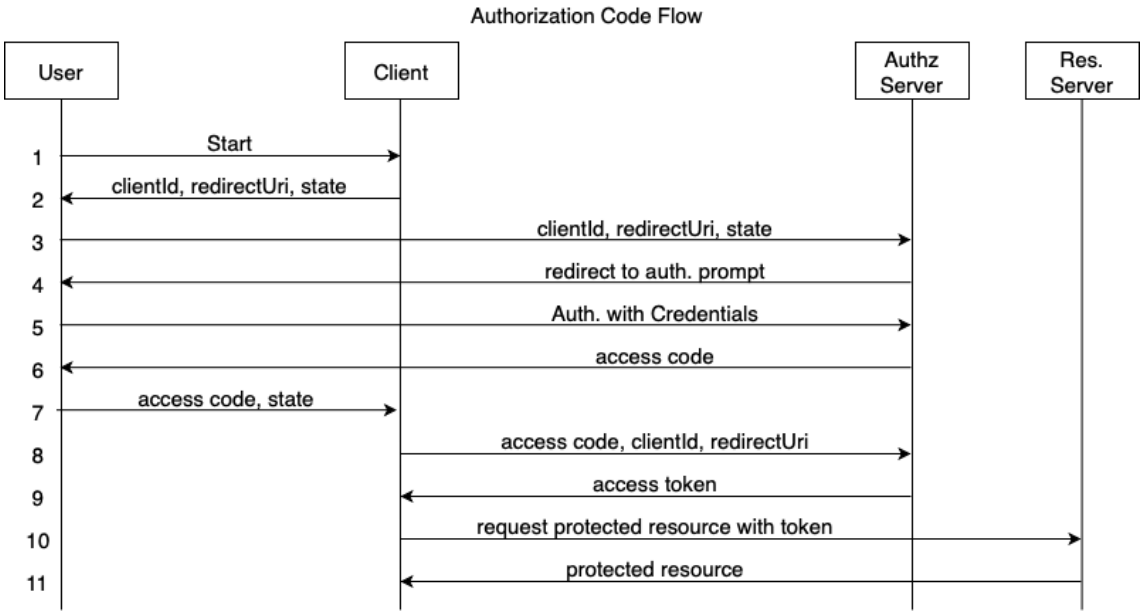


Figure 3.1: Authorization Code Flow for OAuth2.

Even though Authorization Code Flow is the most commonly used flow it is not always preferred. The reason for this that the flow requires a secure channel where the access code can safely be transmitted over. Such a secure channel does not always exist, e.g. in single-page applications where all the JavaScript, HTML, and CSS code can be inspected in the browser. Instead, a different flow called Implicit Grant flow is used. However, this is a less secure flow since the application is incapable of hiding the access code, there is no use for it so the client gets an authorization token earlier in the flow. There are additional flows described in the RFC for OAuth 2.0. Use cases for the different flows are described below:

- **Authorization Code Flow:** If the application is a regular application executing on a server, i.e. code can be kept secret from adversaries, then this is the flow to choose.
- **Implicit Flow:** Used when the application is a single page application executing in the browser without the possibility of hiding information.
- **Client Credentials Flow:** Used when the client is the resource owner, e.g. computer operating systems.
- **Resource Owner Password Flow:** Only used when the client is completely trusted since the client requires the possession of the user's login credentials in order for the flow to work.

The access token can take different forms, with the most common format being the Bearer Token [25]. This is sent in the HTTP header to the RS in order to verify the authenticity and is required to be sent with the use of TLS. The token is a string that is opaque to the client.

A missing aspect in Figure 3.1 is the use of scopes. Scopes allow the authorization

server to restrict what type of access the client is permitted and is embedded in the access token itself. An example of this is a client accessing a Twitter account where scope may allow the client to only read tweets from the feed but is not authorized to post tweets.

OAuth 2.0 also allows the developers the option to include *refresh tokens*. A refresh token is simply another type of token that is used to get a hold of another access token if it is about to expire and the client desires more use of the resources. A refresh token is used to extend access, rather than going through the whole flow of OAuth 2.0 again.

3.1.1 Extensions

As previously stated, throughout the years OAuth 2.0 has been used, various vulnerabilities have sprung up which has required extensions to be added to the framework in order to mitigate attacks.

Token Revocation
Token Revocation is a way for the user to clean up all the tokens granted to a client so that no refresh tokens or access tokens linger after they have been used. This is done by making them one-time use only, rendering them useless after they are consumed, or tightly bound to the client by some cryptographic key [26].

JAR: JWT Secured Authorization Request
JAR is used to encode a single authorization request into a JSON Web Token (JWT). Initially, the request is signable using *JSON web signatures*, providing integrity to the message. Secondly, by encrypting the request using JSON web encryption (JWE), the confidentiality of data is upheld. Lastly, since the request was signed, the source is authenticated bringing with it non-repudiation [27].

PAR: Pushed Authorization Request
This extension enables a client to pre-register complex authorization requests with narrower scopes in the AS where the AS then provides the client with a *request URI*. The URI can then be used to reference the complex request call and provide the client with more narrowly defined scopes [27].

RAR: Rich Authorization Request
Just like PAR, this is an extension for fine-grained authorization. The extension provides guidelines to allow clients to send a JSON object which more narrowly defines the scope that the client wants authorization to [28].

3.2 OAuth 2.1

As previously stated, with the popularity growing for OAuth 2.0, so did its different use-cases. This lead to an increased amount of extensions added on top of the

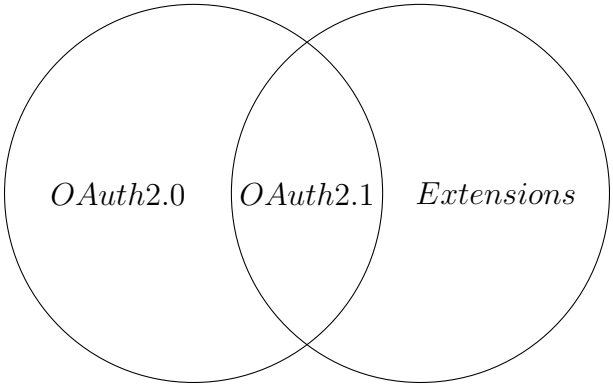


Figure 3.2: Venn diagram showing the relationship between OAuth 2.0 and OAuth 2.1

framework. While the extensions increased security, the cooperation between the extensions of the already complicated protocol became a heavy burden on developers. As far as is known, there are no security audits that have been done on all the different combinations of extensions, leaving the security of the implemented system questionable. A solution to this was a new version, OAuth 2.1. It is not considered an expansion of OAuth 2.0, instead, it exists as a stricter way to implement the framework in order to fill security holes that existed in OAuth 2.0 [29]. In order to fill these holes, some mandatory extensions have to be included in the system in order to be called an OAuth 2.1 implementation. The relationship between OAuth 2.0 and 2.1 is shown in Figure 3.2.

While OAuth 2.1 is built on OAuth 2.0 and its development throughout the years, there are differences between them which are explained below.

PKCE: Proof Key for Code Exchange
PKCE, pronounced *pixie*, is one of the major extensions enforced from the OAuth 2.0 framework [20]. Its function is to mitigate access code interception attacks [30]. PKCE utilizes a code challenge and code verifier schema as follows; the client generates a challenge by, most commonly, hashing a verifier.

$$Challenge = Sha256(Verifier) \tag{3.1}$$

This challenge is then transmitted to the AS to store for the started session. This way, only the client, which knows the verifier, can prove that it is the origin of the request since guessing the verifier is infeasible. Subsequently, when the AS receives the access code from the client, the verifier is also sent, so the AS can be sure that the right client is asking for authorization. This is depicted in Figure 3.3. For each token requesting cycle, the code challenge and code verifier has to be refreshed. If they are not refreshed, an attacker that monitors the HTTP requests/responses can intercept and reuse them [31].

Deprecated Insecure Flows
Two flows have been deprecated; the implicit grant flow and resource owner password

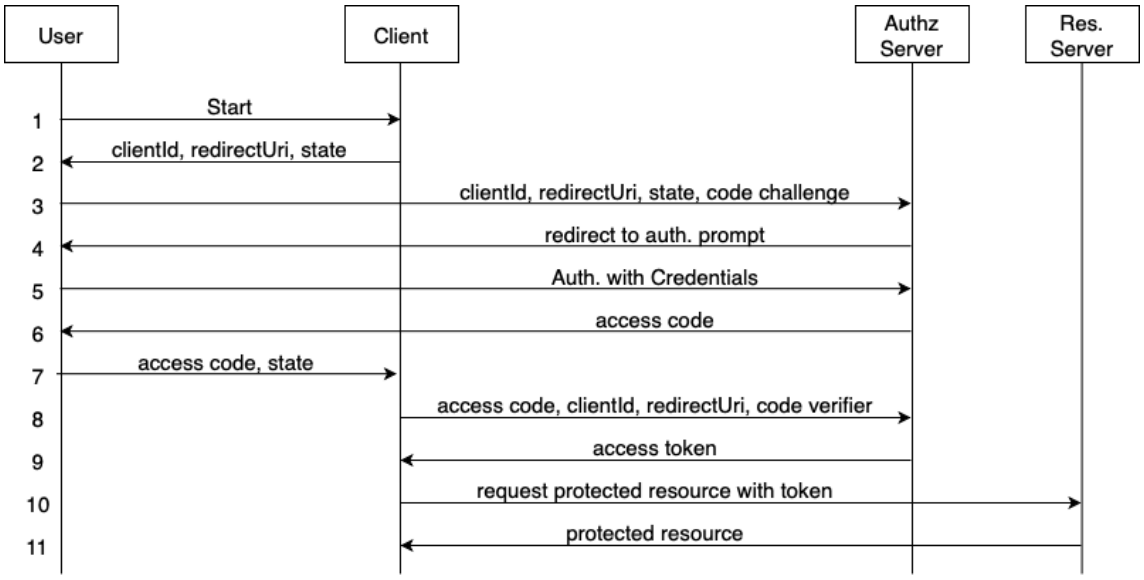


Figure 3.3: Authorzation code flow with the proof key for code exchange extension.

credential flow [20, 30]. The implicit flow has been removed due to the mandatory use of the PKCE extension since this requires the adversary to solve the challenge as well as intercept the access code. As stated earlier in this section, guessing the verifier is infeasible, making it possible to safely exchange access codes, hence rendering the implicit flow useless. The resource owner password credential flow has been deprecated partly due to its limitations in controlling privileges [32].

Exact String Matching

The redirect URIs are now required to be compared with exact string matching. This means whitelisting redirect URIs that consist of general values is not allowed. It might be lucrative as a developer to whitelist a domain such as https://test.com/* in order to be able to test the system with multiple redirect branches. However, this exposes the system to redirect attacks [20, 30].

No Bearer Tokens in URI Query

It is no longer allowed to transmit the access token in the query of the URI as stated by the RFC for OAuth 2.0. Instead, other means to transmit the token are required. For example, using a POST form and including it in the body, preferably encrypted and over TLS in order to reduce the exposure of the data in the browser, thus making it harder to acquire information about the access token [20, 30].

Restrictions to Refresh Tokens

Refresh Token is used by the client to get hold of a new authorization token when it expires in order to bypass the need to do the whole flow again. In OAuth 2.1, refresh token extension has been restricted so it either has to be a one-time-use token or

cryptographically bound to the client with the consent of the RO [30, 33].[1]

3.3 Grant Negotiation and Authorization Protocol

As stated before, GNAP is not built upon OAuth but is instead a complete rewrite of the framework [29]. GNAP removes some of the ambiguity of OAuth and introduces various new security enhancements and new features. One of the new features in GNAP is the separation of the resource owner and the End-User [34]. The separation is important since they might be, but are not exclusively the same entity. This is something OAuth 2.0 is not designed for, but rather is left up to the developer to solve.

The main features the new protocol introduces are the following:

- Different authentication paths for a client to use which does not utilize the redirect URI thus mitigating attacks targeting it [34].
- Request Continuation allows clients to negotiate authentication details during the authentication phase as well as additional privileges and additional access tokens [34].
- Multiple Access Tokens, making it possible for a client to access several different resources at once [34].
- All tokens also come with a secret so that even if a token is intercepted the adversary has to know the secret as well, adding an additional layer of security [34].

In GNAP the focus shifts from parties in the flows to a heavier focus on *roles*. A party is able to hold multiple roles, and as long as the requirements of the roles are fulfilled the protocol will work as intended [35]. The different roles are the RS, AS, Client Instance, RO, and the new end-user. The reason for the end-user being separated from the RO is, as previously stated, that it might not always be the case where the actual owner of the resource is requesting the resource and with the existence of these roles, the protocol will cover cases like that as well [34].

GNAP is also not limited by the browser, since it no longer relies on HTTP query parameters to transmit data. Instead, the protocol sends messages with data embedded in JSON. This allows the protocol to be deployed in a wider amount of applications since applications do not need a browser [34]. Another benefit of using JSON as default, extensions like RAR are built into the protocol, and clients are able to request fine-grained scopes to the AS [35].

The protocol also encompasses the specifications and remedies from OAuth 2.1, where it focuses on token management in order to secure the access tokens. It utilizes the idea of requiring the token to be sent through a secure channel when

[1] There are other changes to the protocol as well. These can be found in the RFC of OAuth 2.1, `https://tools.ietf.org/html/draft-ietf-oauth-v2-1-01`.

authorizing an end-user. The AS also must ensure that the token it has created and given to the client is bound with a key that the client can provide to verify its previous authorization. GNAP does not require that the token is a bearer token, but it is recommended so that the token can be sent with a POST request with the token embedded into the header.

A strong point of GNAP is that it allows the clients to negotiate redirects or other important details during processes. This will allow a client instance to negotiate for supplementary privileges or accesses. The benefit is that it is up to the AS to decide while being queried for authorization if user interaction is needed or not. This means that for less secure operations the AS can decide to skip the user interaction and just use the client's credentials and for higher security operations the user can be involved in multiple ways.

Additional means of authorization have been added; asynchronous authorization. In this configuration, the client starts the flow with the AS but the RO interacts with the AS in other means than the client. The client can then poll the AS to check if the interaction is finished and then proceed with requests. This enables the use of external devices as means of authorization.

GNAP introduces multiple access tokens, allowing for requests to access multiple resources. This is possible in OAuth 2.0 too, but it requires additional grant requests. The idea is to encourage developers to create systems with more granular permissions. An example is that you could request read privileges for one resource and write privileges for another resource. With GNAP, the user only has to approve these requests once.

Finally, GNAP incorporates and relies on signing all requests from the client to the AS. This is done to prevent adversaries from altering messages in transit and prevents malicious clients from pretending to be honest clients, i.e. adversary clients that intercept the flow and pose as real clients. In the first interaction between a client and an AS, the client presents a public key as well as a method of proving possession of this public key. The client then has to present proof of this key so the AS is sure it is communicating with the client and not an impostor. There are several methods of proof allowed by the RFC at this time of writing but this thesis will focus mostly on the detached JSON Web signatures (JWS).

4

Methods

This chapter covers the methodology used to create the threat model required for the analysis of GNAP. The basis of the project and understanding the attacks is achieved by trying out a few attacks made available by a cybersecurity website called PortSwigger. This chapter also introduces a few legacy attacks explained in-depth, which gives the basic knowledge to follow the thesis.

4.1 OAuth 2.0 Implementation Verification

A basic system was constructed since the intention is to present already existing attacks in order to give more understanding of them. It contains a web application client, a web-API server that serves as both a resource server and an authorization server. Instead of the resource server and authorization server being separate entities they now communicate over separate endpoints in the server and have access to the same database. Only the authorization code flow was implemented due to numerous reasons; the flow is the most common one, it stays rather untouched in OAuth 2.1, there exists a similar flow in GNAP and many attacks focus on this particular flow seen in Figure 4.1.

GNAP's RFC has categorized parameters in different classes of importance; optional, recommended, and required. All the required parameters are included in the flow, without them it is not an OAuth 2.0 implementation. However, only some of the recommended and optional parameters are included. The non-required parameters have different effects on the system's exposure to vulnerabilities. Including some while excluding other non-required parameters will expose the system to a larger attack surface to build legacy attacks upon.

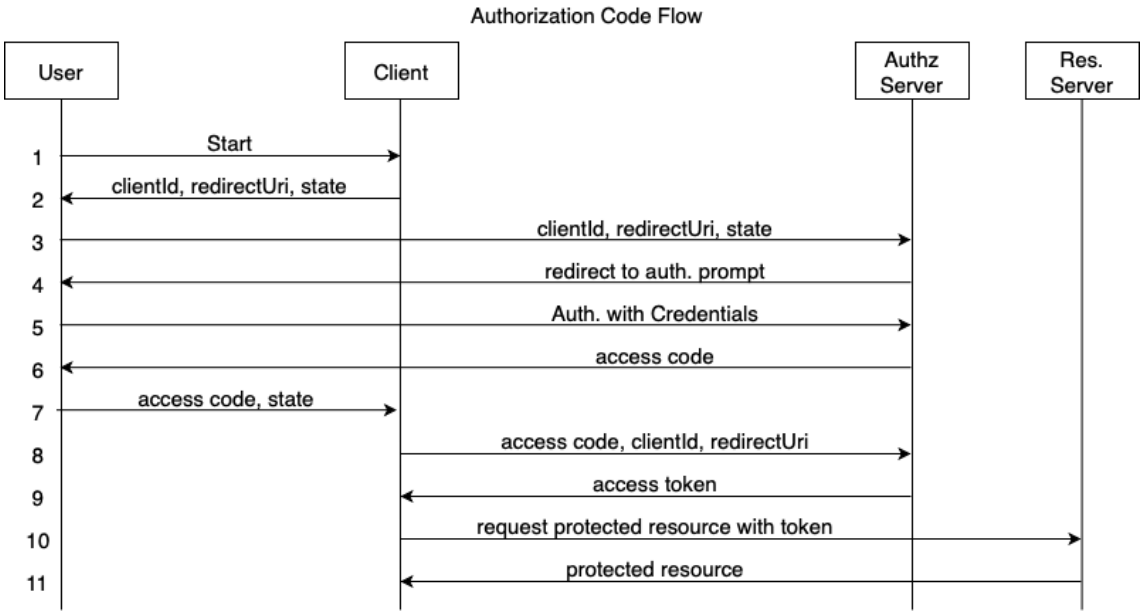


Figure 4.1: Authorization code flow.

4.1.1 Initial Request

As depicted in Figure 4.1 the user starts the flow by notifying the clients that she wants to grant the client authorization to the protected resource (1). The client recognizes this engagement and returns the client's specifications to the user (2). This information is presented in the following itemization.

- **Response Type** - A string that has to be exactly matched to the string "code". REQUIRED.
- **Client Id** - An identifier for the client which is predetermined. REQUIRED.
- **Requested Scopes** - The scopes the client wants to have for the protected resources requested. OPTIONAL.
- **Redirect URI** - The URI the AS will redirect the user to after it gets the access code. OPTIONAL.

When receiving the information from the client, the user is redirected to pass this information to the AS (3). Depicted in Figure 4.2, the user passes both the required parameters as well as some optional in the URI as described by the RFC for OAuth 2.0.

```
GET/home/auth?responseType=code&clientId=ClientApplication&
redirectUri=https://localhost:5001/home/oauth-callback&
requestedScopes=product.read HTTP/1.1
Host: localhost:5003
```

Figure 4.2: The GET request that the Client redirects the user to during the third step in Figure 4.1.

4.1.2 User Credentials and Access Code Retrieval

Information provided by a user is validated by the AS, and if verified, will redirect the user to its authentication endpoint (4). In the authentication endpoint, the user passes her credentials to the AS for verification (5). If the credentials are validated, the AS redirects the user back to the client's redirect URI with the newly created access code (6,7). The HTTP message sent to the client from the user through redirection is depicted in Figure 4.3.
The AS checks that the information provided by the user is valid data and after verification the AS redirects the users to its authentication endpoint (4).

```
GET/home/oauth-callback?accessCode=38aff5b6ff5c5cfa8d352b130ece46fd
HTTP/1.1
Host: localhost:5001
```

Figure 4.3: Redirection to the Clients callback endpoint with the access code retrieved from the AS.

4.1.3 Token Request

When the client is in possession of the access code, step (8) can commence where the client transmits the access code along with the required fields described below.

- **grant_type**, set to "authorization code"
- **code**, the access code given to the client by the AS through the redirect of the user.
- **redirect_URI**, if this was present in the original request it is required to be included in this request as well. The value has to be compared to the original redirect URI with exact string matching.
- **client_id**, has to be present if the client is not authenticated with the AS, which this client is not since it is already stated that the client is public.

This is done through an HTTP POST method where the structure of the message is depicted below in Figure 4.4

```
POST/home/requestToken?code=38aff5b6ff5c5cfa8d352b130ece46fd&
clientId=client%20application&redirectUri=https://localhost:
5001/home/oauth-callback&grantType=authorization_code HTTP/1.1
Host: localhost:5003
```

Figure 4.4: POST request to the AS from the client in order to receive an access token.

4.1.4 Token Response

Once all parameters from the request have been verified by the AS, it will send a response with the access token required for authorization at the resource server, or in this case, the resource endpoint.

Unlike the other messages sent in the flow, the RFC specifies that the token should be transmitted in JSON format through the body of the HTTP response. The system complies with this and the structure of the message is in the form depicted in Figure 4.5.

```
1    "access_token" : "1c3b8eb6f614211ba5d02b9db2b5866f",
2    "grant_type" : "authorization_code"
```

Figure 4.5: Response from the AS to the client with the access token required to gain authorization at the resource server

The client now possesses the token needed to access the protected resource, which completes the authorization code flow. The token has an expiration time of 10 minutes, after that the entire flow has to be re-initiated from start. There are no refresh tokens implemented in this system.

4.2 IdentityServer4

Omegapoint supplied us with a working OAuth 2.1 system that utilizes a framework called IdentityServer4 (IS4). IdentityServer4 is an out-of-the-box authorization server with all the specifications of OAuth 2.1 implemented, as shown by their list of supported specifications [36].
As mentioned in Section 3.2, OAuth 2.1 tries to collect the many BCPs that have been established during OAuth 2.0's lifetime. The following points are the requirements for OAuth 2.1 and a brief rundown of how IdentityServer4 follows these:

- PKCE: IS4 follows the RFC 7636 with incorporating PKCE in its flow. PKCE is not required in the framework and can be toggled on and off. However, since we wanted an OAuth 2.1 server we naturally toggled it on.
- Deprecated Insecure Flows: IS4 supports all four standard flows given in OAuth 2.0, but the sole testing will be on the authorization code flow, leaving out the implicit grant flow and password credential flow. This was simply done by creating the client with these specifications.
- Exact String Matching: This is not something IS4 explicitly states on their specification page. However, since the project is open source we can see in their GitHub repository that they do in fact use exact string matching.[1]
- No Bearer Tokens in URI Query: IS4 supports the 6750 RFC which states how to handle access tokens by using TLS to transmit Bearer tokens. Moreover, the RFC issues more recommendations on how to handle the Bearer Tokens to ensure security.
- Restrictions to Refresh Tokens: An important role of OAuth 2.1 is to ensure that compromised tokens should be handled gently. IS4 follows RFC 7009 and

[1]The URL for redirect URI validation `https://github.com/IdentityServer/IdentityServer4/blob/main/src/IdentityServer4/src/Validation/Default/StrictRedirectUriValidator.cs`

7662, these state that refresh tokens should be single-use only, rendering them useless once they have been used, as well as them being short-lived. A refresh token that has been used, can not be used to generate an additional access/refresh token pair.

4.3 GNAP Implementation Verification

The test bed for attacks against GNAP was performed against an existing implementation called xyz-auth developed by Justin Richer, one of the developers and contributors to GNAP [10]. The following sections describe a complete description of the GNAP implementation in order to verify that the required parts in the protocol are actually present in the flow. The information is directly correlated to the RFC of GNAP in order to verify that the required parameters are included as well as the optional fields behave as intended. The thesis focuses on the configuration that is closest in use case and implementation to the authorization code flow in OAuth 2.0 and OAuth 2.1.

The different ports the endpoints are hosted on are depicted in Figure 4.6. These ports are not required, but they are the ones configured in the xyz project.

Figure 4.6: Presentation of what ports the entities in the system are running on.

4.3.1 Authorization Code Transaction Request

Using the out-of-the-box implementation of GNAP provided, the user is prompted to start a new code transaction. According to GNAP's RFC specification, the different fields of a transaction start has some requirements, these are stated below:

Access token field, RFC 2.1.1: The parameters describe the desired properties of the access token.

- **Access:** A required parameter that describes the rights that the client is requesting from the AS.
- **Label:** A value that is chosen by the client to refer to the access token. This value must be unique and opaque to the AS. This parameter is required for requests of **multiple** access tokens and optional for **single** access tokens. If this value is set in the request the AS must reflect this opaque value as it is in the response. Since this implementation only handles single requests this is

not included.

Client identification fields, RFC 2.3: The client's properties, where focus lays in public key transmission used for verification.

- **Key:** The only required field in the client part of the request. The key should contain either the public key of the client or a reference to it. This key is then used for signing and verification of the data transmitted further in the protocol.
- **Display:** In this parameter, additional information that the client wants the AS to present towards the user can be added. This is an optional field.

Interaction, RFC 2.5: Typically utilized when an AS requires interaction with a user, which is the case for authorization code flow.

- **Start:** The required part that contains information about how the client wants to interact with the user. Can be an application, redirect endpoint, or a user-given code.
- **Finish:** An optional part that tells the AS how the client wants to know that the interaction has finished. Can be either a redirect URI or a PUSH which means that the client is ready to receive a POST request from the AS when the interaction has terminated.
- **Hints:** Other means for the client to send the information it wants the AS to present to the RO. Could for example be already filled in username etc.

By running the out-of-the-box implementation and starting the code transaction, the system returns the request in the command prompt. The structure of this request is presented in Figure 4.7. As seen in the figure, all the required parts of the request are present, as well as some optional parts the implementation has chosen to include which also has been described above.

```
"interact":{
    "finish":{
        "uri": "http://localhost:9839/api/client/callback/
           FUfe7TrJMpwj1Tz",
        "nonce": "sfmtlzrcvUEOX9UsmhdJ",
        "method":"redirect",
        "hash_method":"sha3"
    },
    "start": ["redirect"]
},
"client":{
    "key":{
        "proof":"jwsd",
        "jwk":{
            "kty" : "RSA",
            "e" : "AQAB",
            "kid":"xyz-client",
            "alg":"RS256",
            "n": "jawjdoaijsdoasqdleAXSDloKDWPOk00apKDmdikjaP..."
        }
    },
    "display":{
        "name":"XYZ Redirect Client",
        "uri": "http://localhost:9839/"
    }
},
"access token": {
    "access":["openid","profile","email","phone"]
}
```

Figure 4.7: Initial request for an access code which will authorize the client access to a protected resource.

4.3.2 Response to Initial Request

When the AS has validated the request there are a number of potential responses from the AS. There are no particular fields that are required. Instead, all of them are optional depending on the properties the system tries to fulfill. This implementation responds with the **continue**, **interact** and **instance id** fields as can be seen in Figure 4.8.

```
"instance_id":"604b24a584d08d3df575132b",
"interact":{
   "redirect": "http://localhost:9834/api/as/interact/eije4TOadr",
   "finish":"s38m4Aq5GHTRInQZuEDz",
},
"continue" : {
   "uri":"http://localhost:9834/api/as/transaction/continue",
   "access_token":{
       "value":"TT3pbcqBba1Yw86hFVz0v3pEbQZ9H66U07X9vx9Fqrbr...",
       "bound" : true
   }
}
```

Figure 4.8: Response to a request for an access token to a protected resource, before interaction with the resource owner

- **Instance ID** is a reference to metadata about client information that the AS has stored, an opaque value to the client.
- **Continue** is used when the AS has validated the previous request and further allows the client to negotiate with the AS. The AS sends back the URI, which is required for the AS to handle continuation requests, as well as a bound access token. This serves as the secret that should be used when continuing the request negotiation.
- **Interact** is a field to describe interaction. This interaction is the one proposed by the client in the first request and if the AS deemed it valid it is reflected here. The redirect parameter in this part of the response is one the AS has generated to which it wants the user to be redirected. A nonce can be generated by the AS that the client later can use to validate the callback.

4.3.3 Continuing the Request

In GNAP, all information necessary for authorization can be sent as the first response from the AS to the client. However, this is seldom something desired by the parties involved. More than often the communication requires several rounds of transmission, e.g. in order to facilitate interactions or further negotiate requests. This brings us to the interaction part of the flow.

The interaction utilizes the access code granted from the previous request and includes it in the request then signs it with a detached JWS. With this, the RO is redirected to the interaction endpoint specified in the interaction URI that was provided by the client. As can be seen in Figure 4.9, the message contains the JWS and access code as well as the interaction referenced in the JSON body.

Accept:"application/json, application/*+json", Content-Type:"application/json",
Content-Length:"49",
Authorization: "GNAP TT3pbcqBba1Yw86hFVz0v3pEbQZ9H66uO7X9vx9Fqrbrjx
cxnNbv2tGRHSG4LiOz",
Detached-JWS: "eyJodG0iOiJQT1NUIiwiYXRfaGFzaCI6InZBeEtfTnFWNVR0eUh
TTkdVejNoblEiLCJiNjQiOmZhbHNlLCJjcml0IjpbImI2NCJdLCJraWQiOiJ4..."

Figure 4.9: The access code and detached JWS of the request.

4.3.4 Response to Interaction

After the interaction between AS and RO has been verified, the implementation generates an access token for the protected resource along with a new continuation token. The only requirement for this part of the flow is that no new interaction fields can exist in the response. As seen when comparing Figure 4.10 and Figure 4.7 the client has received an access token with authorization to the scopes that the client requested, this can be seen in the access field in the access token.

```
1  "access_token" : {
2     "value" : "Lsfl5LDUZvjcvN2Tzj9M0i1acWW7tkCoAEaE2wtShGhISf...",
3     "bound" : true,
4     "access" : [ "openid", "profile", "email", "phone" ]
5  },
6  "continue" : {
7     "uri" : "http://localhost:9834/api/as/transaction/continue",
8     "access_token" : {
9         "value" : "K8lJcYIYFjlo1IUPrLzp7gx8oLYse0vnGECl47lH...",
10        "bound" : true
11        }
12 }
```

Figure 4.10: Response from the AS to the client after interaction has been finished with the RO.

The continuation token can be used for further negotiation of access to additional rights by the client. This makes the protocol flexible, since after receiving data the client can, for instance, prompt the user for additional requests it wants to make.

4.3.5 Requesting Resource

The implementation does not query for any resources. However, it uses its final access token to authorize against the resource server. This is done by providing both the access token as well as a detached JWS as seen in Figure 4.11. Figure 4.12 depicts the response from the RS where it presents the client with the scopes that it has been authorized for.

Accept:"text/plain, application/json, application/*+json, */*", Content-Length:"0",
Authorization: "GNAP Lsfl5LDUZvjcvN2Tzj9M0i1acWW7tkCoAEaE2wtShGhISfW
Wt2yPEf1Hmz84WJK9",
Detached-JWS: "eyJodG0iOiJHRVQiLCJhdF9oYXNoIjoiVV90RUtWcEJyVjFrSGto
cU4tT2FNZyIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il0sImtpZCI6Inh5ei1jbG
llbnQiLCJodHUiOiJodHRwOlwvXC9sb2NhbGhvc3Q6OTgzNlwvY..."

Figure 4.11: The access token and detached JWS of the request

```
1 "date" : "2021-03-16T08:43:09.525937300Z",
2 "overall_access" : {
3    "access" : [ "openid", "profile", "email", "phone" ]
4 },
5 "token_access" : [ "openid", "profile", "email", "phone" ],
6 "proof" : "JWSD"
```

Figure 4.12: Response from the RS

Following the protocol, the client instance has received access to the protected resource and all the required fields were present throughout the transaction. Thus, we can conclude that the implementation satisfies the specification.

4.4 Attack Software

In order to perform the attacks, some testing infrastructure had to be used in order to monitor and configure requests sent. For OAuth 2.0 and 2.1, the program Burp Suite (community edition) was used. Burp Suite is a tool for web penetration testing, created by Port Swigger Web Security. It functions as a man-in-the-middle service and is heavily used by penetrations testers in the industry. Burp Suite proxies all traffic to and from the browser through an interface, the user is able to monitor and alter the desired parameters [37].

For one special attack, additional software had to be implemented in order to facilitate the requirements needed to perform the attack. The software developed is an exploitation server that can mimic, alter, and forward messages. This software is used to test the AS mix-up attack described further in Section 4.5.5.

For GNAP however, Burp Suite is ineffective since little data are passed through the browser. Therefore, a separate proxy was set up to monitor and alter the data transmitted. An attempt to make an external program as a forwarding proxy was made. However, this presented some problems. Due to the use of detached JWSs, it is impossible to utilize a forwarding proxy without being in possession of the client's private key. This is because the property of detached JWS where both header and body is used as input for the signing, and in the header the specified host that should receive the request is present, making the signature dependent on the hostname of

the proxy [38]. If the proxy then alters any data, which is required in order to state where to send the request next, the signature no longer holds and the AS will drop the request. In order to solve this, we introduced some malware into the client itself. This means that between the signing of the request and transmission, the malware is able to alter any data it wants in the body. With this malware, we can now alter data and mimic attacks that we discovered in previous iterations.

4.5 Attacks

There are several documentations about common vulnerabilities against OAuth 2.0, mainly targeting easy misses in implementing the network [39]. However, since these attacks are due to wrongful implementation they essentially contradict the RFC and thus are outside the scope of the thesis [40, 41, 42, 43]. Disregarding the common implementation mistakes, three types of attacks are left to analyze:

- Cross-Site Request Forgery attacks.
- Vulnerable Redirect URI.
- Access code hijacking.

In addition to these attacks, we are interested in testing a more sophisticated attack described in "A Comprehensive Formal Security Analysis of OAuth 2.0" by Fett et al. [11]. This attack is called the *"AS mix-up attack"*[2] and focuses on manipulating the authorization grant flow as a whole instead of targeting smaller weak points like the former attacks.

The attacks found targeting the mentioned vulnerabilities in the systems will be explained in the following sections.

4.5.1 Redirect URI Attack

If an attacker manages to steal access codes, they can access the victim's data. Consequently, an attacker may log in as the victim on client applications that are registered with the faulty OAuth 2.0 service.

The authorization code flow sends the access code to a callback endpoint that is specified in the redirect URI parameter. Should the system fail to validate the URI, an attacker can trick a victim by letting them initiate a flow that sends the code to a redirect URI controlled by the attacker.

Given that an attacker successfully tricks the victim and gets a hold of a code, they may use this to either initiate token exchanging or resource exchanging without being authorized [44].

[2]In the paper the attack is called "IdP mix-up attack" since in the paper the authorization server is called identity provider (IdP).

4.5.2 Open Redirect Attacks

This attack targets two main things in the protocol. Firstly, the redirect URI and secondly, some input or directory that executes a URI as a path. By exploiting bad redirect URI matching the adversary can add "dot-dot-slash" (../) after the redirect URI, this is commonly known as a way to traverse up the directory tree [45]. If this is possible the adversary can traverse the directories to some path in the client in which a URI is executed as an input, say for example adding a textbox where you can enter your favorite website. By changing the redirect URI to traverse to an endpoint where an attacker can add arbitrary input, they can add a URI to an exploit server which can catch messages in transit [46].

For example, if the string matching rule parsing the redirect URI checks that the incoming redirect URI starts with `https://website.com/oauth-callback`. Then the adversary can traverse, in this case, to the *favoritewebsite* and pass the exploit server URI as a parameter; `https://website.com/oauth-callback/../favoritewebsite?url=https://exploit.server.com`. This URI will redirect the access code to the adversary's server, even with string matching present. The adversary carefully creates an attack URI, like the one mentioned, and then makes the victim click it leading to the access token being sent to the exploit server run by the adversary. This can then be used to gain the access token needed to retrieve the victim's protected resource [46].

4.5.3 Cross-Site Request Forgery

One of the most common attacks against web applications is the Cross-Site Request Forgery (CSRF) attacks, pronounced "sea surf". The main idea of the attack is to make a victim execute unwanted requests by using a link provided by the adversary. These links can be sent as a URI or be embedded in an image for example [47].

In the case of OAuth, the adversary can construct the start message of the flow with a redirection URI to a server that can catch the information. Then when the victim executes the query and fulfills the authentication, the access token will end up in the hand of the adversary, granting them the key for authorization [48].

4.5.4 Access Code Hijacking

While most attacks focus on redirecting the access code to the adversary in some way, this attack is a more direct approach. Instead of targeting the redirect URI, the attack focus on intercepting the access code in transit before it ends up in the hands of the user. Consequently, the adversary can start its own session, and instead of passing its own access code, it uses the victim's code. This will grant the adversary the access token with the victim's privileges and later access to the victim's protected resource. The actual interception of the code can be achieved through a malicious browser extension since the code is transferred through the front channel, i.e. the browser.

4.5.5 AS Mix-Up Attack

This attack tricks the user into thinking that an Attacking AS (AAS) is the client, while also tricking the client into thinking that the AAS is the user as well as the Honest AS (HAS) [11]. The attack flow is depicted in Figure 4.13, and works as follows:

1. This is done by intercepting the first message and changing that the user wants to access the AAS instead of the HAS.
2. This message is then sent to the client by the AAS.
3. After this, the client replies with its redirect URI, client Id, and state. The client Id is noted as ClientId* in Figure 4.13.
4. The redirect message from the client has then been modified again and the ClientId* is then exchanged for the AAS client id, clientId, and sent to the user.
5. Continue the protocol as normal.
6. Continue the protocol as normal.
7. Continue the protocol as normal.
8. Continue the protocol as normal.
9. The user is now in possession of the access code and promptly redirects it to the client since the redirect URI is pointing towards its endpoint.
10. The client, who thinks the AAS is the HAS, sends the normal token request to the AAS in order to retrieve a token.
11. Now the AAS is in possession of the ClientId registered at the AS, the correct redirect URI, and the access code. This can then be sent to the HAS for token retrieval.
12. Since the request has all the right parameters the HAS validates the request and replies with an access token and the AAS now has access to the protected resource.

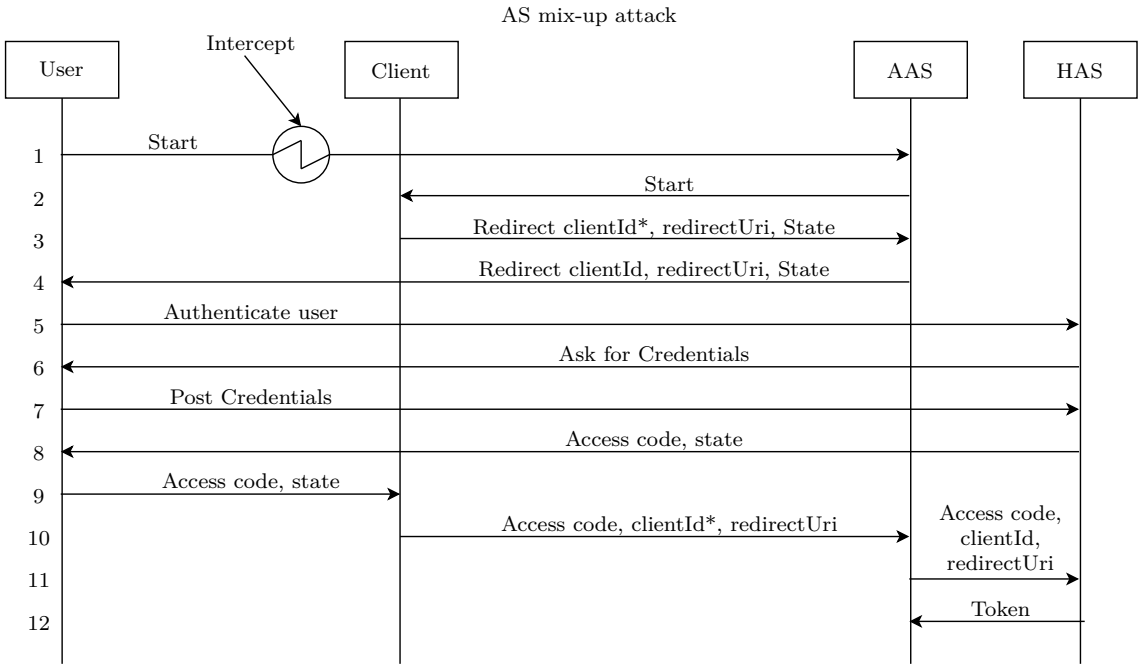


Figure 4.13: AS mix-up attack. AAS = Attacking Authorization Server, HAS = Honest Authorization Server, ClientId* = Id of the client

5
Results

In this chapter, the attacks are presented as well as the findings that the attacks resulted in. The chapter is structured attack by attack where the different frameworks are tested in ascending order, starting with OAuth 2.0 and ending with GNAP.

5.1 Redirect URI Comparison

OAuth 2.0

Testing the Redirect URI is straightforward, firstly, we checked the response from an ordinary request with the original redirect URI.

```
GET/home/auth?responseType=code&clientId=ClientApplication&
redirectUri=https://localhost:5001/home/oauth-
callback/&requestedScopes=product.read HTTP/1.1
```

The response was an OK response as following:

```
HTTP/1.1 200 OK
```

However, changing the redirect URI by adding something in the end yielded the same response:

```
GET/home/auth?responseType=code&clientId=ClientApplication&
redirectUri=https://localhost:5001/home/oauth-callback/testing-for-
breach&requestedScopes=product.read HTTP/1.1
```

Yielded:

```
HTTP/1.1 200 OK
```

OAuth 2.1

On OAuth 2.1 we tested this by sending the original and the modified authorize endpoint request. The original was not tampered with and was as following:

GET/connect/authorize?client_id=mvc&**redirect_uri=https://localhost:5003/signin-oidc**&response_type=code&scope=scope&code_challenge=code_challengeA&code_challenge_method=S256&response_mode=form_post&nonce=nonce&login_hint=Alice&state=state&x-client-SKU=ID_NETSTANDARD2_0&x-client-ver=6.7.1.0 HTTP/1.1

The response was:

HTTP/1.1 302 Found
Connection: close
Date: Mon, 29 Mar 2021 11:58:13 GMT
Server: Kestrel
Content-Length: 0
Location: https://localhost:5009/consent?returnUrl=/connect/authorize/callback?client_id=mvc&redirect_uri=https://localhost:5003/signin-oidc&response_type=code&scope=openidprofileemailoffline_accessproducts.readproducts.write&code_challenge=code_challenge&code_challenge_method=S256&response_mode=form_post&nonce=nonce&login_hint=Alice&state=state&x-client-SKU=ID_NETSTANDARD2_0&x-client-ver=6.7.1.0

A 302 redirection code indicates that the flow has not encountered any errors which will stop the flow. The application will subsequently redirect the user to the URI given in the "Location" header.

If we were to modify the redirect URI as done on OAuth 2.0 we get:

GET/connect/authorize?client_id=mvc&**redirect_uri=https://localhost:5003/signin-oidc/test-for-breach**&response_type=code&scope=scope&code_challenge=code_challengeA&code_challenge_method=S256&response_mode=form_post&nonce=nonce&login_hint=Alice&state=state&x-client-SKU=ID_NETSTANDARD2_0&x-client-ver=6.7.1.0 HTTP/1.1

Yields the result:

HTTP/1.1 302 Found
Connection: close
Date: Mon, 29 Mar 2021 12:03:21 GMT
Server: Kestrel
Content-Length: 0
Location: https://localhost:5009/home/**error?errorId=errorId**

Note how the status code is still 302 and it finds a proper URI. Instead, it redirects the client to an error page which indicates that the redirect URI is invalid, as stated in the RFC, which can be seen in Figure 5.1.

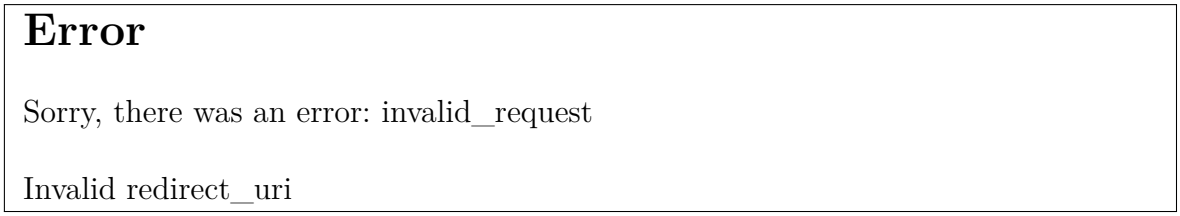


Figure 5.1: The error page when giving an invalid redirect URI to the AS.

GNAP

To mimic this attack we configured the malware to extend the redirect URI with "test-for-breach" to see if it was possible to bypass the string matching like in OAuth 2.0. The resulting message is depicted in Figure 5.2.

```
1 "interact" : {
2    "finish" : {
3        "uri" : "http://localhost:9839/api/client/callback/11
             G5HWDH6LltfFgWUSap2mxPwd810p/test-for-breach/",
4        "nonce" : "0clor9h5tHHK7Npyvz6D",
5        "method" : "redirect",
6        "hash_method" : "sha3"
7    },
8    "start" : [ "redirect" ]
```

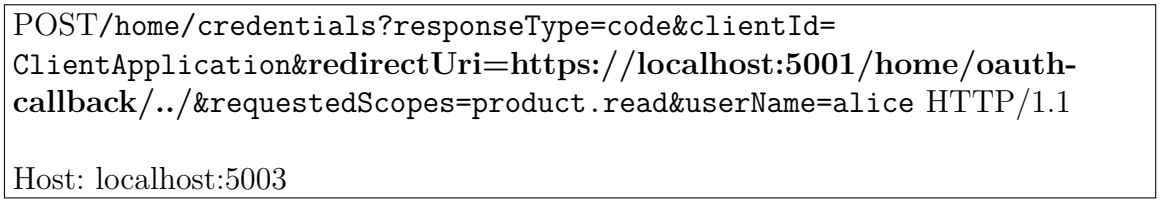
Figure 5.2: The interact field of a GNAP request where the URI has been altered with the extension of "test-for-breach".

The AS returns an "Unable to verify JWS" exception as expected since the content of the body was altered after signing.

5.2 Open Redirect

OAuth 2.0

To test the system's robustness against directory traversals, an interception of the requests was done to modify the redirect URI with a ../ attack.



Which essentially yielded the response:

HTTP/1.1 **302 Found**
Connection: close
Date: Tue, 30 Mar 2021 11:04:33 GMT
Server: Kestrel
Content-Length: 0
Location: `https://localhost:5001/home/`**`oauth-callback/../`**`?accessCode=accessCode&username=alice&role=Admin`

This indicates that the application is open to directory traversal attacks.

OAuth 2.1

As previously seen in Section 5.2, the intention is to test what happens when changing the redirect URI parameter in the authorization endpoint request to the AS. This was done by intercepting the request:

GET `/connect/authorize?client_id=mvc&`**`redirect_uri=https://localhost:5003/signin-oidc`**`&response_type=code&scope=scope&code_challenge=code_challengeA&code_challenge_method=S256&response_mode=form_post&nonce=nonce&login_hint=Alice&state=state&x-client-SKU=ID_NETSTANDARD2_0&x-client-ver=6.7.1.0` HTTP/1.1

The response was:

HTTP/1.1 302 Found
Connection: close
Date: Thu, 08 Apr 2021 11:37:11 GMT
Server: Kestrel
Content-Length: 0
Location: `https://localhost:5009/consent?returnUrl=/connect/authorize/callback?client_id=mvc&redirect_uri=https://localhost:5001/signin-oidc&response_type=code&scope=openidprofileemailoffline_accessproducts.readproducts.write&code_challenge=code_challenge&code_challenge_method=S256&response_mode=form_post&nonce=nonce&login_hint=Alice&state=state&x-client-SKU=ID_NETSTANDARD2_0&x-client-ver=6.7.1.0`

The original request indicates that the AS is behaving normally by giving the information that there is a path to redirect the user to. This can be seen with the 302 status code and the consequential location the user will be sent to.
Altering the redirect URI to incorporate the directory traversal we have:

GET /connect/authorize?client_id=mvc&**redirect_uri=https://localhost:5001/signin-oidc/../**&response_type=code&scope=scope&code_challenge=code_challengeA&code_challenge_method=S256&response_mode=form_post&nonce=nonce&login_hint=Alice&state=state&x-client-SKU=ID_NETSTANDARD2_0&x-client-ver=6.7.1.0 HTTP/1.1

Yields the result:

HTTP/1.1 302 Found
Connection: close
Date: Thu, 08 Apr 2021 11:42:25 GMT
Server: Kestrel
Content-Length: 0
Location: https://localhost:5009/home/**error?errorId=errorId**

The response from the altered request is similar to the original response. However, a keynote to mention here is that even though the server responds with a 302 status code, the location the user will be sent to is an error page which will indicate the faulty parameter as seen in Figure 5.3.

Error

Sorry, there was an error: invalid_request

Invalid redirect_uri

Figure 5.3: The error page when giving an invalid redirect URI to the AS.

GNAP

In order to replicate this attack in GNAP, the malware has to be configured to change the redirect URI after signing. This was done by injecting a dot-dot-slash after the actual redirect URI. The message sent has the same structure as Figure 4.7. In this message, the URI parameter in the finish field has been altered by the introduced malware. The result of the alteration is depicted in Figure 5.4 where it can be seen that the URI has been extended with the "dot-dot-slash".

```
"interact":{
    "finish":{
        "uri": "http://localhost:9839/api/client/callback/nonce
            /../",
        "nonce": nonce,
        "method":"redirect",
        "hash_method":"sha3"
    },
    "start": ["redirect"]
}
```

Figure 5.4: The interact field of a GNAP request where the URI has been altered with the extension of "../".

This alteration does not go over well with the AS and it throws an "Unable to verify JWS" exception thus showcasing that this attack is not possible in GNAP. The reason for this is that the URI is part of the input to the detached JWS and since the malware alters the URI after the signing phase, the AS cannot verify the JWS.

5.3 CSRF Attack Against Code Transaction Initialization

OAuth 2.0

In order to perform the CSRF attack against OAuth 2.0, Burp Suite was used to intercept request between the client and user agents. The goal with the attack is to identify if a user agent can finish the flow started by another user agent.

Firstly, we let the original user agent start the flow and intercept the start of the authentication endpoint:

```
POST /home/credentials?responseType=code&clientId=ClientApplication&
redirectUri=https://localhost:5001/home/oauth-callback&
requestedScopes=product.read&userName=alice HTTP/1.1
Host: localhost:5003
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:88.0)
Gecko/20100101 Firefox/88.0
```

This request was then sent in another browser to see if the other user agent could continue the flow. After executing the subsequent steps in the flow we get the response from the AS with an access code. Using this, we send a request to the client callback endpoint:

```
GET /home/oauth-callback?accessCode=b30be338efa14f74a40949dbafd02b57&
username=alice&role=AdminHTTP/1.1
Host: localhost:5001
Connection: close
sec-ch-ua: "Chromium";v="89", ";Not A Brand";v="99"
```

The response from doing the GET request:

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 11 May 2021 07:37:54 GMT
```

The response indicates that the client accepted the request with an access code sent from another user agent than the original one. Therefore, the system is open to this particular type of CSRF attack.

OAuth 2.1

The OAuth 2.1 system already has countermeasures to CSRF attacks. To make sure these works we tested how the system behaves if we were to act as an attacker client and send an arbitrary state to the AS and see if the system still behaves normally. In step (3) we intercept the request to the authorization server to be able to modify the state parameter.

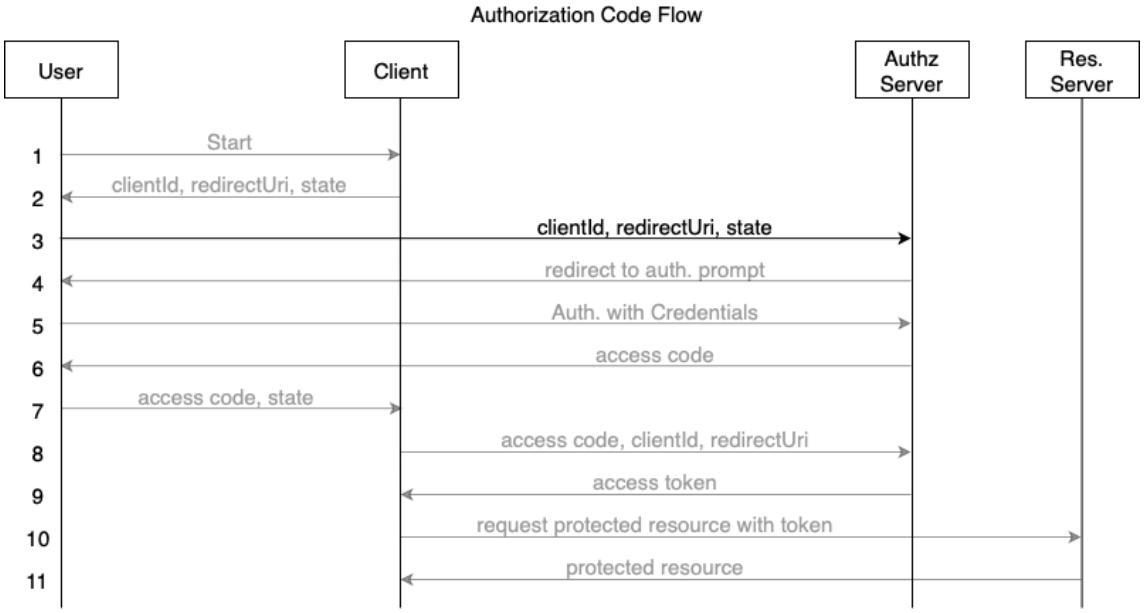


Figure 5.5: The step of where the attacker intercepts the request

```
GET/connect/authorize?client_id=mvc&redirect_uri=redirect_uri&
response_type=code&scope=scope&code_challenge=code_challengeyQ&code_
challenge_method=S256&response_mode=form_post&nonce=nonce\\&login_
hint=Alice&state=state&x-client-SKU=ID_NETSTANDARD2_0&x-client-ver=
6.7.1.0 HTTP/1.1
```

Figure 5.6: We have substituted the values of all parameters since they do not have any meaning and are mostly opaque values.

After intercepting this we simply change the state value to 124356 and send this to the AS.

```
GET/connect/authorize?client_id=mvc&redirect_uri=redirect_uri&
response_type=code&scope=scope&code_challenge=code_challengeyQ&code_
challenge_method=S256&response_mode=form_post&nonce=nonce\\&login_
hint=Alice&state=123456&x-client-SKU=ID_NETSTANDARD2_0&x-client-
ver=6.7.1.0 HTTP/1.1
```

Using this URI will get us to step 7 (seen in Figure 5.7), but at this point the response from the Client was:

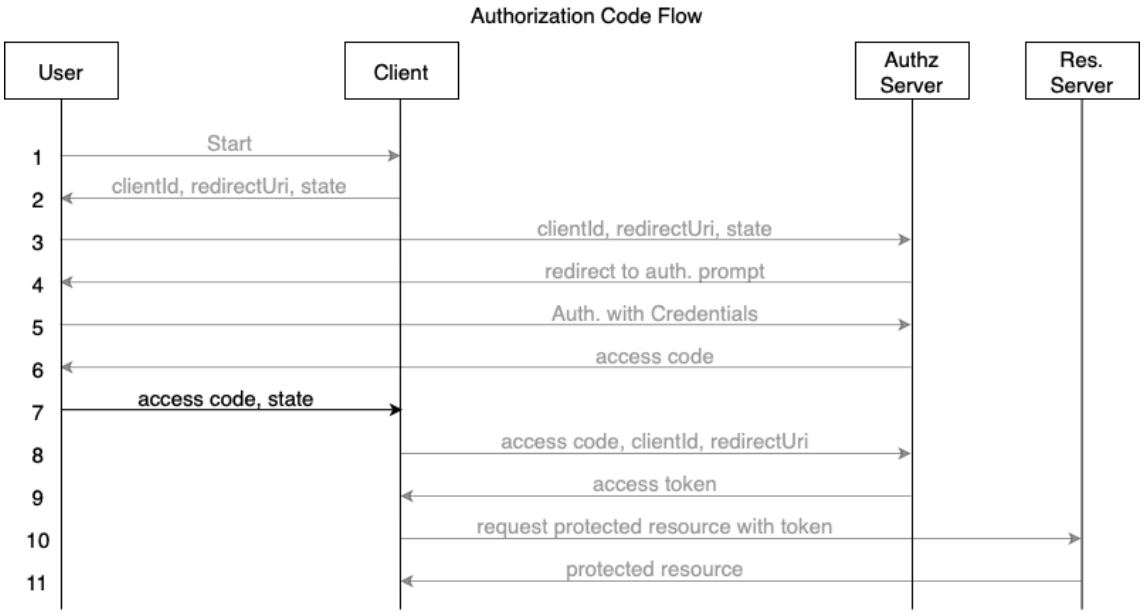


Figure 5.7: The step where the error occurs

```
HTTP/1.1 500 Internal Server Error
```

No error message was received on the web client. However, analyzing the logs from the client application shows the message

```
Error from RemoteAuthentication: Unable to unprotect the message.State.
```

This indicates that there is protection for CSRF attacks since the state parameter was not issued by the client and therefore unrecognizable by the latter.

GNAP

In order to test the system for CSRF attacks, we can use Burp Suite since it is the front-end that the attacker has access to. The first potential attack vector is where the adversary makes the victim interact with and approve a request which it has not created. To do this we used two separate browsers and requested a continuation token for the adversary which can be seen in Figure 5.8.

```
Continuation Token: FgdweTXoTWuNJn0M39MPh2tbJ0Df...
Interaction URL: http://localhost:9834/api/interact/L7Zm1KVYvl
```

Figure 5.8: The prompt after requesting a continuation token from the client. The token has been cut for easier readability and is not a value of importance.

We started the interaction but dropped the message in order to pass the URI to the victim instead, this can be done through social engineering where the adversary makes the victim click on the link. When the URI is interacted with, the URI will be executed in the victim's browser.

```
http://localhost:9834/api/as/interact/L7Zm1kvyVL
```

Since the AS can handle arbitrary URIs and the adversary has already contacted the AS to have a pending interaction waiting on this address the URI is valid and the victim ends up where she can choose to accept or deny the request. However, after trying to accept the adversary's request the client throws an error. The reason for this is due to a failure in the hash validation. The AS then redirects the user back to the client with a hash as well as an interaction reference which the client has to validate in order to know it was originally issued by itself. The structure of the hash interaction is the following:

```
interaction hash = HASH(client nonce, server nonce, interaction reference)
```

The client is then in possession of the interaction reference as well as the hash but in order to validate it needs access to the client nonce and server nonce. These are stored in the session of the user's client interaction from the initial request done to the client. Since the victim has not stored these crucial values in its session it is unable to verify the hash value and rejects the request. This prevents the adversary from performing CSRF attacks against the interaction phase of GNAP.

5.4 Access Code Hijacking

OAuth 2.0

To be able to hijack an access code from a victim utilizing the OAuth 2.0 framework there has to be a way for the adversary to get hold of the access code before the user sends it to the client. This could for example be done with a malicious extension. In our attack, we mimicked such an extension by intercepting the message between AS and the user with Burp Suite to gain possession of the access code. With this code, we could just query the AS with the appropriate parameters to gain an access token meant for the victim.

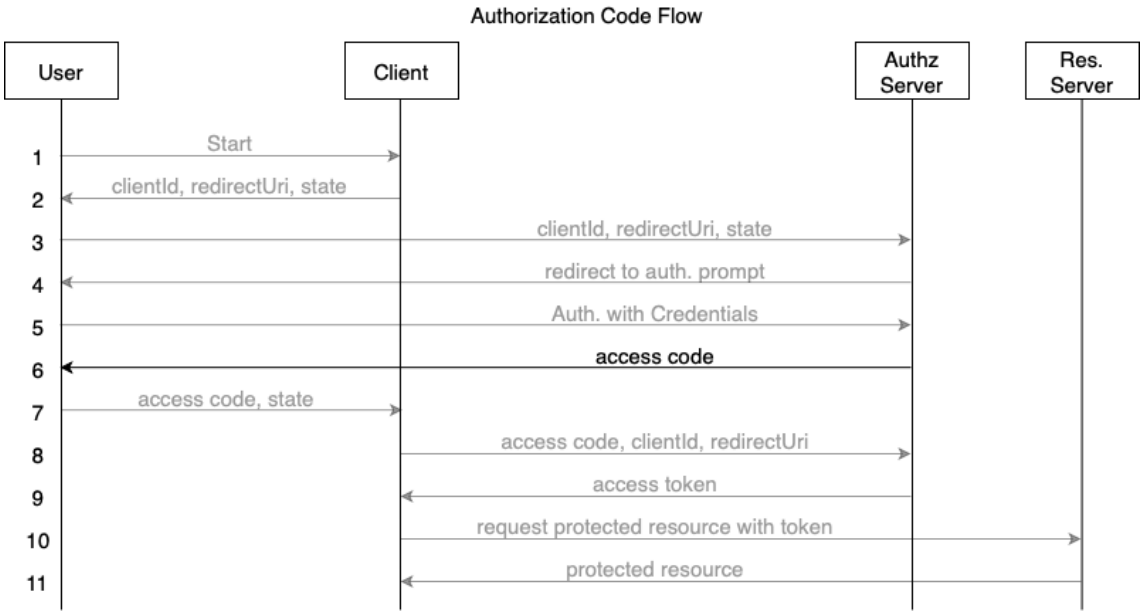


Figure 5.9: Step 6 is where the attacker intercepts and hijacks the access code

OAuth 2.1

In order to perform the attack on OAuth 2.1, we assume that there exists a malicious extension that can intercept messages from the AS. As done on OAuth 2.0 we will intercept the message where the AS sends an access code to the end-user and use that to get an access token.

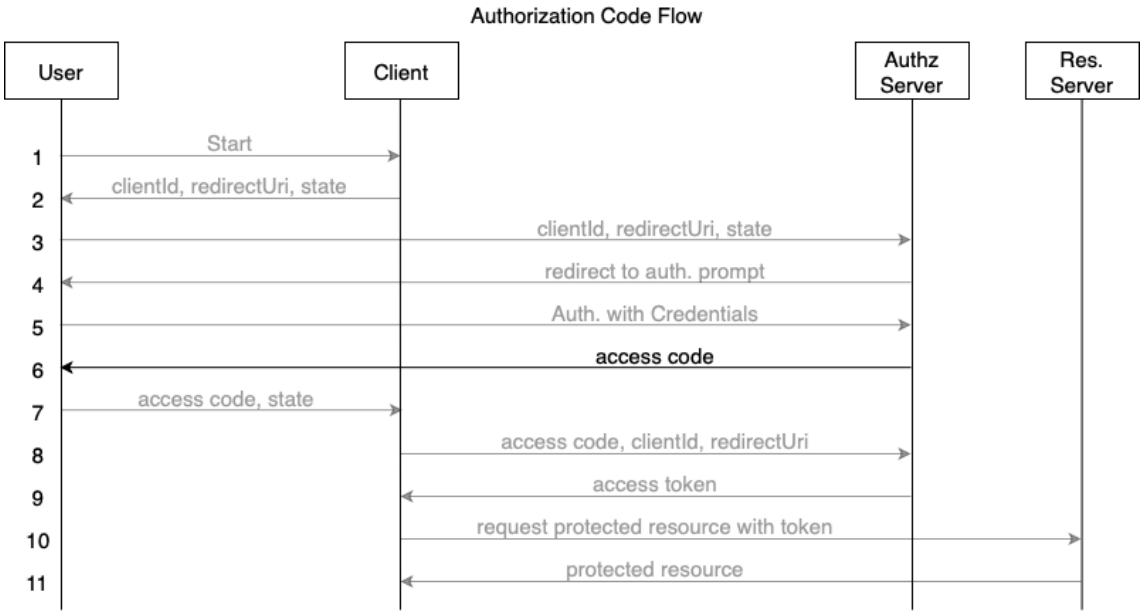


Figure 5.10: Step 6 is where the attacker intercepts and hijacks the access code

Initially, the victim starts the flow regularly, and then at the point where they are sent the access token, we sniff the code and drop their connection with the AS.

HTTP/1.1 200 OK
Connection: close
Date: Mon, 12 Apr 2021 11:39:49 GMT
Content-Type: text/html; charset=UTF-8

code=**code**
scope=**scope**
state=**state**
session_state=**session_state**

Figure 5.11: The response from the AS after entering correct credentials. The actual values are opaque and are not of importance, so only parameter's name are presented to show what is sent back.

The stolen code will be used with the necessary parameters to call for a token from the AS. The required parameters are the client id, redirect URI, grant type, the access code, and lastly the code verifier. We do not have the code verifier since this is a secret between client and AS which we have no knowledge about. However, we need to mock some kind of code verifier and this is done by using an old code verifier from a proper request cycle.

```
POST /connect/token HTTP/1.1
Host: localhost:5009
Content-Type: application/x-www-form-urlencoded

code=code&client_id=client_id&grant_type=authorization_code&redirect_
uri=https://localhost:5009/signin-oidc&code_verifier=old_code_
verifier
```

Figure 5.12: The POST request to the AS with the corresponding values of the parameters needed in order to poll for an access token

Sending this POST request will firstly give us:

```
HTTP/1.1 400 Bad Request

error: invalid_grant
```

The result from this request was that the code verifier did not match the code challenge given at the start of the flow, and thus one cannot hijack the access code on an OAuth 2.1 without also breaking the PKCE challenge.

The error was logged to a console:

```
Transformed code verifier does not match code challenge
```

GNAP

Access codes do not exist in GNAP by name, however, there exists a parameter that can be seen as somewhat similar, the interaction reference. This value is created by the AS after the interaction has concluded. This value might be accessed through the user agent, however, if this is done the protocol clearly states that how it is done is out of scope. An important feature of the interaction reference is that when it is created by the AS it is also bound to the session that started the flow in the first place. This means that even if an adversary managed to hijack the interaction reference she would not be able to use it since it would not be bound to her session. To test this attack, both Burp Suite and Postman were used. Postman can be used to send custom HTTP messages to the target system in order to test the different endpoints it has. Through Burp Suite we intercepted the final interaction reference before it was sent to the AS, since this value can only be used one time it is imperative that it is dropped between the client and the AS. Subsequently, we sent the intercepted code as a post request in postman to see if we could get hold of the token. The AS responded with a 404 error code which indicates that the AS has denied us the token.

5.5 AS Mix-Up Attack

OAuth 2.0

The mix-up attack required a bit of a system overhaul. Firstly, the authorization server location was set dynamically, giving the system the opportunity to interact with an authorization server chosen by the user. Secondly, a man-in-the-middle server that acts as a client and authorization server was created. After the modifications to the system, the attack could be properly performed. The attack was done by intercepting the first request done by the flow; the start request.

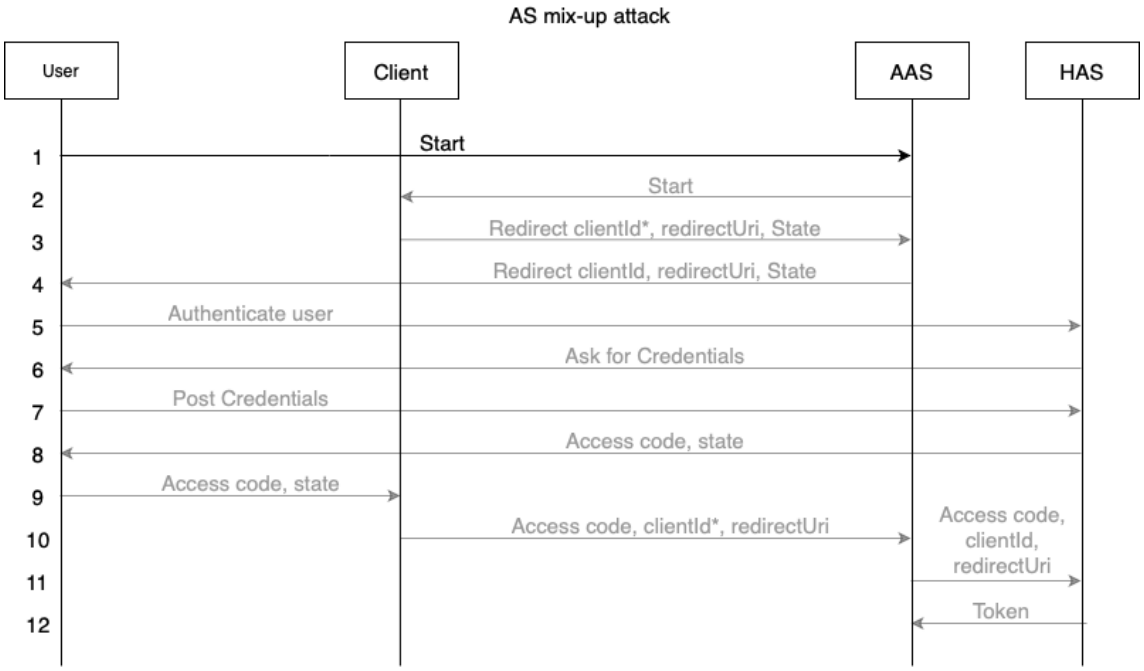


Figure 5.13: The start request is the first intercepted message. *clientId** is the id for the original client, while clientId is the attacker id

The start message now incorporates the server against which the user wants to authorize. After intercepting the message the server is changed from targeting the HAS, to instead target the AAS.

```
GET /home/start?authServer=https://localhost:5004/home/ HTTP/1.1
Host: localhost:5001
Referer: https://localhost:5001/home/
Cookie: cookie
```

The response was:

HTTP/1.1 302 Found
Connection: close
Date: Wed, 21 Apr 2021 11:29:57 GMT
Location: `https://localhost:5004/home/auth?responseType=code&clientId=ClientApplication&redirectUri=https://localhost:5001/home/oauth-callback&requestedScopes=product.read`

Consequently, the AAS will send the user to the HAS where it will continue the flow as it is, but with the AAS's client id rather than the original client id, until the client requests an access token. The client will proceed with the impression that it is communicating with the HAS and subsequently send the access code accordingly. However, since we altered the AS which the client communicates with it will send the request to the AAS. The AAS will take the access code and request for an access token itself, and since the HAS has no established security protocol against this, it will just check that the AAS is pre-registered and that the access code is correct. The result is that the AAS claims the access token while the client encounters an exception.

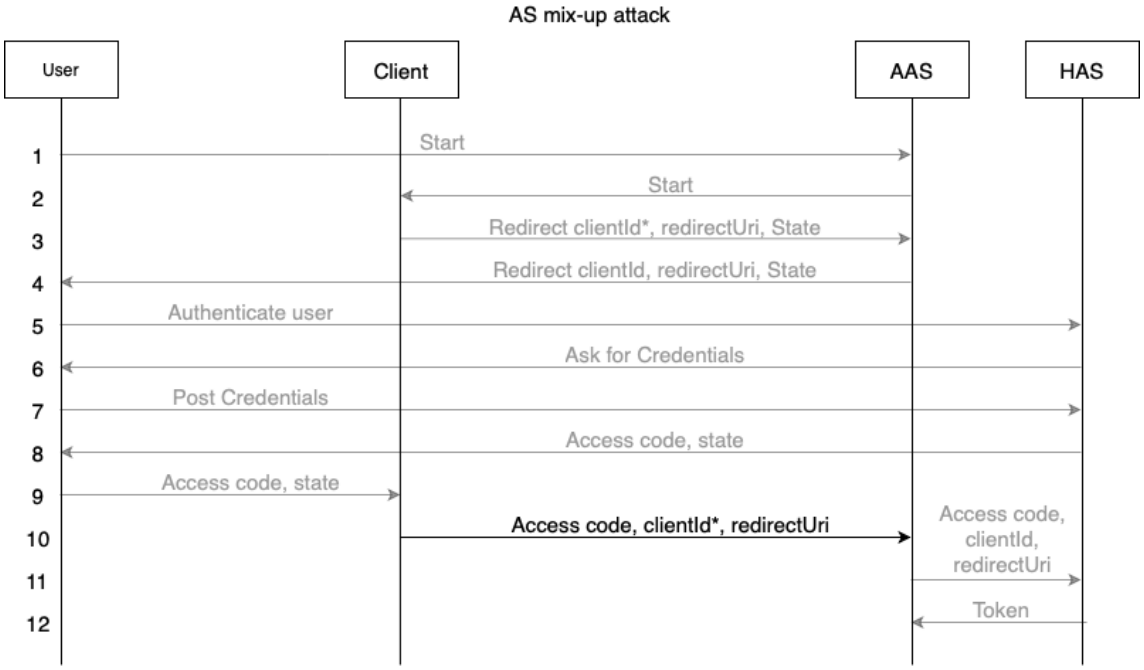


Figure 5.14: The start request is the first intercepted message

The results from running the flow were logged to the console. The logs revealed that we were able to get the access code from the client and use that code to successfully request a token from the HAS.

Access Code: 941c377c73c0efed759c993f1b859526
Access Token: c2c19dfb08b27e37e3508427267f7d95

OAuth 2.1

Due to the fact that in the RFC of OAuth 2.1 there is a specific clause to handle just mix-up attacks, there is no physical test of the attack performed in the thesis. The attack would have been similar to the OAuth 2.0 attack but, as stated before, OAuth 2.1 has security measures specifically against mix-up attacks. OAuth 2.1 forces the client to store the authorization server it requested authorization against, the session that the user agent created when starting the flow. This means that since the AAS initiates the flow, the client will recognize that the entity posting the access code differs from the AAS since its origin is the user agent. This is depicted in Figure 5.15.

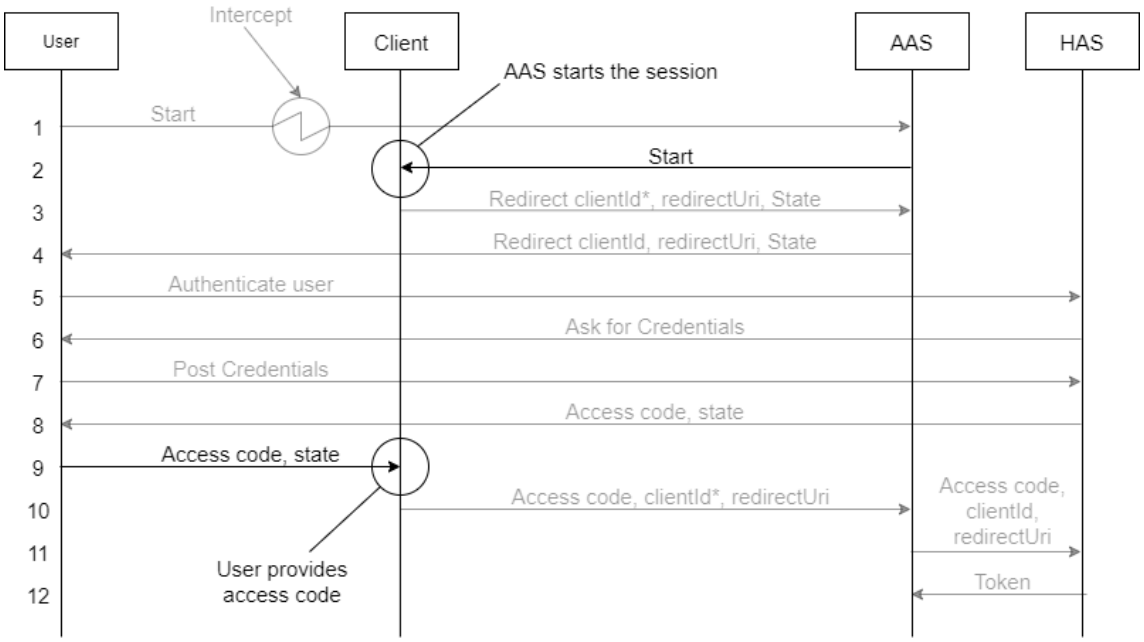


Figure 5.15: Mix-up attack in OAuth 2.1 is prevented since there is not the same session sending the access code as the one starting the flow.

GNAP

The AS mix-up attack is built upon a bold assumption that the client can dynamically choose what AS to authorize against. This assumption may not be applicable to real life, however, it was argued for and used as an assumption in the mix-up attack in Fett et al. [11]. Using the same assumption we set up the following attack.

1. The user starts an interaction with the client as normal.
2. The client is miss configured so instead of accessing the HAS it starts the flow by sending a request to the AAS. In the request from the client to the AAS, the most important fields of the request are depicted in Figure 5.16. The client nonce is created here and is important for hash validation later in the flow. The interaction finish URI is also created here so the HAS knows where to redirect the interact reference later.

3. The AAS now relays the request to the HAS but with its own public key so the HAS can validate the signature of the message.
4. After validating the request the HAS responds with an interaction URI as well as a server nonce used for hash validation later.
5. Both these values are then relayed back to the client by the AAS.
6. Since there was a need for interaction the user agent is now redirected to the HAS interaction URI by the client.
7. Interaction occurs with the HAS, notice that this is the actual AS that the user agent wanted to authorize against.
8. After the interaction is finished the HAS returns the hash challenge as well as the interaction reference to the client. Since the client now is in possession of the client nonce, server nonce, and interaction reference it can validate the hash and determine that nothing suspicious has occurred.
9. The client sends the interaction reference to the AAS.
10. The AAS sends this interaction reference to the HAS. This interaction reference is bound to the key of the AAS and is therefore validated at the HAS and it responds back with a token.
11. The AAS is now in possession of the token and the attack is complete.

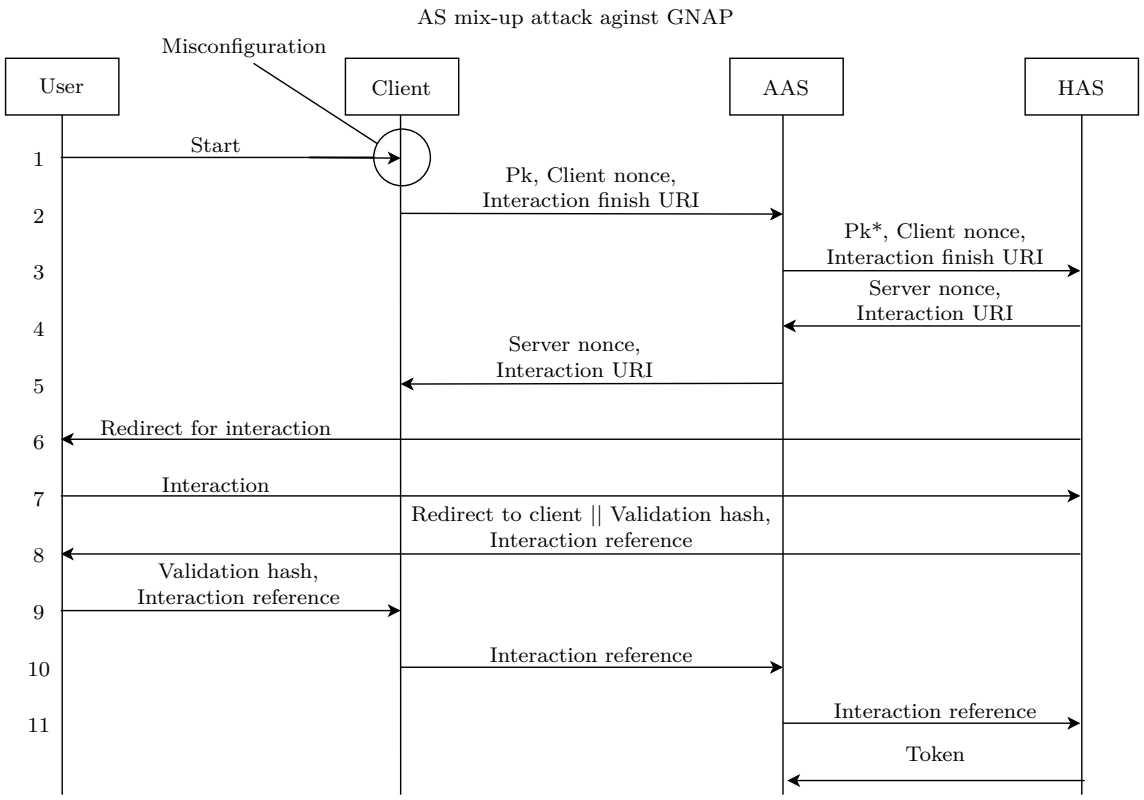


Figure 5.16: The AS mix-up attack against GNAP. Pk = client public key, Pk* = AAS public key

Summary
Under the circumstances the tests were performed, the thesis indicates that OAuth

2.1 has protection against all the attacks tested while GNAP is vulnerable to the AS mix-up attack.

6
Discussion

The purpose of this chapter is to discuss the results from the tests described in previous chapters. The focus will be on GNAP and how further development can affect the protocol. The chapter also considers the juxtaposition between OAuth 2.1 and GNAP. Moreover, the methodology is dissected, where a top-level analysis of the attacks is presented.

6.1 The Attacks

The main part of the thesis is the findings and results from performing the attacks. Redirect attacks and CSRF attacks are common flaws in bad OAuth implementation, while GNAP and OAuth 2.1 provides protection against them. Due to this, there is no need for a lengthy discussion about the results from these. However, cases like the AS mix-up and access code hijacking are in need of some deeper exploration. The reason for this is that; firstly, the hijacking could not be mapped straight forward to GNAP since the way the access codes are transferred works differently. Secondly, the mix-up attack revealed a vulnerability in the protocol design, which we consider interesting to explore further in the future.

6.1.1 Open Redirect and Redirect URI Attack

These attacks are very similar and both focus on manipulation of the redirect URI and are prevented in the same way, thus they are discussed as one. They are prevented by the signature of the requests sent. This is one of the strengths of signatures and hence GNAP, that common attacks like these should no longer work. Furthermore, there are additional features in GNAP that protect from these redirect modification attacks. In GNAP the AS is required to, as in OAuth 2.1, compare the redirect URIs with exact string matching so even if there were some bitflip or misconfiguration the AS should reject a redirect URI that it cannot exactly string match to that session.

6.1.2 CSRF Attacks

CSRF attacks are prominent in systems that are heavily reliant on the web browser. By design, GNAP has no way to inject input, thus making it difficult to perform this attack. However, if the web application has a method to inject any type of input in a manner that is passed to the flow, GNAP takes advantage of its use of cryptographic signing in message passing. With the interaction hash implementation,

GNAP prevents these attacks confidently.

GNAP's interaction hash provides mitigation against a typical CSRF attack, given that the interaction requires the adversary to be in possession of both the client- and server nonce. Given these components, the adversary's only opportunity to trick the victim into a CSRF attack is to get a hold of them to break the interaction hash.

6.1.3 Access Code Hijacking

As stated in the result, the traditional access code does not really exist in GNAP. Instead of after authorization and authentication from the user, the client is in possession of the actual token. Intercepting the interaction reference does not work. Tokens in GNAP are passed through back-channel networks, so gaining access to them is difficult compared to in OAuth. The attack instead mimics an attempt of stealing the access token instead.

Since tokens are passed over TLS, it is unlikely that an adversary can hijack the access token. There is no encryption stated in the protocol, making it infeasible to catch messages in transit given that TLS is implemented correctly. Another method to gain access tokens is to reference the token request from the user agent or have malware that can listen to the port's traffic. This is considered wrongful implementation and was not further explored.

There is a possibility of hijacking an access token. Assuming that the adversary has access to the server the client is running on, intercepting an access token is possible. However, if the access code is visible it has to be bound to the client instance's key. Furthermore, requesting resources is done through the back-channel, which requires a private key to be able to use an access token. This is not feasible since it breaks the protocol in every way, and if the private key is in the possession of the adversary, it is game over.

In the GNAP implementation, we can use Burp Suite to intercept the reference to the request from one user to another. Copying this reference URI to another browser and executing it can get us the desired data. However, this is one of the essential features of the authorization protocols. Meanwhile, the token's expiration time has not yet run out, every user agent should be able to use the access token to acquire the resource. This feature is for the usability of the application adopting these protocols, such that a user is not forced to accept the service multiple times in different browsers.

6.1.4 AS Mix-Up

The mix-up attack discovered is by far the most interesting one in the thesis since this is the only vulnerability found when performing legacy attacks. The attack is built upon a rather bold assumption. Although bold, it is not non-existent and therefore cannot be overlooked when performing an audit for attack vectors. The

attack exploits the openness and the new negotiation feature of the protocol.

GNAP is very well protected from the AS's viewpoint since all the messages are signed. This means that the AS is always sure that the information received is exactly the same as it was when it was first transmitted. However, more importantly in this case, the AS always knows who it is talking to.

On the contrary, there is no signing from AS to the client. Thus, the redirect from the interaction is not checked to make sure that it came from the intended authorization server. This leads the client to redirect the user to authorize against another AS than the client has established; a mix-up. Then when the reference is retrieved and the hash value is verified, the client sends the interact reference to the AAS instead of the AS that the user authorized against.

6.2 Client Malware

In the practical tests for GNAP, the client follows an honest-but-curious (HBC) model, where the client never deviates from the protocol but will also interact with all legitimate received messages [49]. The reason for using this model is that an AS should be considered as a trusted entity. It can be treated with the same respect as a certified authority in TLS, meaning that an honest AS will not produce malicious results.

On the contrary, clients are much more vulnerable to malware. Clients are developed independently of the AS and by the structure of authorization protocols, there exist more clients than authorization servers. Therefore, this accessibility along with human error makes manipulating them more feasible and worth exploring.

6.3 OAuth 2.1 vs GNAP

The following paragraphs explain the top-level differences between OAuth and GNAP.

Flexibility

Flexibility is something that is an impactful difference GNAP applies, giving the users more ways to start and finishing interactions. OAuth 2.1 currently relies on the user having access to web browsers in order to start interaction with an AS. GNAP is set to give the user the opportunity to use different interaction methods, e.g. mobile device or entering a user code.

The majority of the weaknesses in OAuth come from the combination of utilization of URI query parameters and its dependence on the browser as a way to transfer information. GNAP instead, utilizes the body of HTTP requests which is far more flexible compared to the URI since it enables features such as signing, encryption,

and more data structures.

Additionally, GNAP offers a dynamic way of handling grants. OAuth typically has to start new grants if circumstances change during the flow. GNAP however, can provide a grant identifier if the server determines that a grant can be continued. This means that if the client had "write" and "read" access but realizes that it does not need "write" anymore, it could change the scope and the authorization server will respect this request and since the breadth of the scope is reduced, it will not require authentication.

Interoperability

A strong difference between OAuth 2.0 and GNAP is how strongly bound the AS and the RS are in the former. GNAP's interoperability is designed to decentralize identity standards while providing an authorization layer more in line with human-centered design. This allows GNAP to support different communication patterns between resource servers and authorization servers with added extensions [35].

GNAP allows the requesting client to determine what kind of interaction it supports. The AS will then create this interaction so that it is used to communicate with the RO. Currently, GNAP allows for interaction through redirecting to arbitrary URLs, an app on a mobile or user code which can be displayed [35].

Cryptographic-based design

A very apparent reason why the attacks tried in this thesis fails on GNAP is the cryptographic-based design. Most communication in GNAP is tightly bound to a key held by the client instance which is used for signing all messages. It uses this design to authenticate the client as well as binding the access tokens. Since GNAP is not built on a specific signature schema, but instead just provided some examples, GNAP is well prepared for future signature-based schemes as they are introduced with time.

6.4 Future Work

This thesis has explored some of the most common attacks from OAuth 2.0. However, more attacks than these exist. For example, attacks targeting the scope parameter [50]. Further testing of legacy attacks could be of interest for future work on GNAP.

Furthermore, a niche subset of the industry is calling for the use of OAuth in what is called Finacial-grade APIs, FAPIs [51]. These are utilized by industry sectors that require higher security like the financial sector, e-government, and e-health [52]. However, in order to make OAuth cope with the requirements a new profile had to be developed with where new extensions and practices are introduced. How well GNAP holds up to the requirements of FAPIs is of much interest if the protocol

should ever go mainstream [52].

The thesis has focused on a subset of configurations in GNAP. For example in the case of the means of interaction, the redirect URI was explored since this best mimics the OAuth frameworks. However, there exist more ways for the RO to interact with the AS. Further exploration of the different ways data can flow in GNAP is of utmost interest.

Lastly, as stated in the thesis, GNAP is inspired by the OAuth frameworks. This inspiration was our reason for testing existing attacks against these frameworks on the new protocol to see how well it could cope against them. However, these are not the only authorization frameworks used. For example, Security Assertion Markup Language (SAML) is another authorization standard for web applications [53]. Looking into the vulnerabilities of these standards might provide more insights into the security of GNAP.

Recommendations

We believe that GNAP has the potential to become the next generation authorization protocol. It resolves many of the issues OAuth 2.0 has in a more modern manner compared to OAuth 2.1, which makes it interesting to follow the development of the protocol further.
Security protocols strive to be proven completely secure, which is almost impossible in a protocol as flexible and diverse as GNAP. However, with more external analysis the goal can perhaps be somewhat reached. Therefore, we encourage the community to further explore, investigate, and develop this protocol.

7
Conclusion

The purpose of this thesis was to explore GNAP and its strengths against legacy attacks. This part of the thesis presents the answers to the questions given in the problem statement, as well as some conclusions drawn from the attacks performed.

How robust is GNAP against legacy attacks?

According to our research, we can conclude that GNAP is well prepared for the most common vulnerabilities that OAuth 2.0 was exposed to. However, it still has its shortcoming when it comes to the mix-up attack, which OAuth 2.1 has protection against.

How does GNAP handle the vulnerabilities differently than OAuth?

By incorporating signatures in the message passing between the client and the AS, GNAP makes alteration of data hard from external sources. The thesis demonstrates that in our scenarios, even with the malware running on the client system, if the messages are correctly signed it is well protected against data alteration from an adversary. For example, attacks manipulating redirect URIs in OAuth 2.1 are done by exact string matching. While GNAP does this as well as check the signature, providing a deeper level of protection.

CSRF attacks which are a common attack vector in OAuth are handled mostly the same in both GNAP and OAuth 2.1; in GNAP the requests are bound to a session-id so if a user agent with the wrong session-id tries to perform a forged request it is denied. This is more or less the same in OAuth 2.1 where this value is stored in the state parameter.

Furthermore, our tests show that GNAP is not protected against the mix-up attack given our environment. OAuth 2.1 solves this problem by checking that the user agent which issued the initial request is the same agent providing the access code. Even though GNAP failed to protect against the mix-up attack it is important to note that the thesis performed the attacks against the fifth draft of the RFC of GNAP. While GNAP is not completed, the underlying thoughts about combining ideas from OAuth with signatures and more flexibility make it promising for the future of authorization on the Internet.

Summary

Both GNAP and OAuth 2.1 are robust against the attacks we performed. However, GNAP is vulnerable to the AS mix-up attack. GNAP is still in development and the findings from the thesis was reported to the working group. We will continue having discussions with the working group with the hope that the thesis contributes to their great work.

Bibliography

[1] G. Levin. 4 most used rest api authentication methods. https://blog.restcase.com/4-most-used-rest-api-authentication-methods/. Accessed 2020-12-11.

[2] F. B. Schneider. Least privilege and more [computer security]. *IEEE Security & Privacy*, 1(5):55–59, 2003.

[3] S. Kosten. Security authentication vs. authorization: What you need to know, 2020. https://towardsdatascience.com/security-authentication-vs-authorization-what-you-need-to-know-b8ed7e0eae74, Accessed 2021-01-25.

[4] Z. Grossbart. What you need to know about OAuth2 and logging in with facebook. https://www.smashingmagazine.com/2017/05/oauth2-logging-in-facebook/. Accessed 2021-01-20.

[5] D. Hardt. Rfc 6749: The OAuth 2.0 authorization framework, 2012.

[6] M. Raible. What the Heck is OAuth? https://developer.okta.com/blog/2017/06/21/what-the-heck-is-oauth, 2017. accessed 2020-12-01.

[7] Is OAuth enough for financial-grade api security? https://nordicapis.com/is-oauth-enough-for-financial-grade-api-security/. Accessed 2021-01-20.

[8] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett. OAuth 2.0 security best current practice, 2018.

[9] J. Richer. The case for OAuth 3.0. https://justinsecurity.medium.com/the-case-for-oauth-3-0-5c7537e3f9c3. Accessed 2020-11-30.

[10] Xyz: Transactional authorization. https://oauth.xyz. Accessed 2021-01-21.

[11] D. Fett, R. Küsters, and G. Schmitz. A comprehensive formal security analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1204–1215, 2016.

[12] F. Yang and S. Manoharan. A security analysis of the OAuth protocol. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 271–276. IEEE, 2013.

[13] Wanpeng Li, Chris Mitchell, and Thomas Chen. Oauthguard: Protecting user security and privacy with oauth 2.0 and openid connect, 01 2019.

[14] M. Shehab and F. Mohsen. Securing OAuth implementations in smart phones. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, New York, NY, USA, 2014. Association for Computing Machinery.

[15] R. Paul. OAuth and OAuth WRAP: defeating the password anti-pattern, 2010. https://arstechnica.com/information-technology/2010/

01/oauth-and-oauth-wrap-defeating-the-password-anti-pattern/, Accessed 2021-02-10.

[16] M. Biehl. The password anti pattern, 2015. https://api-university.com/blog/the-password-anti-pattern/, Accessed 2021-02-11.

[17] A.M Colman. *A dictionary of psychology*. Oxford quick reference, 2015.

[18] A. Parecki. Differences between OAuth 1 and 2. https://www.oauth.com/oauth2-servers/differences-between-oauth-1-2/, Accessed 2021-02-10.

[19] OAuth0. Which OAuth 2.0 flow should i use? https://auth0.com/docs/authorization/which-oauth-2-0-flow-should-i-use, Accessed 2021-01-22.

[20] D. Moore. What's new in OAuth 2.1? https://fusionauth.io/blog/2020/04/15/whats-new-in-oauth-2-1/, 2020. Accessed 2020-12-11.

[21] D. Lindau. 8 vital OAuth flows and powers. https://nordicapis.com/8-types-of-oauth-flows-and-powers/. Accessed 2020-12-04.

[22] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu. Vulnerability assessment of oauth implementations in android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 61–70, 2015.

[23] J. Richer. Grant Negotiation and Authorization Protocol draft 00, 2020. https://tools.ietf.org/html/draft-ietf-gnap-core-protocol-00, Accessed 2021-02-01.

[24] San-Tsai Sun and K. Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 378–390, 2012.

[25] D. Hardt M. Jones. The OAuth 2.0 authorization framework: Bearer token usage. Technical report, RFC 6750, October, 2012.

[26] T. Lodderstedt and M. Scurtescu. Oauth 2.0 token revocation. *IETF RFC 7009*, 2013.

[27] T. Kawasaki. Implementer's note about JAR (JWT secured authorization request), 2020. https://darutk.medium.com/implementers-note-about-jar-fff4cbd158fe, Accessed 2021-02-11.

[28] T. Lodderstedt. Rich OAuth 2.0 authorization requests, 2019. https://medium.com/oauth-2/rich-oauth-2-0-authorization-requests-87870e263ecb, Accessed 2021-02-11.

[29] A. Parecki. It's time for OAuth 2.1, 2019. https://aaronparecki.com/2019/12/12/21/its-time-for-oauth-2-dot-1, Accessed 2021-02-12.

[30] OAuth. Oauth 2.1. https://oauth.net/2.1/, Accessed 2021-02-12.

[31] OAuth0. Authorization code flow with proof key for code exchange (PKCE). https://auth0.com/docs/flows/authorization-code-flow-with-proof-key-for-code-exchange-pkce, Accessed 2021-02-10.

[32] S. Brady. Why the resource owner password credentials grant type is not authentication nor suitable for modern applications, 2017. https://www.scottbrady91.com/OAuth/Why-the-Resource-Owner-Password-Credentials-Grant-Type-is-not-Authentication-nor-Suitable-for-Modern-Applications.

[33] A. Parecki. Refreshing access tokens. https://www.oauth.com/oauth2-servers/access-tokens/refreshing-access-tokens/, Accessed 2021-02-12.

[34] A.C. Oliver. Gnap: Oauth the next generation, 2020. https://www.infoworld.com/article/3596345/gnap-oauth-the-next-generation.html, Accessed 2021-02-15.

[35] J. Richer, A. Perecki, and F. Imbault. Grant Negotiation and Authorization Protocol draft 03, 2021. https://datatracker.ietf.org/doc/draft-ietf-gnap-core-protocol/, Accessed 2021-02-01.

[36] T. Allen and D. Baier. Supported specifications identityserver4, 2021. https://identityserver4.readthedocs.io/en/latest/intro/specs.html.

[37] R. Davis. What is burp suite. https://www.pentestgeek.com/what-is-burpsuite.

[38] T. Zwierzchoń. JSON web signature (JWS) and JWS detached for a five-year-old., 2019. https://medium.com/swlh/json-web-signature-jws-and-jws-detached-for-a-five-year-old-88729b7b1a68.

[39] A. Sanso. Top 10 OAuth 2 implementation vulnerabilities. http://blog.intothesymmetry.com/2015/12/top-10-oauth-2-implementation.html.

[40] O.Khomiak. The most common OAuth 2.0 hacks. https://habr.com/en/post/449182/.

[41] D. Kapil. Attacking the OAuth protocol. https://dhavalkapil.com/blogs/Attacking-the-OAuth-Protocol/.

[42] PortSwigger. OAuth 2.0 authentication vulnerabilities. https://portswigger.net/web-security/oauth.

[43] C. Morgan. Attacking and defending OAuth 2.0 (part 1 of 2: Introduction, threats, and best practices), 2020. https://www.praetorian.com/blog/attacking-and-defending-oauth-2-0-part-1/.

[44] T. Krishnamohan. The Authorization Code Redirect URI Manipulation attack in OAuth2.0. https://www.thearmchaircritic.org/mansplainings/the-authorization-code-redirect-uri-manipulation-attack-in-oauth-2-0.

[45] Detectify. Owasp top 10 2013: Unvalidated redirects and forwards, 2016. https://blog.detectify.com/2016/08/15/owasp-top-10-unvalidated-redirects-and-forwards-10/.

[46] PortSwigger. Lab: Stealing oauth access tokens via an open redirect. https://portswigger.net/web-security/oauth/lab-oauth-stealing-oauth-access-tokens-via-an-open-redirect.

[47] KristenS. Cross site request forgery (CSRF), 2021. https://owasp.org/www-community/attacks/csrf.

[48] D. Syer. Cross site request forgery and OAuth2, 2011. https://spring.io/blog/2011/11/30/cross-site-request-forgery-and-oauth2, Accessed 2021-02-02.

[49] A.J. Paverd, A. Martin, and I. Brown. Modelling and automatically analysing privacy properties for honest-but-curious adversaries. *Univ. Oxford Tech. Rep*, 2014.

[50] PortSwigger. Oauth 2.0 authentication vulnerabilities. https://portswigger.net/web-security/oauth#flawed-scope-validation.
[51] OpenID. Financial-grade api (fapi) wg. https://openid.net/wg/fapi/.
[52] T. Kawasaki. Financial-grade api (fapi), explained by an implementer, 2019. https://darutk.medium.com/financial-grade-api-fapi-explained-by-an-implementer-d09fcf2ff932.
[53] J. Petters. What is saml and how does it work?, 2020. https://www.varonis.com/blog/what-is-saml/.