International Journal of Computer Applications (IJCA)
Volume 3, Issue 1, January-December 2022, pp. 33-49, Article ID: IJCA_03_01_005
Available online at https://iaeme.com/Home/issue/IJCA?Volume=3&Issue=1
Impact Factor (2022): 5.91 (Based on Google Scholar Citation)
ISSN Online: 2341-7801, Journal ID: 2969-1566

© **IAEME** Publication

OPEN ACCESS

UNVEILING VULNERABILITIES: A HOLISTIC ANALYSIS OF OAUTH PROTOCOL SECURITY IN CONTEMPORARY WEB ENVIRONMENTS

Suranjit Kosta
SAGE University, Indore, India

Suraj Disuja
SAGE University, Indore, India

ABSTRACT

The OAuth protocol has emerged as a vital component within contemporary web applications, facilitating the secure authentication and authorization of users for third-party services. Nonetheless, the widespread adoption of OAuth has heightened concerns regarding potential security vulnerabilities. This research endeavors to conduct a thorough security evaluation of the OAuth protocol as implemented in modern web applications, aiming to both identify and mitigate these risks. By scrutinizing the OAuth 2.0 specification in the context of contemporary systems-centric attacks, this study reveals vulnerabilities such as redirect_uri validation weaknesses that leave Identity Providers susceptible to redirect confusion and brute-force attacks on OAuth client_credentials grant types. Furthermore, it delves into the misuse of the scope parameter, demonstrating its potential for enabling unauthorized access. Through the presentation of end-to-end attack scenarios, which amalgamate various attack techniques with prevalent web application vulnerabilities, this research elucidates the possibility of complete compromise in the secure delegated access promised by OAuth 2.0. Moreover, the study includes the development of a laboratory environment tailored for a common OAuth scenario, intended to aid developers and security researchers in simulating exploitation and fostering a deeper understanding of the associated risks.

Keywords: Oauth 2.0, Redirect Uri, Host/Path Confusion, Bruteforce, Account Takeover

Cite this Article: Suranjit Kosta and Suraj Disuja, Unveiling Vulnerabilities: A Holistic Analysis of Oauth Protocol Security in Contemporary Web Environments, International Journal of Computer Applications (IJCA) 3(1), 2022, pp. 33-49.
https://iaeme.com/Home/issue/IJCA?Volume=3&Issue=1

1. INTRODUCTION

OAuth, short for Open Authorization, stands as a pivotal industry-standard protocol facilitating secure authorization and authentication among disparate systems. It empowers users to grant access to their resources without divulging their credentials directly. The protocol operates on the principle of access tokens, which are issued by an authorization server subsequent to user authentication and consent. These tokens serve as the means for clients to access protected resources on behalf of the user, ensuring a standardized and secure method of authorization delegation [1].

In the authorization code grant type [2], the process unfolds as follows: the client directs the user to the authorization server, where authentication and authorization occur. Upon successful authentication, the authorization server furnishes the client with an authorization code. This code is then exchanged for an access token by the client via a request to the authorization server, incorporating the authorization code, client credentials, and requisite parameters. Following validation, the authorization server issues an access token, enabling the client to interact with protected resources on the user's behalf.

However, vulnerabilities lurk within OAuth implementations, particularly concerning redirect_uri misconfigurations. These flaws are often exploited by malevolent entities to reroute users to unauthorized or malicious destinations, potentially leading to unauthorized resource access, phishing incidents, or redirection to harmful websites.

The scope parameter within OAuth requests delineates the extent of access granted to the client application. Illicit manipulation of scope parameters poses a significant threat, empowering attackers to escalate privileges and breach intended authorization levels. Such breaches could culminate in unauthorized data access, leakage, or other security breaches.

The Client Credentials Flow, as stipulated in OAuth 2.0 RFC 6749, section 4.4 [1], revolves around applications exchanging their credentials, like client ID and client secret, for access tokens. This flow is tailored for Machine-to-Machine (M2M) applications, such as CLIs, daemons, or backend services, wherein the system authenticates and authorizes the application, rather than a user.

Client credentials, encompassing the client ID and client secret, serve as the linchpin authentication mechanism for OAuth clients. The vulnerability to brute force attacks targeting these credentials underscores the imperative of robust authentication mechanisms, as such assaults could furnish attackers with unauthorized access to OAuth authorization servers and valid access tokens. Notably, weak or predictable client credentials are particularly susceptible to exploitation, emphasizing the necessity for fortified security measures [3, 4].

2. RELATED WORK

Feng Yang and Sathiamoorthy Manoharan in their paper “A security analysis of the OAuth protocol” [4] examined security threats in OAuth protocols, contrasting OAuth 1.0's focus on security with OAuth 2.0's trade-off for simplicity, which resulted in reduced security measures. The study conducted a systematic analysis of security threats in OAuth, identifying various network attacks like replay attacks, eavesdropping, CSRF attacks, and impersonation. The root causes of these vulnerabilities included the lack of TLS protection on callback endpoints, allowing multiple uses of authorization codes, absence of signature requirements for authorization requests, lack of vetting process for client applications, insufficient URI validation mechanism, and absence of authorization code authenticity verification.

The analysis concentrated on communication between user-agent and authorization server, as well as between user-agent and client application, with a future recommendation for further examination of access token transmission security between the client application and authorization server.

Wanpeng Li and Chris J. Mitchell, in 2014,[5] assessed the security of 60 OAuth 2.0 implementations used for federation-based Single Sign-On (SSO) on major Chinese websites. It revealed that nearly half of these implementations were vulnerable to Cross-Site Request Forgery (CSRF) attacks during the federation process, posing a significant risk to user accounts. These attacks enabled malicious parties to link their Identity Provider (IdP)-managed accounts to users' IdP-managed accounts without needing their account credentials. Additionally, due to the absence of a standardized federation process, the study uncovered logic flaws in real-world implementations, further facilitating the binding of an attacker's IdP-managed account to a user's RP (Relying Party)-managed account. The researchers notified affected RPs and IdPs of the identified vulnerabilities and provided potential mitigation strategies. The study aims to raise awareness among IdPs and RPs about the vulnerabilities to CSRF attacks in the OAuth 2.0 identity federation process. Ultimately, the researchers advocate for the standardization of a robust federation process for OAuth 2.0 to mitigate future risks effectively.

Eugene Ferry John O Raw Kevin Curran achieved a comprehensive evaluation of a developed solution based on OAuth 2.0[6], focusing on its performance and security aspects. They successfully implemented the solution, which demonstrated a high level of security and conformity to the OAuth 2.0 specification, albeit with some limitations in meeting the OpenID Connect specification requirements. The development process was facilitated by leveraging the DotNetOpenAuth (DNOA) library, which simplified the creation of client applications and the authorisation server, although they encountered challenges due to poor documentation. Despite constraints on time and resources, the authors managed to complete the solution, primarily implementing the mainstream server-side flow grant. They identified potential areas for future expansion, such as contributing to the DNOA open-source project and incorporating additional OAuth grant flows. Additionally, they recognized the importance of aligning OAuth implementations across vendors for better interoperability. The writers also discussed considerations for developers when choosing OAuth support libraries, highlighting the balance between vendor-specific and generic libraries. They foresaw potential advancements in OAuth integration on mobile devices, such as the development of native OAuth consent dialogs. Furthermore, the authors examined the broader landscape of OAuth adoption, anticipating changes in authentication standards and the potential incorporation of two-step authentication into OAuth consent dialogs. They emphasized the ongoing evolution of OAuth and the need for continued efforts to improve validation and conformity among vendors, ultimately aiming for OAuth to become the definitive authentication standard on the web.

Fett, Daniel, Ralf Küsters, and Guido Schmitz conducted the first extensive formal analysis of OAuth 2.0[7], utilizing a comprehensive and expressive web model. Their analysis covered all modes (grant types) of OAuth, including potential vulnerabilities from malicious Relying Parties (RPs), Identity Providers (IdPs), and compromised browsers/users. The underlying generic web model employed in their analysis is the most comprehensive to date. Their thorough examination revealed four attacks on OAuth, including those affecting OpenID Connect, which builds upon OAuth. The following attack on the OAuth flow were discussed – 1. 307 Redirect Attack, 2. IdP Mix-Up Attack, 3. State Leak Attack, 4. Naïve RP Session Integrity Attack. Their thorough examination revealed four attacks on OAuth, including those affecting OpenID Connect, which builds upon OAuth. The authors not only verified these attacks but also proposed fixes and reported them to the working groups for OAuth and OpenID Connect.

Subsequently, the working groups acknowledged the attacks, and fixes and recommendations are currently under discussion or already incorporated into a draft for a new RFC.

3. INTRODUCTION TO OAUTH

3.1. OAuth Overview

OAuth (Open Authorization) is a widely adopted protocol for enabling secure authorization and authentication between different systems, particularly in the context of web and mobile applications. It allows users to grant third-party applications limited access to their resources without sharing their credentials directly. OAuth operates through a set of components that work together to facilitate secure authorization and access delegation:

Resource Owner: The resource owner is the user who owns the protected resources, such as data or services, that the client application seeks to access. The resource owner has the authority to grant or deny access to their resources.

Client: The client is the third-party application seeking access to the user's resources. It could be a web or mobile application, a desktop application, or a service running on a server. The client interacts with the authorization server to obtain access tokens and subsequently access protected resources.

Authorization Server: The authorization server is responsible for authenticating the resource owner and issuing access tokens to clients after successful authentication and authorization. It verifies the identity of the resource owner and ensures that the client is authorized to access the requested resources.

Resource Server: The resource server hosts the protected resources that the client seeks to access. It is responsible for handling requests for these resources and determining whether to grant or deny access based on the validity of the access token presented by the client.

Access Token: An access token is a credential issued by the authorization server that represents the authorization granted to the client to access specific resources on behalf of the resource owner. Access tokens are typically short-lived and scoped to specific resources and actions.

Authorization Grant: An authorization grant is a credential representing the resource owner's consent for the client to access their resources. It is obtained by the client from the authorization server through various grant types, such as the authorization code grant, implicit grant, resource owner password credentials grant, or client credentials grant.

Redirect URI: A redirect URI is a callback URL provided by the client to the authorization server, where the resource owner is redirected after authenticating and authorizing the client. The redirect URI is used to return authorization codes or access tokens to the client securely.

Client ID: The client ID serves as a unique identifier assigned to a client application by the authorization server in OAuth. It is publicly known and is used to associate incoming requests with the registered client.

Client Secret: The client secret is a confidential credential known only to the client application and the authorization server. It is used to authenticate the client and verify its identity during OAuth exchanges

3.2. Comparison of OAuth with other protocols

OAuth is often compared with other authentication protocols, such as OpenID Connect (OIDC) and Security Assertion Markup Language (SAML), each serving different purposes in the realm of identity and access management. Here's a comparison between OAuth and these protocols:

3.2.1 OIDC: Built on top of OAuth, OIDC adds authentication capabilities, allowing applications to verify the identity of users using JSON Web Tokens (JWTs). It's widely used for single sign-on (SSO) and identity federation across multiple applications.

Used for authentication and single sign-on (SSO) across multiple applications within a single domain or across different domains. It provides a standardized identity layer on top of OAuth for verifying user identities.

3.2.2 SAML: SAML is an older protocol primarily focused on single sign-on and identity federation. It allows for the exchange of authentication and authorization data between identity providers and service providers using XML-based assertions. Based on XML and HTTP, SAML relies on XML-based assertions and protocols for exchanging authentication and authorization data between identity providers and service providers.

3.2.3 LDAP: Designed for accessing and managing directory information, such as user accounts, groups, and other organizational data. LDAP is commonly used for centralized authentication and directory services within enterprise networks.

Operates on a hierarchical data model, typically using the Directory Information Tree (DIT) to organize directory entries. Data is represented in a structured format, often using the Lightweight Data Interchange Format (LDIF) or similar standards.

3.3. OAuth Authorization grant type

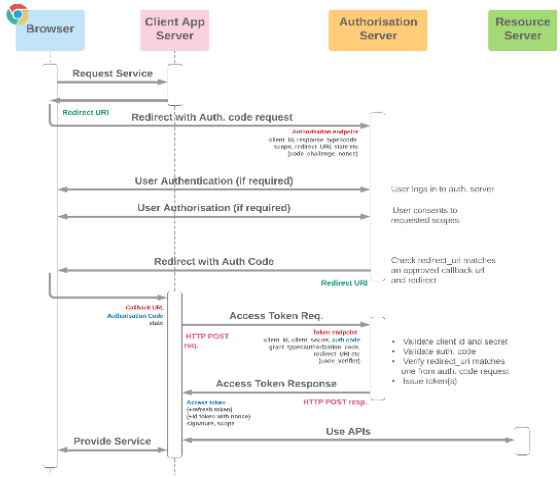


Figure 1. OAuth authorization_code flow

OAuth 2.0 Authorization Code Flow is a widely used and highly secure method for obtaining access tokens to authenticate and authorize user access to protected resources on the web. This flow is commonly employed in scenarios where the client application needs to access resources on behalf of the user, without exposing the user's credentials to the client.

The flow typically involves several steps:

1. **Authorization Request**: The client initiates the flow by redirecting the user to the authorization server, requesting authorization to access protected resources. This request includes parameters such as the client ID, requested scope (permissions), redirect URI, and optionally, a state parameter for preventing CSRF attacks.
2. **User Authentication and Authorization**: The user is prompted to authenticate themselves with the authorization server. Upon successful authentication, the user is presented with a consent screen where they grant or deny the requested permissions to the client application.
3. **Authorization Grant**: If the user grants permission, the authorization server generates an authorization code and redirects the user back to the client application's redirect URI, along with the authorization code.
4. **Token Request**: The client application exchanges the received authorization code for an access token by making a back-end call to the authorization server's token endpoint. Along with the authorization code, the client includes its client ID, client secret (if applicable), redirect URI, and grant type (authorization code).
5. **Access Token Response**: Upon successful validation of the authorization code, the authorization server issues an access token to the client application. Additionally, it may also issue a refresh token, which can be used to obtain a new access token when the current one expires.
6. **Accessing Protected Resources**: With the access token, the client application can now make API requests to the resource server on behalf of the user. The resource server validates the access token and grants access to the requested resources if the token is valid.

3.4. OAuth Client Credentials grant type

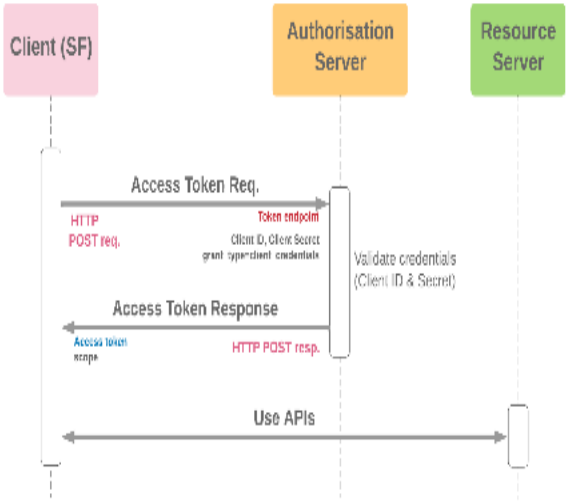


Figure 2. OAuth client_credentials flow

OAuth 2.0 Client Credentials Flow is a simplified authorization flow primarily used when the client application itself is the resource owner, meaning it accesses its own resources rather than on behalf of a user. This flow is typically employed in server-to-server communication or when the client application requires access to its own resources without user involvement.

The Client Credentials Flow follows a straightforward process:

1. **Client Authentication**: The client application authenticates itself directly with the authorization server, typically using its client ID and client secret. These credentials are securely stored within the client application and are used to prove its identity to the authorization server.
2. **Token Request**: Once authenticated, the client application sends a token request to the authorization server, typically via a secure back-end call to the token endpoint. This request includes the client credentials, the requested scope (permissions), and the grant type, which in this case is "client_credentials."
3. **Access Token Response**: Upon successful validation of the client credentials, the authorization server issues an access token directly to the client application. The access token represents the client's authorization to access the requested resources. Additionally, the token may include an expiration time to enforce security and limit exposure.
4. **Accessing Protected Resources**: With the obtained access token, the client application can make authorized requests directly to the resource server, passing the access token in the request headers. The resource server validates the access token and grants access to the requested resources if the token is valid and authorized for the requested scope.

Unlike other OAuth flows, the Client Credentials Flow does not involve user authentication or consent, making it suitable for scenarios where user involvement is unnecessary or impractical. However, it's important to note that since this flow relies solely on client credentials, it should only be used when the client application itself is trusted and authorized to access the requested resources.

4. OAUTH SECURITY ANALYSIS

4.1. Unvalidated redirect_uri parameters

The unvalidated redirect URI parameter vulnerability in OAuth is a security issue that arises when an OAuth authorization server accepts and redirects users to arbitrary redirect URIs without proper validation. This vulnerability can be exploited by attackers to redirect users to malicious websites, leading to various forms of attacks such as phishing, session hijacking, and credential theft.When a user initiates the OAuth authorization flow, they are redirected to the authorization server's login page, where they authenticate themselves and authorize the client application.

During the authorization process, the client application includes a redirect URI parameter in the authorization request. This parameter specifies the URI to which the authorization server redirects the user after successful authentication and authorization.Insecure OAuth implementations may fail to adequately validate the redirect URI parameter supplied by the client application. Instead, they blindly accept and redirect users to the URI specified, regardless of its authenticity

For example, consider the following HTTP request, which initiates the OAuth flow –

GET
/authorize?response_type=code&client_id=client_id&redirect_uri=**http://example.com/callback**&scope=read_profile HTTP/1.1

Host: authorization_server.com

In the above request, example.com is the service provider, and in a regular flow, when the user makes the above request, the following response is sent –

```
HTTP/1.1302 Found
Location: http://example.com/callback?code=AUTHORIZATION_CODE
```

However, if the redirect_uri is not being validated, an attacker can force the victim to visit the following URL, and redirect to malicious websites, with the authorization code –

```
GET
/authorize?response_type=code&client_id=client_id&redirect_uri=http://malicious.com/callback&scope=read_profile HTTP/1.1
Host: authorization_server.com
```

Response –

```
HTTP/1.1302 Found
Location: http://malicious.com/callback?code=AUTHORIZATION_CODE
```

There are instances where this types of misconfigurations have been mitigated, however, sometimes it is still possible to bypass these protections using special url characters. Consider the following cases –

1. The authorization server allow only **subdomains** for the redirect_uri**.** In this case, the following redirect_uri could be used, in order to redirect the user to malicious.com host -

https://malicious.com?www.example.com

2. In the second case, the authorization server allow only the intended host, but allow arbitrary path to the redirect_uri url parameter. For instance – https://www.example.com/*

Here, if an open redirection exists on the www.example.com host, it might be possible to exfiltrate the OAuth code to malicious domain, similar to this –

```
https://www.example.com/openredirect?redirect=https://malicious.com

--302—

https://malicious.com?code=OAuthcode
```

4.2. Weak client_credentials

In OAuth 2.0, the Client Credentials Flow is commonly used by confidential clients (applications that can securely store and authenticate with a client secret) to obtain access tokens directly from the authorization server. However, the security of this flow relies heavily on the confidentiality of the client credentials, including the client ID and client secret. If these credentials are leaked or guessed, they can be exploited through brute force attacks to obtain access tokens unauthorizedly.

Below are sample HTTP requests demonstrating how an attacker could attempt to brute force the client secret in the Client Credentials Flow to obtain an access token:

Brute Force Attack:

```
HTTP Request

POST /token HTTP/1.1
Host: authorization_server.com
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&client_id=attacker_client_id&c
   lient_secret=guess1

HTTP Response
POST /token HTTP/1.1
Host: authorization_server.com
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&client_id=attacker_client_id&c
   lient_secret=guess2
```

The attacker sends multiple token requests to the authorization server's token endpoint, each with a different guessed value for the client secret. The attacker systematically tries different secret values (e.g., "guess1", "guess2", etc.) in an attempt to find the correct one through brute force.

Successful Token Request (If the Brute Force Succeeds):

```
HTTP Request
POST /token HTTP/1.1
Host: authorization_server.com
Content-Type: application/x-www-form-

grant_type=client_credentials&client_id=attacker_client_id&clie
   nt_secret=correct_secret
```

If the attacker successfully guesses the correct client secret, they can obtain an access token from the authorization server. The access token can then be used to make unauthorized requests to access protected resources.

It's important to note that in a real-world scenario, the attacker would automate the process of sending token requests with various secret values, typically using specialized software or scripts. Additionally, the authorization server may implement rate limiting or other security measures to detect and prevent brute force attacks.

4.3. Unvalidated scope parameters.

The "scope" parameter in OAuth 2.0 is used to specify the level of access that the client application requests when it requests authorization from the resource owner (typically the user). This parameter defines the permissions or scopes that the client application is requesting access to on behalf of the user. While the scope parameter is essential for ensuring that the client application only receives the necessary access rights, it can potentially be exploited by attackers to gain higher privileges than intended.

An attacker may attempt to manipulate the scope parameter to request access to additional resources or perform actions that they are not authorized to perform. For example, if a client application is originally granted access to read a user's profile information (e.g., "profile" scope), the attacker could modify the scope parameter to include additional permissions such as write access to the user's profile (e.g., "profile write"). This would grant the attacker higher privileges than initially intended.

Below are sample HTTP requests and responses demonstrating how an attacker could request elevated scopes:

Original Authorization Request: The client application initiates the authorization process by sending a request to the authorization server, specifying the desired scope.

```
HTTP Request
GET
/authorize?response_type=code&client_id=client_id&redirect_uri=http://examp
le.com/callback&scope=profile HTTP/1.1
Host: authorization_server.com
```

In this request, the client application requests access to the user's profile information (`scope=profile`).

A**ttacker's Manipulated Authorization Request**: The attacker intercepts the original authorization request and manipulates the scope parameter to include additional, elevated scopes.

```
HTTP Request
GET
/authorize?response_type=code&client_id=attacker_client_id&redirect_uri=htt
p://attacker.com/callback&scope=profile%20write%20admin HTTP/1.1
Host: authorization_server.com
```

Here, the attacker adds additional scopes (`write` and `admin`) to the original request, attempting to gain elevated privileges such as the ability to write to the user's profile and access administrative functionalities.

Authorization Server Response: The authorization server processes the authorization request and prompts the user to grant or deny the requested scopes.

```
HTTP Response
HTTP/1.1 302 Found
Location: http://example.com/callback?code=AUTHORIZATION_CODE
```

The authorization server redirects the user to the specified redirect URI (`http://example.com/callback`) along with an authorization code.

Token Request with Authorization Code: The client application exchanges the received authorization code for an access token, including the elevated scopes requested by the attacker.

HTTP Request

```
POST /token HTTP/1.1
Host: authorization_server.com
Content-Type:application/x-www-form-urlencoded

grant_type=authorization_code&code=AUTHORIZATION_CODE&redirect_uri=http://e
xample.com/callback&client_id=client_id&client_secret=client_secret&scope=p
rofile%20write%20admin
```

In this request, the attacker includes the elevated scopes (`profile write admin`) along with the authorization code to obtain an access token with higher privileges.

By manipulating the scope parameter in the authorization request, attackers attempt to gain elevated privileges beyond what the client application is originally authorized for. This highlights the importance of proper validation and authorization checks on the authorization server's side to prevent unauthorized scope elevation.

5. DEMONSTRATION LABS

In this section, sample laboratory environment is presented, designed to demonstrate vulnerabilities related to OAuth, focusing on redirect_uri misconfigurations, client credentials brute force attacks, and unauthorized scope privilege escalation. The lab aims to provide hands-on experience with identifying, exploiting, and mitigating these vulnerabilities in OAuth implementations within modern web applications.

The lab covers the following scenarios-

I. Redirect_URI Misconfigurations:

- The lab scenario simulates a misconfiguration in the redirect_uri parameter, a common vulnerability in OAuth implementations that can lead to unauthorized access or phishing attacks.
- Participants are provided with a vulnerable web application that includes OAuth integration for authentication and authorization.
- Through a series of exercises, participants attempt to exploit the misconfigured redirect_uri parameter to redirect users to malicious websites, steal authorization codes, or bypass authentication controls.
- Mitigation strategies, such as strict validation of redirect URIs and the use of state parameters, are discussed to prevent and mitigate such vulnerabilities.

The labs consists of various levels with respect to the mitigations-

Level 1 – The redirect_uri allows the following regular expression[8] -

^http:\/\/.*$

Due to usage of weak, regex, it is possible to allow any URL in the redirect_uri parameter, leading to OAuth code leakage to arbitrary hosts-

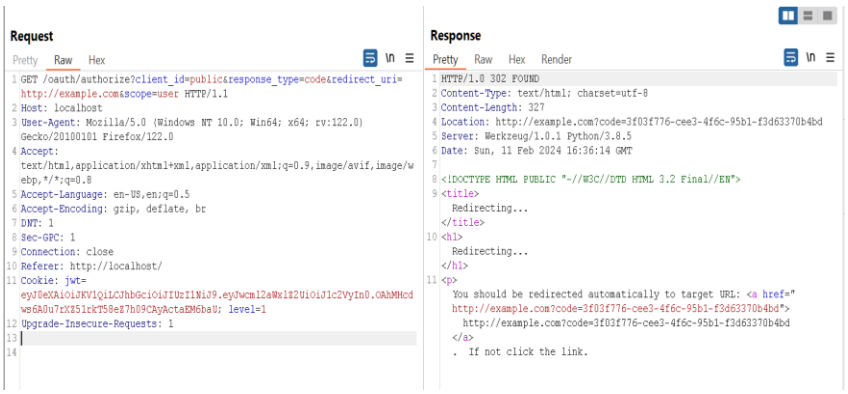


Figure 2 Intended redirect_uri

Figure 3 Modfied redirect_uri parameter

Level 2 – The redirect_uri allows the following regular expression-

^http:\/\/.*\.example\.com$

In this level ,the server only allow subdomains for “example.com” domain. This restriction can however be bypassed by using URL characters such as “?”,”#” or “/”, to break the domain from example.com to attacker.com, as seem in the screenshot –

Figure 4.HTTP request with Modified redirect_uri parameter

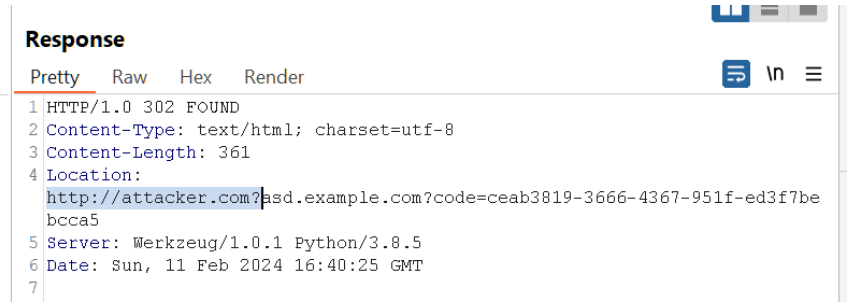


Figure 5. Response for the request.

The user woud be redirected to https://attacker.com instead of the intended example.com subdomain, along with the OAuth code.

Level 3 – The redirect_uri allows the following regular expression -

http:\/\/example.com.*$

In this case, the redirect_uri only allows host example.com and anything ending with it. This means that the host must start with example.com. This however, could be bypassed by adding ". attacker.com" at the end, and making this host a subdomain. See the screenshots below –

Figure 6 HTTP request, with modified redirect_uri

Level 4 – The redirect_uri allows the following regular expression - http:\/\/example.com:80.*$

In this level ,the server restricts host and port for the redirect_uri URL. This restriction can be bypassed by using URL characters such as "@" to break the domain from example.com to attacker.com, as seem in the screenshot –

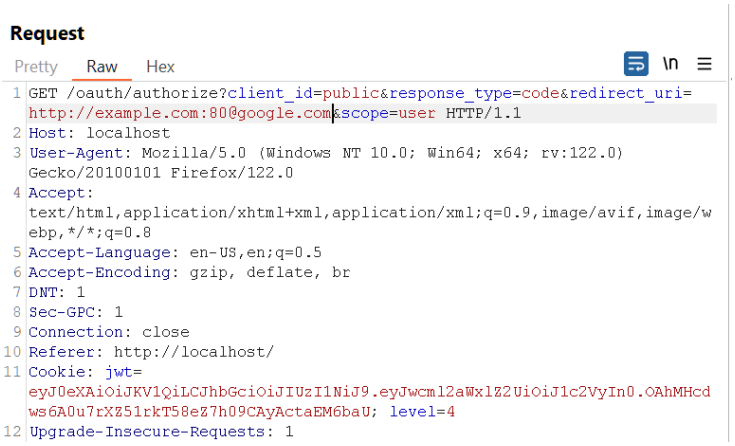


Figure 7. HTTP request, with modified redirect_uri

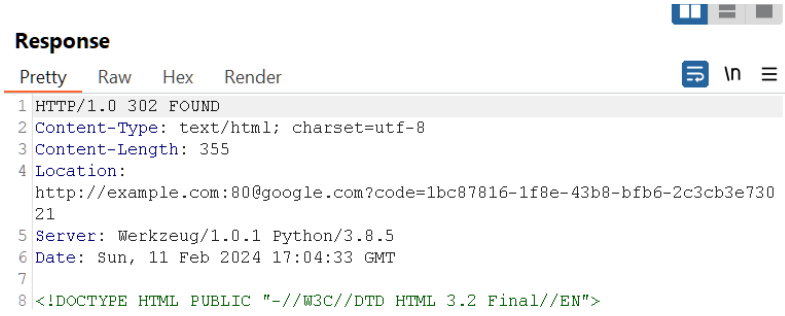


Figure 8. Response for the request.

II. Unauthorized Scope Privilege Escalation

- In this lab scenario, participants explore the risk of privilege escalation through unauthorized scope manipulation in OAuth requests.
- Participants interact with a vulnerable web application that grants access tokens with overly permissive scopes without proper validation.
- Through exercises, participants attempt to manipulate the scope parameter in OAuth requests to escalate their privileges and gain access to resources beyond their intended authorization level.
- Mitigation techniques, such as implementing fine-grained access control and validating scopes against predefined policies, are presented to mitigate the risk of unauthorized scope privilege escalation.

In the lab, the following attack can be demonstrated–

1. The attacker switches the scope from “user” to “admin”

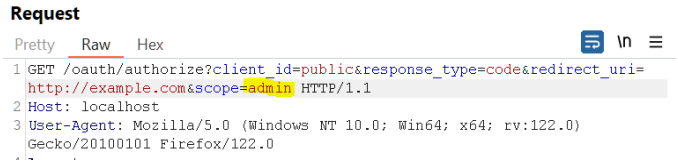
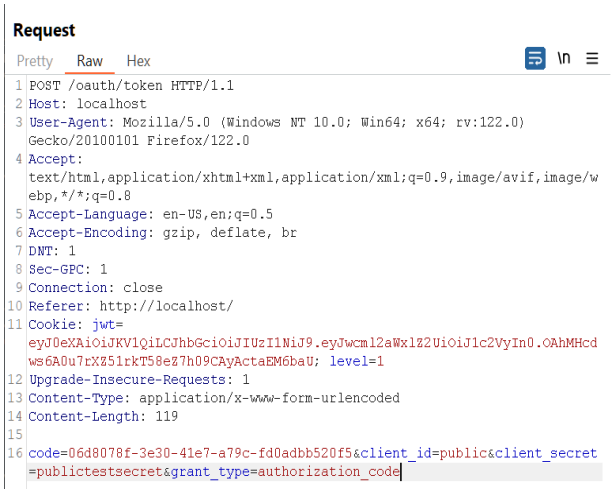


Figure 9. Modified scope parameter

2. The attacker exchanges code with token, using the /token endpoint

Request -



Response –

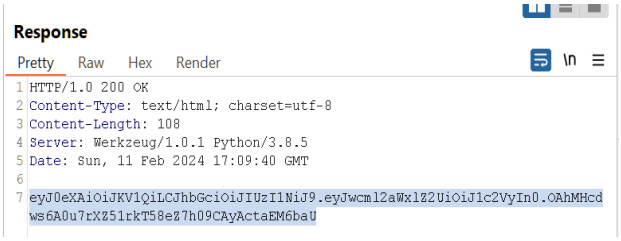


Figure 10. HTTP Response, with access token

3. The attacker can now access the admin panel, using the exchanged token –

Figure 11. Attacker accesses admin panel, without authorization

III. Client Credentials Brute Force Attacks:

- This lab scenario explores the risk of brute force attacks targeting client credentials, such as the client ID and client secret, used in OAuth client authentication.
- Participants are given a vulnerable OAuth client application with weak or predictable credentials.
- Through practical exercises, participants attempt to brute force the client credentials to gain unauthorized access to the OAuth authorization server or obtain valid access tokens.
- Countermeasures, including enforcing strong client credential security practices (e.g., using long and randomly generated secrets, implementing rate limiting), are discussed to mitigate the risk of brute force attacks.

To demonstrate, client_id and client_secret is brute forced on the /token endpoint. Burp suite's[9] inbuilt tool called intruder[10] is used for it. After some time, client_id and client_secret is correctly guessed, as it can be seen in the following screenshot –

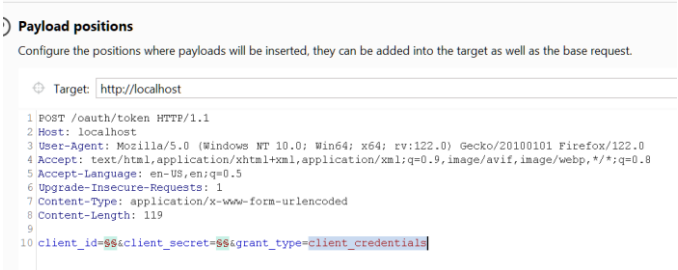


Figure 12. Attack configuration

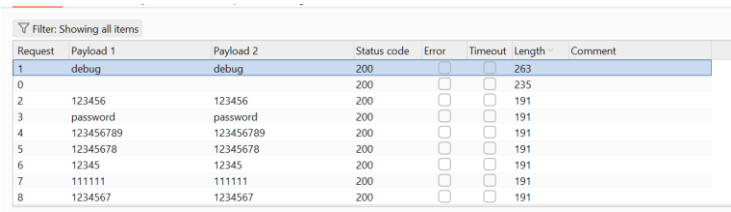
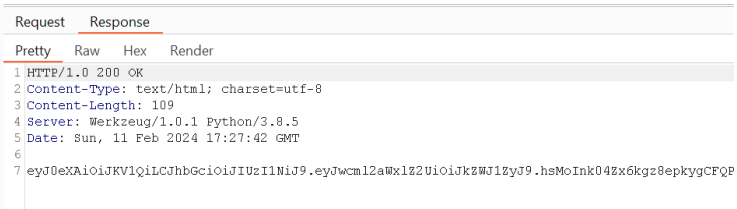


Figure 13. debug client_id and client_secret bruteforced for token

Access token, with admin privileges returned in the response –



7. CONCLUSION

In conclusion, our research project has provided a comprehensive analysis of the OAuth protocol's security aspects in modern web applications. Through meticulous examination of OAuth 2.0 specifications and real-world attack scenarios, key vulnerabilities such as redirect_uri misconfigurations, weak client credentials, and unauthorized scope privilege escalation were identified. Practical demonstration labs have further underscored the importance of addressing these vulnerabilities to safeguard user data and ensure the integrity of OAuth-based authentication and authorization processes.

While OAuth remains a cornerstone of secure authorization and authentication between different systems, our findings highlight the critical need for robust implementation practices and ongoing vigilance in the face of evolving cyber threats. Mitigation strategies such as strict validation of redirect URIs, enforcement of strong client credential security measures, and fine-grained access control for scope management are essential to bolster OAuth security.

Looking ahead, future research efforts should focus on exploring advanced threat models, evaluating emerging authentication mechanisms, and developing standardized best practices for OAuth implementation across diverse web application environments. By addressing these challenges collaboratively, we can enhance the resilience of OAuth-based security frameworks and ensure continued trust in digital identity management systems.

REFERENCES

[1] IETF. (2003). RFC 674: HTTP/1.1, Semantics and Content. [Online].

[2] OAuth 2.0. (n.d.). Authorization Code Grant Type. [Online].

[3] OAuth 2.0. (n.d.). Client Credentials Grant Type. [Online]

[4] Yang, F., & Manoharan, S. A security analysis of the OAuth protocol. In 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM) (2013, August) (pp. 271-276). IEEE.

[5] Li, W., & Mitchell, C. J. Security issues in OAuth 2.0 SSO implementations. In International Conference on Information Security (2014, October) (pp. 529-541) Cham: Springer International Publishing

[6] Chen, E. Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., & Tague, P. OAuth demystified for mobile application developers. In Proceedings of the 2014 ACM SIGSAC conference on computer and communications security (2014, November) (pp. 892-903).

[7] Fett, D., Küsters, R., & Schmitz, G. (2016). A comprehensive formal security analysis of OAuth 2.0. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.

[8] Mozilla. (n.d.). Regular expressions. [Online]. Available

[9] PortSwigger. (n.d.). Burp Suite. [Online]. Available