Assessing the security status of systems using the OAuth 2.0 authorization protocol

D. KANIA
dominik.kania100@gmail.com

Netcompany Poland
Puławska St. 180, 02-670 Warsaw, Poland

P. Górny
piotr.gorny@wat.edu.pl

Military University of Technology, Faculty of Cybernetics
Kaliskiego St. 2, 00-908 Warsaw, Poland

The article presents an analysis of the security of systems using the OAuth 2.0 protocol. It is an authorization protocol widely used in websites, including the largest in the world. It is characterized by high complexity and complicated operation. However, when used correctly, it provides significant convenience for users in sharing data with web applications from other websites. This describes specialized tests that were conducted to identify weaknesses in the basic OAuth 2.0 implementation. Based on these tests, the threats and risks associated with inaccurate implementation were identified, along with a presentation of their potential effects.

Keywords: authorization, protocol, vulnerability.

DOI: 10.5604/01.3001.0055.0853

1. Introduction

In today's technology-enabled environment, which is characterized by a growing number of web and mobile applications and an evolving microservices infrastructure, ensuring security and managing access to resources are becoming key challenges for software engineers. The OAuth 2.0 protocol, a major authorization standard, has gained popularity due to its flexibility and ability to securely manage access through tokens.

OAuth 2.0 is widely used by major technology companies, including Google, Facebook and Microsoft, as well as by developers creating applications in various industries. Despite its many advantages, the OAuth 2.0 protocol is not free of security challenges. Implementing this protocol requires a thorough understanding of its mechanisms and an awareness of the potential risks that can arise from improper implementation.

The purpose of the article is to identify security threats to implementations of the OAuth 2.0 protocol using an original test environment for the most common vulnerabilities encountered in its implementations.

Additional motivation to take up this topic was the discovery of numerous vulnerabilities and attacks that had been carried out on systems using this protocol. Of particular interest was the work of Egor Homakov [1], in which the author, using a combination of several minor vulnerabilities, created a powerful attack vector on the GitHub system enabling the free and undetectable theft of user data. It is easy to find examples of attacks on other large sites such as Facebook [2], [3], BitBucket [4], again Github [5] or on demo sites [6].

Attack scenarios against various components of the authorization system are described, such as an unprotected *redirect_uri* parameter, incorrect validation of the access range and failure to verify the *state* parameter. Each experiment includes a detailed attack scheme, the course of the experiment, the countermeasures used, and the effects observed after the experiment.

The summary discusses the advantages and disadvantages of the analyzed implementation of the OAuth 2.0 protocol, as well as provides recommendations on best practices for securing applications using this protocol. In addition, the possibilities and limitations of using out-of-the--box open-source solutions to support the secure implementation of OAuth 2.0 are indicated.

2. OAuth 2.0

2.1. Context

With the advent of Web 2.0 and the development of social media, several websites have emerged that allow users to upload content, such as photos. For example, a user may have wanted a photo printing service (e.g., www.photos.com) to access his photos from a social media site. Without a proper solution, the user would have had to share their social media login information. If the site's data were leaked, the user's social media account would be compromised, and the user would have no control over what the site could do with his data.

The solution to this problem turned out to be the OAuth 2.0 protocol, which allowed users to authorize applications to access their data on another site without having to reveal their login credentials. It allows applications to access a user's resources with a special credential called a token, which allows it to call the resource server's API [7].

There is some inconsistency in the first sentence displayed on the protocol's website which says that *OAuth 2.0 is an industry-standard protocol for authorization* [8]. On the other hand, the official *Request For Comments* (RFC) document defines OAuth 2.0 as a *framework* for authorization [9].

The difference is significant because protocols require strict adherence to rules and step-by-step action according to specifications. Framework, on the other hand, allows a loose approach to implementation. It leaves room for individual interpretation by the user, which can sometimes be disastrous. Proper implementation requires a full understanding of the OAuth 2.0 schema, which can be difficult, as the protocol itself is quite complex, and an inexperienced developer can easily make critical mistakes.

1.2. Access token

An *access token* is a credential that allows access to a secured resource. In *token-based authentication*, the holder of such a token has access to the data set by the authorization server when it is issued. As a rule, it takes the form of an encoded string of characters to facilitate transport between applications.

1.3. Refresh token

It is a good practice that the access tokens generated should have restrictions imposed on them. The authorization server should generate tokens authorizing only a limited range of data only for the time needed to use them. For this reason, tokens must have a lifetime, after which they become inactive and unusable. Then, in this situation, the client must restart the entire process of acquiring the access token redundantly repeating the same steps. OAuth 2 provides a solution for this.

A *refresh token* is a token used to obtain access tokens from an authorization server. In form, it is identical to an access token but does not allow access to resources from the server. It is used in requests to the authorization server to retrieve a new token in case the previous access token has expired. In this way, once the client authorizes once, it does not have to repeat it. At the same time the authorization server has more control over the number of active tokens. A diagram of the token refresh process is shown in Figure 1.

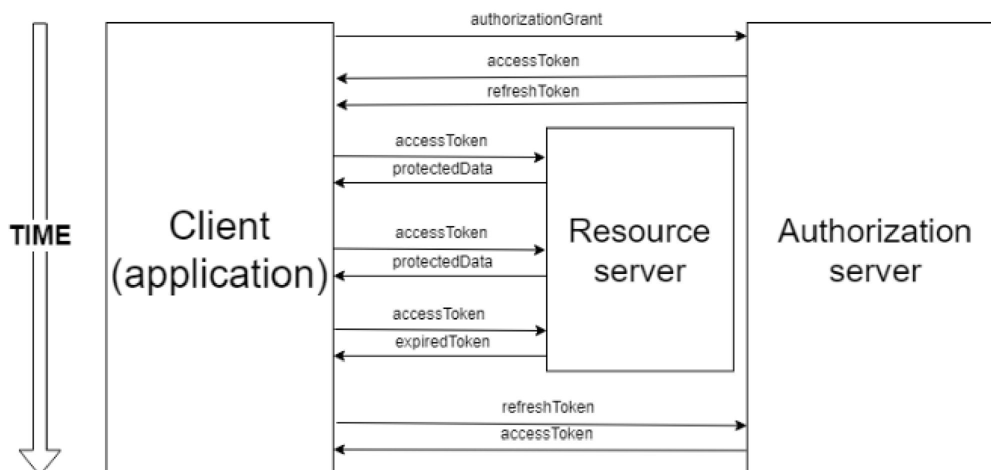

Fig. 1. Flowchart of refreshing the access token
Source: Own study based on [9]

3. General scheme of the protocol

A simplified diagram of how the OAuth 2.0 protocol works is shown in Figure 2.

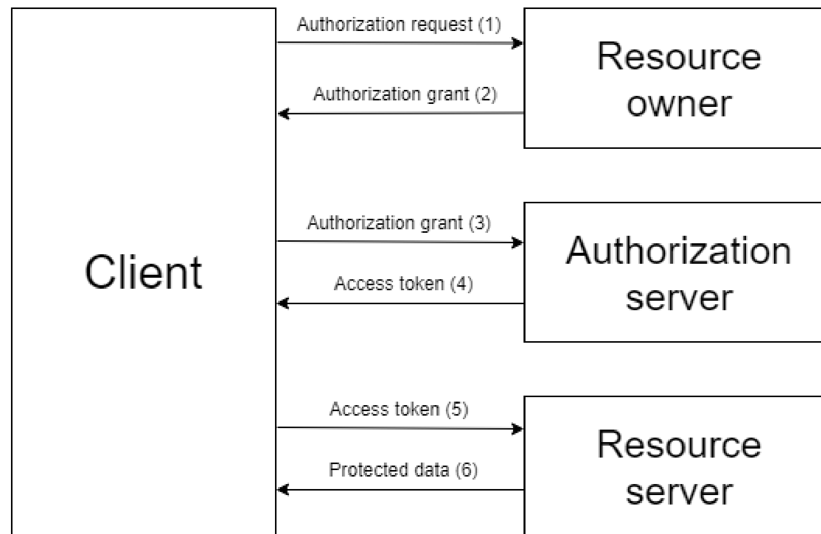

Fig. 2. Activity diagram of OAuth 2.0
Source: Own study based on [9]

4. Risks of Using OAuth 2.0

4.1. Unsecured redirect uri parameter

The OAuth 2.0 protocol uses a **redirect_uri** mechanism as part of the authorization process. The user is redirected to the service provider's authorization page and then, after successful authorization, is redirected back to the client application.

The vulnerability related to the lack of *redirect_uri* validation is that if the authorization server does not validate the *redirect_uri* parameter, the attacker can provide his own *redirect_uri* value. This allows the user to be redirected to a page controlled by the attacker after successful authorization.

In such a scenario, an attacker can intercept the authorization code that is passed through the URL parameter on the redirect page. This code can then be used to obtain a token to access user resources.

To prevent this type of attack, the authorization server should strictly validate the *redirect_uri* parameter. The validation should check that the redirect is allowed for the client and that the redirect URL is registered and matches the URL that was set during the OAuth client configuration [10].

4.2. Open Redirection Vulnerability

Today's websites often use HTTP or URL parameters that automatically direct the user to a specific URL without any user interaction. *Open redirection* is a vulnerability in which an attacker can manipulate the value of a redirection parameter, leading to users being unintentionally redirected away from the destination pages of [11].

4.3. Attack scenario in OAuth 2.0

In the context of the OAuth 2.0 protocol, the parameter that redirects the user to other services is **redirect_uri.** If this parameter is unprotected and not verified on the authorization server side, then the tokens or Authorization Codes generated by this server can be sent anywhere, not necessarily back to the client application.

The scenario of an example attack on an unsecured OAuth 2.0 parameter presents itself as follows:

1. Attacker Mallory constructs a malicious HTTP request to the authorization server containing a reworked **redirect_uri** parameter. The attacker sets his own domain there, which he specifically prepared to take over the Authorization Code. Examples of the original and rewritten request are shown in codes 1 and 2:

```
1   https://client.original:8080/authorize
2   ?response_type= authorization_code
3   &redirect_uri=
4    https://client.original:8080/callback
5   ...
```

Code 1. Example of correct authorization request
Source: Own study

```
1   https://client.original:8080/authorize
2   ?response_type= authorization_code
3   &redirect_uri=
4    https://client.attacker:8086/callback
5   ...
```

Code 2. Example of modified malicious authorization request
Source: Own study

2. The attacker sends the Alice victim a malicious link to this request and encourages her to click on it. It can be done this in the form of a link in an image, through an email encouraging a quick login to the site, or through another phishing method.
3. Alice clicks on a malicious link that redirects her to the real authorization server. There, Alice logs in and grants authorization convinced that everything is in order.
4. The authorization server generates an Authorization Code (or Access Token if the implicit mode is used) and sends it to the malicious address specified in **redirect_uri**.
5. The malicious attacker's client accepts the request with the received code and saves it. In doing so, it receives Alice's credentials.
6. The attacker can additionally perform a redirect to the original client application (e.g. to the login page) to hide his actions from the victim to simulate an error on the side of the authorization server.

5. Incorrect validation of access range

In each OAuth flow, the user must approve the requested access based on the scope defined in the authorization request. The resulting token allows the client application to access only the scope approved by the user. However, in some cases, an attacker can "enhance" the access token (stolen or obtained through a malicious client application) with additional permissions due to faulty validation by the OAuth service. The process depends on the type of grant.

5.1. Verification with Authorization Code

In the case of the authorization code grant type, user data is requested and sent via secure server-to-server communication, which an outside attacker is usually unable to directly manipulate. However, it may still be possible to achieve the same result by registering your own client application with the OAuth service.

Let's assume that the attacker's malicious client application initially requested access to the user's email address using the **read_email** scope. After the user approves this request, the malicious client application receives an authorization code. Since the attacker controls its client application, it can add another scope parameter to the code */token* exchange request containing an additional **read_contacts**, profile scope that allows reading contacts. An example of a modified malicious request is shown in code 3:

```
1   POST /oauth/token HTTP/1.1
2   Host: authorization-server.com
3
4   code=Yzk5ZDczMzRlNDEwY
5   &grant_type=code
6   &redirect_uri=https://exam.com/cb
7   &client_id=id
8   &client_secret=supersecret
9   &code_verifier=Th7UHJdLswIYQx...
10  &scope=read_email,read_contacts
```

Code 3. HTTP request with modified scope
Source: Own study based on [12]

If the server does not verify this range with the range from the initial authorization request, it will generate an access token using the new range and send it to the attacker's client application.

```
1  {
2   "access_token": "AYjcyMz…",
3   "refresh_token": "RjY2NjM…",
4   "token_type": "Bearer",
5   "expires": 3600,
6   "scope": [read_email,
7     read_contacts]
8  }
```

Code4. Access Token with modified scope
Source: Own study based on [12]

Along with this token, the attacker can both read emails and read the user's contact information despite not being permitted to do so.

5.2. Verification in Implicit Mode

In the OAuth 2.0 protocol, the implicit permission granting type is mainly intended for browser-based applications that cannot securely store sensitive information, such as the client's password. In this case, the access token is sent directly to the browser after the user grants permission, and the browser passes it to the client application.

However, this process carries certain risks. If an attacker can intercept an access token (e.g., through a man-in-the-middle attack or by exploiting a browser security vulnerabilities), he can use the token to directly access user resources.

In the described scenario, the attacker, having the stolen access token in his possession, can send a request to the authorization server by manually adding a new value to the **scope** (range) parameter. The **scope** specifies which resources the application has access to.

In an ideal scenario, the authorization server should check that the scope value matches the one used when generating the token. However, this is not always checked. If the requested permissions do not exceed the level of access previously granted to that client application, an attacker can potentially access additional data without further permission from the user.

5.3. Attack scenario in OAuth 2.0

A scenario for an example attack on incorrect or missing validation of **scope** access in OAuth 2.0 is as follows (example for *Authorization Code* protocol mode):

1. The attacker Mallory constructs and sends an authorization request via a client application for permissions that are available to him,
2. The authorization server authenticates the attacker with his credentials and performs a compliant and correct authorization.
3. The authorization server generates and transmits the Authorization Code and then sends it to the attacker's client application.
4. The client application accepts the request and does not verify its state (state parameter). It assumes in advance that the authorization process has been executed correctly and started through the correct authorization address.
5. A client application (e.g., a browser application) receives an Authorization Code, with which it constructs a request to the endpoint /token to acquire an Access Token.
6. The attacker intercepts this request and adds an additional permission to the scope parameter that allows him to perform actions not allowed for him. He then sends the crafted request to the authorization server.
7. The authorization server correctly validates the received Authorization Code (since it generated it itself) and generates an Access Token based on the scope received in the new request /token. It then returns this token to Mallory.
8. The attacker is given an Access Token that grants him additional privileges that were not authorized.

The result of this action is that the attacker has access to data and permissions that he has not authorized, and also to data and permissions to which he was never entitled. If the attacker knows or can guess the possible values of the Resource Server's administrator **scope**, he can gain administrator access to the entire system.

6. No verification of the state parameter

6.1. CSRF attack scenario in OAuth 2.0

OAuth 2.0 uses the state parameter (*state*) as a safeguard against Cross-Site Request Forgery (CSRF) attacks. It is based on sending a unique, random **state** parameter in the authorization request to the authorization server. This parameter is then sent back to the client application in a response, allowing the application to verify if the response is related to the original user request.

If the client application does not generate and verify this parameter, it is vulnerable to CSRF attacks. The attacker can then create a malicious authorization request that the user unknowingly accepts, leading to unauthorized access to user resources by the attacker.

A scenario for an example attack on an unprotected or missing **state** parameter in OAuth 2.0 presents itself as follows:

1. Attacker Mallory constructs a malicious page that contains a script that sends a malicious HTTP request to the authorization server.
2. The attacker provides the Alice victim with a malicious link to this request and encourages her to click on it. It can be done this in the form of a link in an image, through an email encouraging a quick login to the site, or through another phishing method.
3. Alice, unaware, clicks on a malicious link that redirects her to a real authorization server. There, she logs in and grants

authorization convinced that everything is fine.

4. The authorization server depending on protocol flow generates and transmits the Authorization Code or Access Token and sends it to the client application.
5. The client application accepts the request and does not verify its state (**state** parameter). It assumes in advance that the authorization process has been executed correctly and started through the correct authorization address.

6.2. Implications

Although the attack does not result in the attacker directly taking over the user's credentials (Authorization Code or Access Token), it did reveal an inaccuracy in the client's implementation that could lead to several serious threats in the future, among others:

- Falsifying authorization responses of the authorization server,
- Ability to force the user to log back into the application,
- Ability to force user to re-authorize in the authorization server,
- Lack of integrity of the authorization process: the client application in no way verifies the received response from the authorization server. It is unable to identify the fraudulent response from the attacker,
- Facilitated exploitation of other attacks (e.g. on unsecured **redirect_uri**),
- If the attacker could steal Access Tokens in the client application, he could then generate his own token,
- The possibility of changing the type of authorization grant to a less secure one.

7. Summary of Experimental results

Three approaches to exploit popular vulnerabilities in the OAuth 2.0 protocol implementation are presented here (three attacks related to them were implemented). The key service in the context of security, but also the proper operation of the protocol, is the Auth Server implementation. It is responsible for, among other things:

- user authentication,
- presentation a consent screen to them,
- generation of Authorization Code,
- transmission of the Authorization Code,
- generation of Access Token,
- transferring of the Access Token,
- signing tokens,
- providing a mechanism to validate tokens.

This is a multitude of tasks, each of which is crucial in maintaining the correctness and security of the entire process. If any of these activities is implemented carelessly or inaccurately, the entire protocol can be compromised.

The first attack was based on the failure to secure and verify the **redirect_uri** parameter during the authorization process of the client application in the authorization server. During the experiment, the attacker generated a fake authorization request, which initially works correctly. The user was redirected to the authorization server, where they authenticated themself and gave permission to access their server data from resources. That resulted in the generation and return of the client application's Access Token, but this was sent to the attacker. Thus, the attacker received a token which allows full access to the authorized data without any restrictions.

The solution to protect against this attack was to implement a mechanism linking the Client Application to an appropriately matched **redirect_uri**. When registering the application with the Authorization Server, it had to provide the exact **redirect_uri** value to which tokens could be sent. After setting such an address for the Auth Client, the Auth Server could send the Authorization Code and Access Token only to approved address.

The second attack was based on missing or malfunctioning validation of the **scope** parameter during the authorization process. The attack took place without any additional user other than the attacker. The attacker started the authorization process in the Auth Server to grant access to the data available to any user. After proper authentication and approval, he received the Authorization Code. However, during the standard process of retrieving the Access Token using this Code, he modified the **scope** to also grant access to non-standard and inaccessible data to the attacker.

The solution to protect against this attack was to implement an additional field in the component that stores Authorization Codes. Information about the scope assigned to the code was added to the map responsible for associating the client's name with the corresponding Code. In this way, when retrieving a token using the */token* request, the Auth Server checked not only whether the grant belonged to that client, but also whether the scope requested in the Access Token matched the one assigned to it during authorization.

The third attack took advantage of the lack of obligation to use the state parameter during protocol operation. The specification of a protocol depicts that the **state** parameter is a *RECOMMENDED* field. This means that its presence is not mandatory for proper protocol flow. As a result, the Auth Client accepted every request to its return address (*/callback*) without verifying that it was the source of the authorization request. The attacker used phishing to persuade the victim to click on a malicious link leading to a re-authorization of the Auth Client along with another scope. If the attacker continues the process and completes the authorization, then a new Access Token containing a different authorization than the one generated previously will be passed to the Auth Client. In this way, the operation of the Auth Client can be disrupted and/or prevented.

The security solution was to add the recommended **state** parameter to Auth Server authorization requests. When generating an Auth Client request to Auth Server to endpoint */authorize*, a random string is generated which is the identifier of that session and stored in the Auth Client component. The Auth Server, while accepting the authorization request, saves and then returns as an additional parameter when sending the Authorization Code.

Auth Server then, when receiving the Authorization Code, checks whether the additionally received **state** string matches the one generated earlier. If it does, the client is assured that this Code is the answer to its request. Otherwise, the operation of the protocol is interrupted.

8. Conclusion

Through protocol analysis and experimentation, several positive and negative aspects of using the OAuth 2.0 protocol can be distinguished.

Advantages

OAuth 2.0 is now a very popular choice for API security and a method for delegating permissions to external applications. This is due to its many advantages and simplifications:

1. Enables delegated authorization: with the OAuth 2.0 protocol, applications can gain access to protected data through delegated authorization of the resource owner. The user does not have to provide their credentials to the application. whose security they do not trust and control the level of authorization they grant.
2. Refresh token mechanism: OAuth 2.0 supports the use of refresh tokens. When an access token expires, the user does not have to go through the entire process of generating a new token all over again. They only need to use the refresh token received on the first authorization to generate a new access token directly from the authorization server. In addition, the user does not need to authenticate with the authorization server once again and give authorization approval.
3. Ability to add a *user consent screen*: OAuth 2.0 allows a user to review and decide whether to share permissions in a separate step. This makes it easier for the user to maintain an appropriate level of security.
4. Flexibility: The use of different types of grants as well as a mechanism for creating custom ones gives a lot of freedom to developers in implementing the protocol. The protocol can be effectively used in web, desktop or mobile applications alike. It is possible to adapt OAuth to different scenarios.
5. Standardization: the protocol being an officially approved standard supports its interoperability. It is much easier to integrate services or applications that use the same protocol.

Disadvantages

It should be noted that despite its clear advantages and widespread recognition, OAuth 2.0 is not without its drawbacks and challenges. Discussing these aspects is key to fully understanding the details of the protocol's implementation and security:

1. Complexity: OAuth 2.0 is a complicated and quite complex protocol. While the authorization process itself is straightforward, the multitude of technical details the protocol is characterized by makes it difficult to master quickly. Implementation requires a great deal of care and full understanding so as not to leave vulnerabilities in it.
2. No defined confidentiality: The specification does not propose any cryptographic mechanism to secure connections between actors. It leaves this task to the implementing developer, who must ensure the security of the connection between all actors.
3. Lack of uniform implementation: OAuth 2.0's flexibility is its great advantage, but this can become its big disadvantage. The protocol has no reference

implementation nor official model secure adaptation. This places all responsibility for implementation security on the shoulders of developers.

4. No defined token management: Both access token and refresh token have a defined syntax with which they must be implemented. Their description also includes the sentence "(...) the token MUST remain confidential during transport and storage" [9]. However, the specification does not propose any specific solution for managing these tokens included:
   - no handling of the denial with expired access token,
   - no support for access token refresh,
   - no support for token transfer between actors,
   - no proposal for token validation and verification,
   - no proposal for cryptographic mechanisms to be used in token handling.
5. Use of the bearer tokens: The bearer token is extremely simple and convenient to use. Access is granted to any user who has this token. However, there is a high risk involved there. If the bearer token is stolen, an adversary can access protected resources without any possibility of detecting it.

The protocol allows a great deal of implementation freedom for developers and system architects. However, it carries considerable risks if done by someone unfamiliar with OAuth. An example is the lack of a requirement to use the security-critical parameters **redirect_uri** and **state**. The first one is defined as *Optional* and the second as *Recommended*. The standard basic implementation does not need to use both parameters while still being correct. The effect of this, however, is that its security is inadequate at many points, while the system itself is at risk to various vulnerabilities. Some of them have a high impact on the security of the entire system, as they allow the Access Token to be stolen silently (no **redirect_uri**). Others, on their own, posed a low threat (no **state** parameter), but when combined with other vulnerabilities not necessarily related to the protocol itself, they can become a powerful weapon against the system.

Only the introduction of suggested validations and verifications protects against these risks. Each of the fixes requires additional work and developer's ideas to solve the problem, since the specification does not offer any out-of-the-box implementation or even pseudo-code suggesting how to handle the various protocol elements in the system.

The decision to build your own implementation of the entire OAuth 2.0 protocol is a responsible one and requires a sufficiently large amount of knowledge to make a system that is fully secure. However, it should be remembered that this is not impossible thing, especially if the administrator wants to have full control over the individual steps of the protocol. On the other hand, if someone is not familiar with or does not want to spend extra time working on authorization mechanisms, they can use ready-made tools to support security systems.

In the case of a large website using multiple clients, scopes and involving different types of authorization, the use of an off-the-shelf open solution can significantly facilitate the construction and subsequent maintenance and management of such a system.

An example of an off-the-shelf solution to security issues in a system is the use of Keycloak software. The software is an open-source *Identity and Access Management* (IAM) platform developed by Red Hat [14]. It is designed to facilitate the management of user authorization and authentication in applications and services. Keycloak offers a set of features that help developers and administrators control access to resources in a secure and efficient manner. Among them is software to support and manage the OAuth 2.0 protocol [15].

For the Spring Boot framework, it may also be convenient to use the off-the-shelf authorization server functionality provided and developed by the Spring Security team. In May 2023, Spring announced on its website that the Spring Authorization Server became available in the Spring Initialzr application [16]. This announcement was accompanied by instructions on how to erect own simple environment implementing the OAuth 2.0 protocol with a very simple configuration. Such a solution implements protection against all previously tested attacks and is constantly updated with the latest solutions. At the time of writing this work (May 2024), the next major update of the project numbered 1.3.0 [17] has come out. Thus, even novice Spring developers can construct their own systems with no trouble, without having to read deeply into the protocol specification. This work can help raise awareness of potential risks and increase the level of security in systems using OAuth 2.0. It can also benefit system architects implementing their own OAuth 2.0 and/or OpenID Connect protocol solution by increasing their knowledge of common bugs that can occur in their systems.

9. Bibliography

[1] Homakov E., "How I hacked Github again", https://homakov.blogspot.com/2014/02/how-i-hacked-github-again.html (accessed: 29.05.2024).

[2] Baikar A., "Facebook OAuth Framework Vulnerability", https://www.amolbaikar.com/facebook-oauth-framework-vulnerability/ (accessed: 29.05.2024).

[3] "Chained Bugs to Leak Victim's Uber's FB Oauth Token", https://hackerone.com/reports/202781 (accessed: 29.05.2024).

[4] Reizelman M., "Creating comments on confidential issues through mass assignment", https://gitlab.com/gitlab-org/gitlab-foss/-/issues/56500 (accessed: 29-05-2024).

[5] Katz T., "Bypassing GitHub's OAuth flow", https://blog.teddykatz.com/2019/11/05/github-oauth-bypass.html (accessed: 29.05.2024).

[6] Stepankin M., "Hidden OAuth attack vectors", https://portswigger.net/research/hidden-oauth-attack-vectors (accessed: 29.05.2024).

[7] Wilson Y., Hignikar A., *Solving Identity Management in Modern Applications*, *Demystifying OAuth 2.0, OpenID Connect and SAML 2.0*, Apress, San Francisco 2019.

[8] Parecki A., "OAuth 2.0", https://oauth.net/2/ (accessed: 11.12.2023).

[9] Hardt D. (Ed.), *The OAuth 2.0 Authorization Framework*, Microsoft, Internet Engineering Task Force (IETF), 2012, https://datatracker.ietf.org/doc/html/rfc6749 (accessed: 28.05.2024).

[10] Port Swigger, "OAuth 2.0 authentication vulnerabilities", https://portswigger.net/web-security/oauth (accessed: 29.05.2024).

[11] CWE, "CWE-601: URL Redirection to Untrusted Site ('Open Redirect')", https://cwe.mitre.org/data/definitions/601.html (accessed: 29.05.2024.

[12] Parecki A., "Auth Server Example Flow", https://www.oauth.com/oauth2-servers/server-side-apps/example-flow/ (accessed: 28.05.2024).

[13] Vickie L., "Stealing OAuth Tokens With Open Redirects", https://sec.okta.com/articles/2021/02/stealing-oauth-tokens-open-redirects (accessed: 04.04.2024).

[14] Keycloak, "Authorization Services Guide", https://www.keycloak.org/docs/26.1.4/authorization_services/ (accessed: 01.04.2025).

[15] Roate N., "How to architect OAuth 2.0 authorization using Keycloak", https://www.redhat.com/architect/oauth-20-authentication-keycloak (accessed: 29.05.2024).

[16] Riesenberg S., "Spring Authorization Server is on Spring Initializr!", https://spring.io/blog/2023/05/24/spring-authorization-server-is-on-spring-initializr (accessed: 28.05.2024).

[17] Spring OAuth 2.0 Authorization Server project repository, https://github.com/spring-projects/spring-authorization-server (accessed: 28.05.2024).

Ocena stanu bezpieczeństwa systemów korzystających z protokołu autoryzacji OAuth 2.0

D. KANIA, P. GÓRNY

Artykuł przedstawia analizę stanu bezpieczeństwa systemów wykorzystujących protokół OAuth 2.0. Jest to protokół autoryzacji szeroko stosowany w serwisach internetowych, w tym największych na świecie. Charakteryzuje się dużą złożonością i skomplikowanym działaniem. Prawidłowo wykorzystywany zapewnia jednak użytkownikom znaczną wygodę w udostępnianiu danych aplikacjom internetowym z innych serwisów. Opisano tu specjalistyczne testy, które zostały przeprowadzone w celu zidentyfikowania słabych punktów w podstawowej implementacji OAuth 2.0. Na podstawie tych testów zidentyfikowano zagrożenia i ryzyka związane z niedokładną implementacją, wraz z przedstawieniem ich potencjalnych skutków.

Słowa kluczowe: autoryzacja, protokół, podatność.