Zajęcia 6

Podstawy bezpieczeństwa usług internetowych

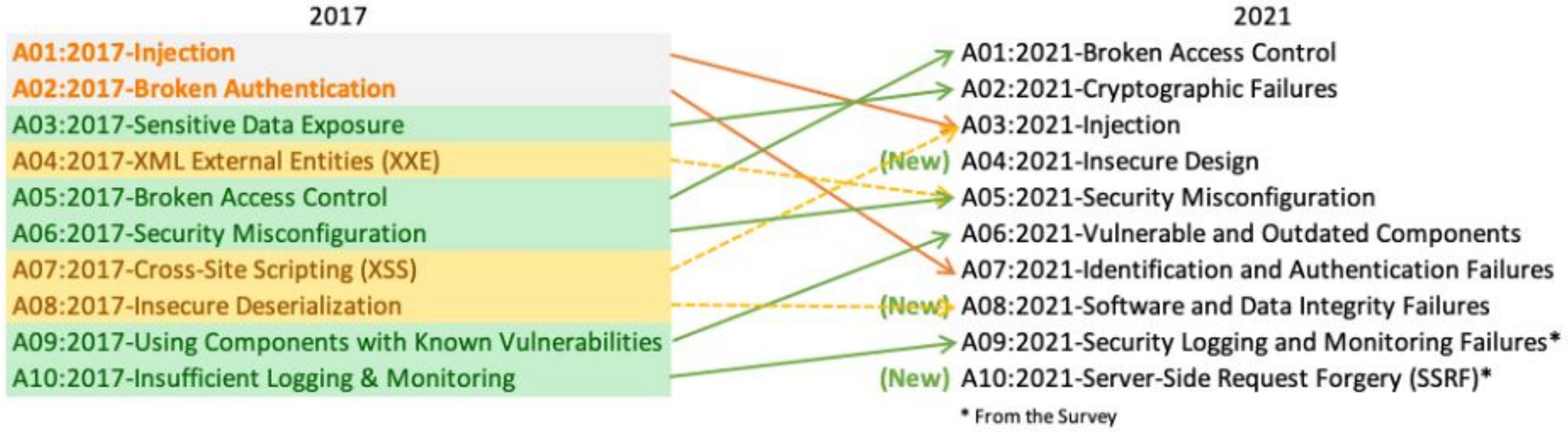
Dominik Michta
dominik.michta@allegro.pl

allegro

Agenda

1. OWASP TOP 10
2. Przykłady implementacji zabezpieczeń: https://github.com/allegro-agh-2022/lab-oauth
    a. Basic auth
    b. RBAC
    c. OAuth2
3. Pytania

OWASP TOP10



2017

- A01:2017-Injection
- A02:2017-Broken Authentication
- A03:2017-Sensitive Data Exposure
- A04:2017-XML External Entities (XXE)
- A05:2017-Broken Access Control
- A06:2017-Security Misconfiguration
- A07:2017-Cross-Site Scripting (XSS)
- A08:2017-Insecure Deserialization
- A09:2017-Using Components with Known Vulnerabilities
- A10:2017-Insufficient Logging & Monitoring

2021

- A01:2021-Broken Access Control
- A02:2021-Cryptographic Failures
- A03:2021-Injection
- (New) A04:2021-Insecure Design
- A05:2021-Security Misconfiguration
- A06:2021-Vulnerable and Outdated Components
- A07:2021-Identification and Authentication Failures
- (New) A08:2021-Software and Data Integrity Failures
- A09:2021-Security Logging and Monitoring Failures*
- (New) A10:2021-Server-Side Request Forgery (SSRF)*

* From the Survey

A01:2021 – Broken Access Control

Vulnerabilities:

1. Violation of the principle of least privilege or deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone.
2. Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool modifying API requests.
3. Permitting viewing or editing someone else's account, by providing its unique identifier (insecure direct object references)
4. Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, or a cookie or hidden field manipulated to elevate privileges or abusing JWT invalidation.

How to prevent:

1. Except for public resources, deny by default.
2. Implement access control mechanisms once and re-use them throughout the application, including minimizing Cross-Origin Resource Sharing (CORS) usage.
3. RBAC - role based access control
4. Log access control failures, alert admins when appropriate (e.g., repeated failures).
5. Rate limit API and controller access to minimize the harm from automated attack tooling.
6. Stateful session identifiers should be invalidated on the server after logout. Stateless JWT tokens should rather be short-lived so that the window of opportunity for an attacker is minimized. For longer lived JWTs it's highly recommended to follow the OAuth standards to revoke access.

A02:2021 – Cryptographic Failures

Vulnerabilities:

1. Is any data transmitted in clear text? This concerns protocols such as HTTP, SMTP, FTP also using TLS upgrades like STARTTLS. External internet traffic is hazardous. Verify all internal traffic, e.g., between load balancers, web servers, or back-end systems.
2. Are any old or weak cryptographic algorithms or protocols used either by default or in older code?
3. Are default crypto keys in use, weak crypto keys generated or re-used, or is proper key management or rotation missing? Are crypto keys checked into source code repositories?
4. Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are needed?

How to prevent:

1. Classify data processed, stored, or transmitted by an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.
2. Don't store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.
3. Make sure to encrypt all sensitive data at rest.
4. Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
5. Encrypt all data in transit with secure protocols such as TLS with forward secrecy (FS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).

A03:2021 – Injection

Vulnerabilities:

1. User-supplied data is not validated, filtered, or sanitized by the application.
2. Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
3. Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
4. Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.

How to prevent:

1. The preferred option is to use a safe API, which avoids using the interpreter entirely, provides a parameterized interface, or migrates to Object Relational Mapping Tools (ORMs).
2. Use positive server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
3. For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
4. Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

A04:2021 – Insecure Design

How to prevent:

1. Establish and use a secure development lifecycle with AppSec professionals to help evaluate and design security and privacy-related controls
2. Establish and use a library of secure design patterns or paved road ready to use components
3. Use threat modeling for critical authentication, access control, business logic, and key flows
4. Integrate security language and controls into user stories
5. Integrate plausibility checks at each tier of your application (from frontend to backend)
6. Write unit and integration tests to validate that all critical flows are resistant to the threat model. Compile use-cases and misuse-cases for each tier of your application.
7. Segregate tier layers on the system and network layers depending on the exposure and protection needs
8. Segregate tenants robustly by design throughout all tiers
9. Limit resource consumption by user or service

A05:2021 – Security Misconfiguration

Vulnerabilities:

1. Missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services.
2. Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, or privileges).
3. Error handling reveals stack traces or other overly informative error messages to users.
4. The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values.
5. The software is out of date or vulnerable (see A06:2021-Vulnerable and Outdated Components).

How to prevent:

1. A repeatable hardening process makes it fast and easy to deploy another environment that is appropriately locked down. Development, QA, and production environments should all be configured identically, with different credentials used in each environment. This process should be automated to minimize the effort required to set up a new secure environment.
2. A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.
3. A segmented application architecture provides effective and secure separation between components or tenants, with segmentation, containerization, or cloud security groups (ACLs).

A06:2021 – Vulnerable and Outdated Components

You are likely vulnerable:

1. If the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries.
2. If you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use.
3. If you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure to fixed vulnerabilities.
4. If software developers do not test the compatibility of updated, upgraded, or patched libraries.

How to prevent:

1. Remove unused dependencies, unnecessary features, components, files, and documentation.
2. Continuously inventory the versions of both client-side and server-side components (e.g., frameworks, libraries) and their dependencies using tools like versions, OWASP Dependency Check, retire.js, etc. Continuously monitor sources like Common Vulnerability and Exposures (CVE) and National Vulnerability Database (NVD) for vulnerabilities in the components. Use software composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components you use.
3. Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component (See A08:2021-Software and Data Integrity Failures).
4. Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.

A07:2021 – Identification and Authentication Failures

Vulnerable application:

1. Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords.
2. Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin".
3. Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe.
4. Uses plain text, encrypted, or weakly hashed passwords data stores (see A02:2021-Cryptographic Failures).
5. Has missing or ineffective multi-factor authentication.
6. Does not correctly invalidate Session IDs. User sessions or authentication tokens (mainly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

How to prevent:

1. Where possible, implement multi-factor authentication to prevent automated credential stuffing, brute force, and stolen credential reuse attacks.
2. Do not ship or deploy with any default credentials, particularly for admin users.
3. Implement weak password checks, such as testing new or changed passwords against the top 10,000 worst passwords list.
4. Limit or increasingly delay failed login attempts, but be careful not to create a denial of service scenario. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
5. Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session identifier should not be in the URL, be securely stored, and invalidated after logout, idle, and absolute timeouts.

A08:2021 – Software and Data Integrity Failures

How to prevent:

1. Use digital signatures or similar mechanisms to verify the software or data is from the expected source and has not been altered.
2. Ensure libraries and dependencies, such as npm or Maven, are consuming trusted repositories. If you have a higher risk profile, consider hosting an internal known-good repository that's vetted.
3. Ensure that a software supply chain security tool, such as OWASP Dependency Check or OWASP CycloneDX, is used to verify that components do not contain known vulnerabilities
4. Ensure that there is a review process for code and configuration changes to minimize the chance that malicious code or configuration could be introduced into your software pipeline.
5. Ensure that your CI/CD pipeline has proper segregation, configuration, and access control to ensure the integrity of the code flowing through the build and deploy processes.
6. Ensure that unsigned or unencrypted serialized data is not sent to untrusted clients without some form of integrity check or digital signature to detect tampering or replay of the serialized data

A09:2021 – Security Logging and Monitoring Failures

Vulnerabilities:

1. Auditable events, such as logins, failed logins, and high-value transactions, are not logged.
2. Warnings and errors generate no, inadequate, or unclear log messages.
3. Logs of applications and APIs are not monitored for suspicious activity.
4. Logs are only stored locally.
5. Appropriate alerting thresholds and response escalation processes are not in place or effective.
6. Penetration testing and scans by dynamic application security testing (DAST) tools (such as OWASP ZAP) do not trigger alerts.
7. The application cannot detect, escalate, or alert for active attacks in real-time or near real-time.

How to prevent:

1. Ensure all login, access control, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts and held for enough time to allow delayed forensic analysis.
2. Ensure that logs are generated in a format that log management solutions can easily consume.
3. Ensure log data is encoded correctly to prevent injections or attacks on the logging or monitoring systems.
4. Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
5. DevSecOps teams should establish effective monitoring and alerting such that suspicious activities are detected and responded to quickly.

A10:2021 – Server-Side Request Forgery (SSRF)

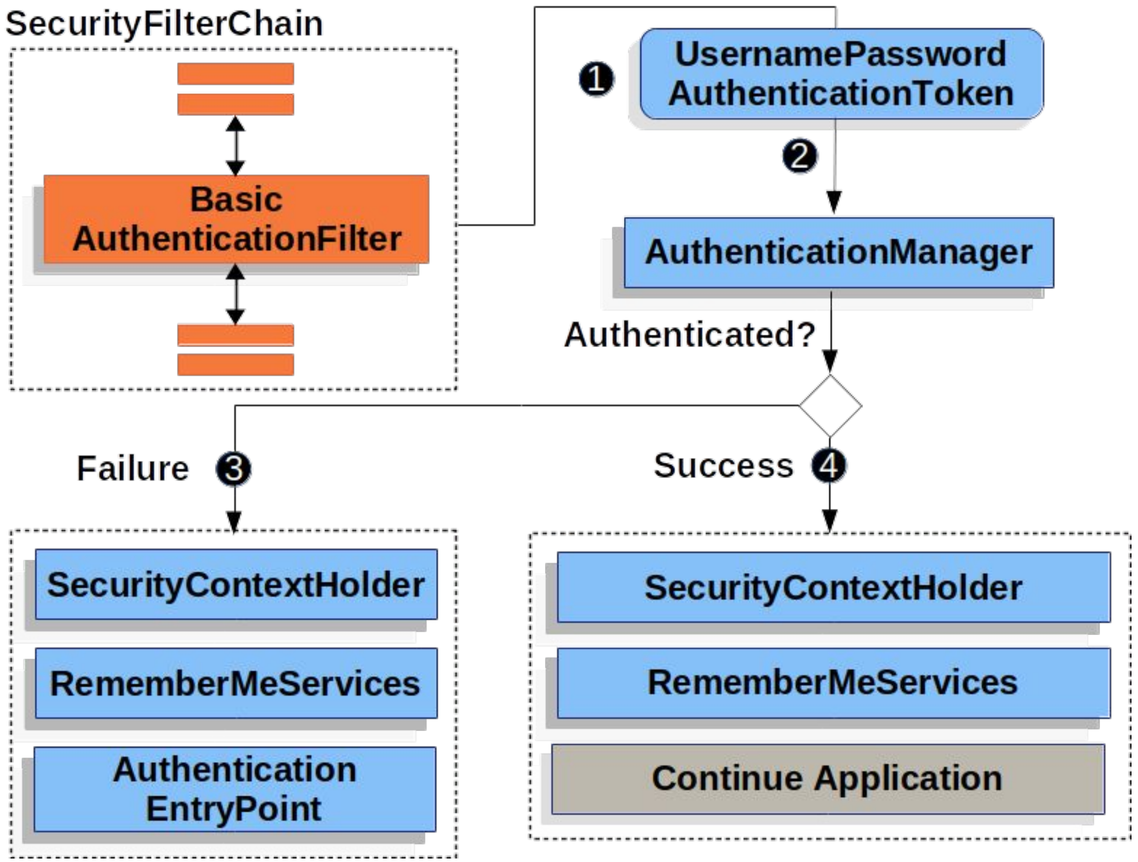
SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network access control list (ACL).

How to prevent:

1. Segment remote resource access functionality in separate networks to reduce the impact of SSRF
2. Enforce "deny by default" firewall policies or network access control rules to block all but essential intranet traffic.
3. Sanitize and validate all client-supplied input data
4. Enforce the URL schema, port, and destination with a positive allow list
5. Do not send raw responses to clients
6. Disable HTTP redirections
7. Be aware of the URL consistency to avoid attacks such as DNS rebinding and "time of check, time of use" (TOCTOU) race conditions

allegro

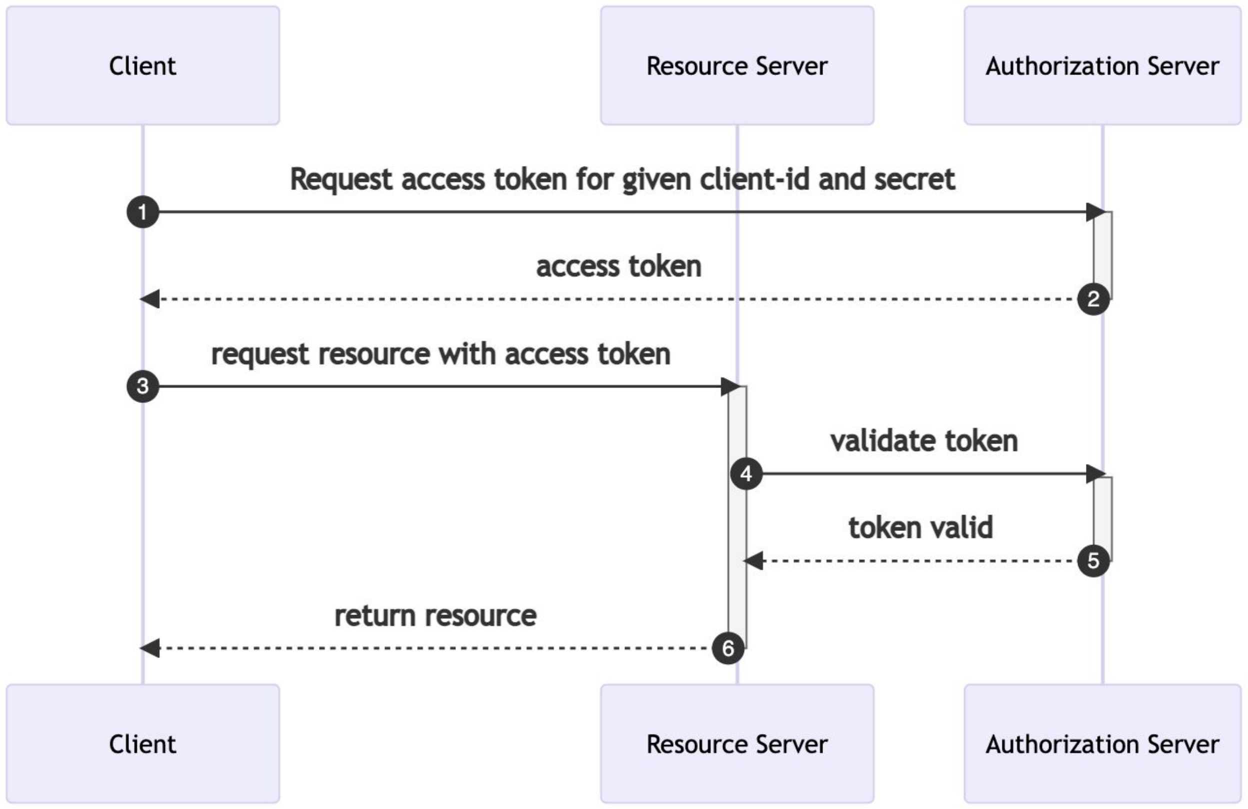
Spring Security



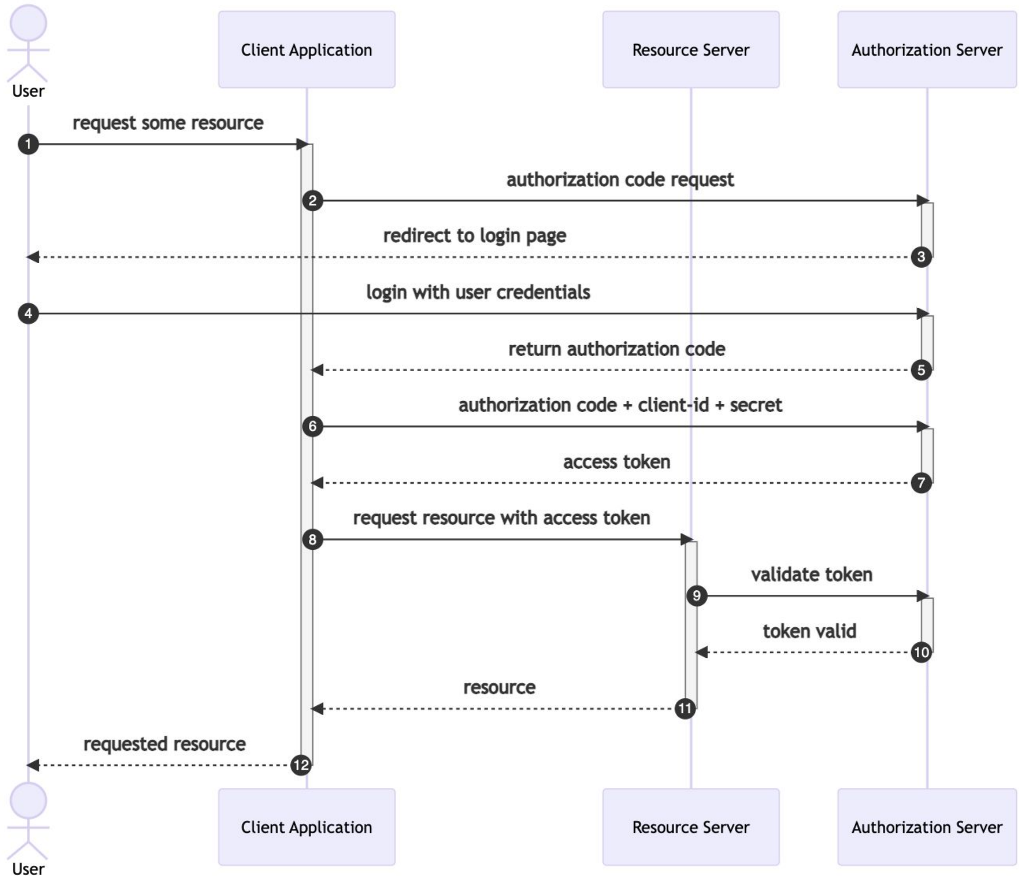
OAuth 2

- Właściciel zasobu (ang. resource owner) – osoba lub inny podmiot będący właścicielem zasobu chronionego poprzez wymóg autoryzacji.
- Klient (ang. client) – w rozumieniu OAuth aplikacja, która chce uzyskać dostęp do zasobu.
- Serwer autoryzacyjny (ang. authorization server) – serwer udzielający w imieniu właściciela zasobu poświadczenia, na podstawie którego klient może uzyskać dostęp do chronionego zasobu.
- Serwer zasobu (ang. resource server) – serwer przechowujący chronione zasoby. W wielu przypadkach, w sensie logicznym, serwer ten jest tym samym, czym serwer autoryzacyjny.
- Zakres (ang. scope) – abstrakcyjna definicja określająca wybrany fragment uprawnień do chronionych zasobów. Definiowanie nazw zakresów oraz ich znaczenia leży po stronie serwera autoryzacyjnego.
- Access token – token wystawiany przez serwer autoryzacyjny, który klient może wykorzystać, by uzyskać dostęp do określonych zasobów.

Client Credentials Grant Type



Authorization Code Grant Type



Pytania?

dominik.michta@allegro.pl
Slack: allegro-agh-2022

Dziękuję!