|  |  |
| --- | --- |
| Vulnerability Name | Broken Object Level Authorization |
| Target URL | <http://localhost:3000/api/getNote?username=FUZZ> |
| Steps to Produce | 1. Visit the target URL 2. Fuzz the parameter with alphanumeric values 3. You will get BOLA |
| Severity | Critical |
| CVSS Score | 9.8 |
| Attack Vector | CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H |
| Proof Of Concept | STEP 1 – We have Endpoint where we can modify parameter and fetch the information about the user     * Result in browser     STEP 2 – Try to access other user data by passing different user value in parameter |
| Impact | Unauthorized access to other users’ objects can result in data disclosure to unauthorized parties, data loss, or data manipulation. Under certain circumstances, unauthorized access to objects can also lead to full account takeover. |
| Remediation | * Implement a proper authorization mechanism that relies on the user policies and hierarchy. * Use the authorization mechanism to check if the logged-in user has access to perform the requested action on the record in every function that uses an input from the client to access a record in the database. * Prefer the use of random and unpredictable values as GUIDs for records' IDs. * Write tests to evaluate the vulnerability of the authorization mechanism. Do not deploy changes that make the tests fail. |
| Reference | <https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/> |

|  |  |
| --- | --- |
| Vulnerability Name | Broken Authentication |
| Target URL | <http://localhost:3000/api/login> |
| Steps To Produce | 1. Try to modify token 2. We can see that token includes payload isAdmin 3. We try to modify the payload parameter 4. Server dose not update on server side 5. So we brute force for the secrete 6. We got secrete 7. We generated new symmetric private key so we can sign the token 8. We tampered the values and got login |
| Severity | Critical |
| CVSS Score | 9.8 |
| Attack Vector | CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H |
| Proof Of Concept | STEP 1 – We have endpoint lets try to pass valid Credentials     * We have JWT Token here      * Let’s forward this to Burp Suite by setting proxy in Postman      * Send the request from postman and let’s listen the request on Burp Suite      * We got Request on Burp Suite      * Send it to repeater      * Let’s use JWT Editor and try to edit values in token and bypass Authentication      * In token we have payload which named isAdmin with value false * Let modify it to true and then send and see the response      * We got 200 OK Success but if we see token in response we can see that parameter or payload value in token is still false      * So, what’s problem here is that our send token is not signed for admin it was signed for user Yash we need to sign token for admin * To do so we need secrete key to sign the token   STEP 2 – Brute Force JWT Token to Find Secret we will use Token with user set as admin and isAdmin set false.     * Let’s brute force using Hash cat * Wordlist we will use is <https://github.com/wallarm/jwt-secrets>     SYNTAX – Hash cat -m 16500 -a 0 (token) wordlist.txt     * Got secrete: secret123 * Let’s sign the token * Go to burp Decoder tab and paste the secrete key * Encode as Base-64 * Copy the value      * Go to JWT Editor and click on New symmetric key      * Click on Generate      * Replace the value with the Secrete that you have encoded in key field      * Click Ok. Now we can use it to sign any token     STEP 3 – From Postman try to access /api/profile endpoint and try to tamper token and access admin user profile     * Request got Intercepted in Burp Suite      * let’s send to repeater * modify the token values      * Click on Sign      * Select the secrete we have created      * Then send the request     Here we got Flag |
| Impact | Attackers can gain complete control of other users’ accounts in the system, read their personal data, and perform sensitive actions on their behalf. Systems are unlikely to be able to distinguish attackers’ actions from legitimate user ones. |
| Remediations | * Make sure you know all the possible flows to authenticate to the API (mobile/ web/deep links that implement one-click authentication/etc.). Ask your engineers what flows you missed. * Read about your authentication mechanisms. Make sure you understand what and how they are used. OAuth is not authentication, and neither are API keys. * Don't reinvent the wheel in authentication, token generation, or password storage. Use the standards. * Credential recovery/forgot password endpoints should be treated as login endpoints in terms of brute force, rate limiting, and lockout protections. * Require re-authentication for sensitive operations (e.g. changing the account owner email address/2FA phone number). * Use the [OWASP Authentication Cheatsheet](https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html). * Where possible, implement multi-factor authentication. * Implement anti-brute force mechanisms to mitigate credential stuffing, dictionary attacks, and brute force attacks on your authentication endpoints. This mechanism should be stricter than the regular rate limiting mechanisms on your APIs. * Implement [account lockout](https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/04-Authentication_Testing/03-Testing_for_Weak_Lock_Out_Mechanism(OTG-AUTHN-003))/captcha mechanisms to prevent brute force attacks against specific users. Implement weak-password checks. * API keys should not be used for user authentication. They should only be used for [API clients](https://cloud.google.com/endpoints/docs/openapi/when-why-api-key) authentication. |
| Reference | <https://owasp.org/API-Security/editions/2023/en/0xa2-broken-authentication/> |

|  |  |
| --- | --- |
| Vulnerability Name | Broken Object Property Level Authorization |
| Target URL | <http://localhost:3000/api/scores> |
| Steps To Produce | 1. Go to /api/score 2. You can see there is score parameter 3. So, let’s try to register new user and try to add extra score parameter 4. After registering new user, we get token for that user and set that token in request so we don’t face any auth related issues 5. Let’s try to face user account score details we can see the values of scores got assigned |
| Severity | Critical |
| CVSS Score | 9.8 |
| Attack Vector | CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H |
| Proof Of Concept | STEP 1 – Here on endpoint, we can see that we can see the score     * So, there is Key Value named Score as we can see in POC * We can try Mass Assignment to Add-up parameters form client side * For Example, we can assign isAdmin = true while registering for user lets try it * Let’s go for /register page and try to assign the score parameter for the new user we are registering      * Here we have passed the parameter we have seen that request is successful let’s try to verify it get update on server side      * Let’s try to copy and update the token and access to the try to fetch it on /api/scores * Here we got the Misassignment means target is vulnerable |
| Impact | Unauthorized access to private/sensitive object properties may result in data disclosure, data loss, or data corruption. Under certain circumstances, unauthorized access to object properties can lead to privilege escalation or partial/full account takeover. |
| Remediations | * When exposing an object using an API endpoint, always make sure that the user should have access to the object's properties you expose. * Avoid using generic methods such as to\_json() and to\_string(). Instead, cherry-pick specific object properties you specifically want to return. * If possible, avoid using functions that automatically bind a client's input into code variables, internal objects, or object properties ("Mass Assignment"). * Allow changes only to the object's properties that should be updated by the client. * Implement a schema-based response validation mechanism as an extra layer of security. As part of this mechanism, define and enforce data returned by all API methods. * Keep returned data structures to the bare minimum, according to the business/functional requirements for the endpoint. |
| Reference | <https://owasp.org/API-Security/editions/2023/en/0xa3-broken-object-property-level-authorization/> |

|  |  |
| --- | --- |
| Vulnerability Name | Unrestricted Resource Consumption |
| Target URL | <http://localhost:3000/api/profile/upload> |
| Steps To Produce | 1. Create image with large size of data 2. Try to upload using upload functionality 3. If file is uploaded successfully then we can say it’s Unrestricted Resource Consumption Vulnerability |
| Severity | Critical |
| CVSS Score | 9.8 |
| Attack Vector | CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H |
| Proof Of Concept | STEP 1 – We have upload functionality on /upload API endpoint    API requests consume resources such as CPU/memory, bandwidth, storage, and integrations with other services. Attackers can cause high resource consumption by sending excessive requests, leading to unresponsive APIs or increased business expenses.  STEP 2 – Let’s Create an Image with huge resource and upload it using functionality    We created One file now let’s Increase its size using dd command    Let’s try to share file on windows and try to upload on endpoint    Try to upload file bigfile.jpeg    Here we uploaded file successfully and got the flag |
| Impact | Exploitation can lead to DoS due to resource starvation, but it can also lead to operational costs increase such as those related to the infrastructure due to higher CPU demand, increasing cloud storage needs, etc. |
| Remediations | * Use a solution that makes it easy to limit [memory](https://docs.docker.com/config/containers/resource_constraints/#memory), [CPU](https://docs.docker.com/config/containers/resource_constraints/#cpu), [number of restarts](https://docs.docker.com/engine/reference/commandline/run/#restart), [file descriptors, and processes](https://docs.docker.com/engine/reference/commandline/run/#ulimit) such as Containers / Serverless code (e.g. Lambdas). * Define and enforce a maximum size of data on all incoming parameters and payloads, such as maximum length for strings, maximum number of elements in arrays, and maximum upload file size (regardless of whether it is stored locally or in cloud storage). * Implement a limit on how often a client can interact with the API within a defined timeframe (rate limiting). * Rate limiting should be fine-tuned based on the business needs. Some API Endpoints might require stricter policies. * Limit/throttle how many times or how often a single API client/user can execute a single operation (e.g. validate an OTP, or request password recovery without visiting the one-time URL). * Add proper server-side validation for query string and request body parameters, specifically the one that controls the number of records to be returned in the response. * Configure spending limits for all service providers/API integrations. When setting spending limits is not possible, billing alerts should be configured instead. |
| Reference | <https://owasp.org/API-Security/editions/2023/en/0xa4-unrestricted-resource-consumption/> |

|  |  |
| --- | --- |
| Vulnerability Name | Broken Function Level Authorization |
| Target URL | <http://localhost:3000/api/user/user> |
| Steps To Produce | 1. Try to fetch other details by changing username at end 2. We have access of other user’s data by modifying URL 3. Let’s try to Use DELETE Method and try to delete user 4. We can access admin functionality that is meant not to be accessible for normal users |
| Severity | Critical |
| CVSS Score | 9.8 |
| Attack Vector | CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H |
| Proof Of Concept | STEP 1 – Try to see that if we can access another user’s data through endpoint    STEP 2 – Let’s try to DELETE user and we can access this functionality which Is meant for admin only then its valid vulnerability in API    As we can see we have successfully deleted user and retrieved flag |
| Impact | Such flaws allow attackers to access unauthorized functionality. Administrative functions are key targets for this type of attack and may lead to data disclosure, data loss, or data corruption. Ultimately, it may lead to service disruption. |
| Remediations | * The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific roles for access to every function. * Review your API endpoints against function level authorization flaws, while keeping in mind the business logic of the application and groups hierarchy. * Make sure that all of your administrative controllers inherit from an administrative abstract controller that implements authorization checks based on the user's group/role. * Make sure that administrative functions inside a regular controller implement authorization checks based on the user's group and role. |
| Reference | <https://owasp.org/API-Security/editions/2023/en/0xa5-broken-function-level-authorization/> |

|  |  |
| --- | --- |
| Vulnerability Name | Unrestricted Access to Sensitive Business Flows |
| Target URL | <http://localhost:3000/api/getticket> |
| Steps To Produce | 1. Make request on endpoint 2. Intercept in Burp and send to repeater 3. Try to send 100 Null payloads |
| Severity | Critical |
| CVSS Score | 9..8 |
| Attack Vector | CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H |
| Proof Of Concept | STEP 1 – this vulnerability, consider an ecommerce application that is about to run a flash sale wherein limited amount of an exclusive item is sold. Customers eagerly wait for the sale to start. When the sale starts, an attacker uses an automated script to quickly order all the items on the sale. Using this opportunity of scarcity of the exclusive products, they resell at higher prices to make a profit.    STEP 2 – Send request to burp to intercept    STEP 3 – Send it to Intruder    STEP 4 – In Intruder set Null payloads    STEP 5 – Start Attack and after filter using length    We got the flag |
| Impact | In general, technical impact is not expected. Exploitation might hurt the business in different ways, for example: prevent legitimate users from purchasing a product, or lead to inflation in the internal economy of a game. |
| Remediations | * Device fingerprinting: denying service to unexpected client devices (e.g headless browsers) tends to make threat actors use more sophisticated solutions, thus more costly for them * Human detection: using either captcha or more advanced biometric solutions (e.g. typing patterns) * Non-human patterns: analyse the user flow to detect non-human patterns (e.g. the user accessed the "add to cart" and "complete purchase" functions in less than one second) * Consider blocking IP addresses of Tor exit nodes and well-known proxies |
| Reference | <https://owasp.org/API-Security/editions/2023/en/0xa6-unrestricted-access-to-sensitive-business-flows/> |

|  |  |
| --- | --- |
| Vulnerability Name | Server-Side Request Forgery |
| Target URL | <http://localhost:3000/api/addNoteWithLink> |
| Steps To Produce | 1. Modify the link to localhost 2. Find open port by fuzzing the parameter |
| Severity | Critical |
| CVSS Score | 9.8 |
| Attack Vector | CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H |
| Proof Of Concept | STEP 1 – We have parameter it is requesting resource from a particular link    STEP 2 – Let’s Modify it to local host and try to fetch from local host and fuzz port number with burp intruder    Here we got another flag |
| Impact | Successful exploitation might lead to internal services enumeration (e.g. port scanning), information disclosure, bypassing firewalls, or other security mechanisms. In some cases, it can lead to DoS or the server being used as a proxy to hide malicious activities. |
| Remediations | * Isolate the resource fetching mechanism in your network: usually these features are aimed to retrieve remote resources and not internal ones. * Whenever possible, use allow lists of:   1. Remote origins users are expected to download resources from (e.g. Google Drive, Gravatar, etc.)   2. URL schemes and ports   3. Accepted media types for a given functionality * Disable HTTP redirections. * Use a well-tested and maintained URL parser to avoid issues caused by URL parsing inconsistencies. * Validate and sanitize all client-supplied input data. * Do not send raw responses to clients. |
| Reference | <https://owasp.org/API-Security/editions/2023/en/0xa7-server-side-request-forgery/> |

|  |  |
| --- | --- |
| Vulnerability Name | Security Misconfiguration |
| Target URL | <http://localhost:3000/api/user/user> |
| Steps To Produce | 1. Modify the token bit 2. Try to fetch the profile 3. You will get an error which revels details |
| Severity | Critical |
| CVSS Score | 9.8 |
| Attack Vector | CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H |
| Proof Of Concept | STEP 1 – Modify the token and try if we can fetch the details    STEP 2 – We removed last letter form token and saved and try to fetch profile    We got the flag |
| Impact | Security misconfigurations not only expose sensitive user data, but also system details that can lead to full server compromise. |
| Remediations | * Ensure that all API communications from the client to the API server and any downstream/upstream components happen over an encrypted communication channel (TLS), regardless of whether it is an internal or public-facing API. * Be specific about which HTTP verbs each API can be accessed by: all other HTTP verbs should be disabled (e.g. HEAD). * APIs expecting to be accessed from browser-based clients (e.g., WebApp front-end) should, at least:   1. implement a proper Cross-Origin Resource Sharing (CORS) policy   2. include applicable Security Headers * Restrict incoming content types/data formats to those that meet the business/ functional requirements. |
| Reference | <https://owasp.org/API-Security/editions/2023/en/0xa8-security-misconfiguration/> |

|  |  |
| --- | --- |
| Vulnerability Name | Improper Inventory Management |
| Target URL | <http://localhost:3000/api/allChallenges> |
| Steps To Produce | 1. Find the field or object you can modify 2. Modify the released to unreleased 3. You will get details about unreleased products |
| Severity | Critical |
| CVSS Score | 9.8 |
| Attack Vector | CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H |
| Proof Of Concept | STEP 1 – Sometimes developer creates new endpoint or fields but try to exploit that fields or endpoint    STEP 2 – Change field to unreleased    We got the flag |
| Impact | Attackers can gain access to sensitive data, or even take over the server. Sometimes different API versions/deployments are connected to the same database with real data. Threat agents may exploit deprecated endpoints available in old API versions to get access to administrative functions or exploit known vulnerabilities. |
| Remediations | * Inventory all API hosts and document important aspects of each one of them, focusing on the API environment (e.g. production, staging, test, development), who should have network access to the host (e.g. public, internal, partners) and the API version. * Inventory integrated services and document important aspects such as their role in the system, what data is exchanged (data flow), and their sensitivity. * Document all aspects of your API such as authentication, errors, redirects, rate limiting, cross-origin resource sharing (CORS) policy, and endpoints, including their parameters, requests, and responses. * Generate documentation automatically by adopting open standards. Include the documentation build in your CI/CD pipeline. * Make API documentation available only to those authorized to use the API. * Use external protection measures such as API security specific solutions for all exposed versions of your APIs, not just for the current production version. * Avoid using production data with non-production API deployments. If this is unavoidable, these endpoints should get the same security treatment as the production ones. * When newer versions of APIs include security improvements, perform a risk analysis to inform the mitigation actions required for the older versions. For example, whether it is possible to backport the improvements without breaking API compatibility or if you need to take the older version out quickly and force all clients to move to the latest version. |
| Reference | <https://owasp.org/API-Security/editions/2023/en/0xa9-improper-inventory-management/> |

|  |  |
| --- | --- |
| Vulnerability Name | Unsafe Consumption of APIs |
| Target URL | <http://localhost:3000/api/login> |
| Steps To Produce | 1. Go to login endpoint 2. Try injection payloads 3. Lack of input validation on endpoint gives access to account takeover |
| Severity | Critical |
| CVSS Score | 9.8 |
| Attack Vector | CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H |
| Proof Of Concept | STEP 1 – We have login endpoint lets try injection payloads to test is there any input validation    STEP 2 – Let’s try to inject NoSQL payload    We got flag |
| Impact | The impact varies according to what the target API does with pulled data. Successful exploitation may lead to sensitive information exposure to unauthorized actors, many kinds of injections, or denial of service. |
| Remediations | * When evaluating service providers, assess their API security posture. * Ensure all API interactions happen over a secure communication channel (TLS). * Always validate and properly sanitize data received from integrated APIs before using it. * Maintain an allowlist of well-known locations integrated APIs may redirect yours to: do not blindly follow redirects. |
| Reference | <https://owasp.org/API-Security/editions/2023/en/0xaa-unsafe-consumption-of-apis/> |