API1:2023 Broken Object Level Authorization

Threat agents/Attack vectors	Security Weakness	Impacts
API Specific : Exploitability **Easy**	Prevalence **Widespread** : Detectability **Easy**	Technical **Moderate** : Business Specific
Attackers can exploit API endpoints that are vulnerable to broken object-level authorization by manipulating the ID of an object that is sent within the request. Object IDs can be anything from sequential integers, UUIDs, or generic strings. Regardless of the data type, they are easy to identify in the request target (path or query string parameters), request headers, or even as part of the request payload.	This issue is extremely common in API-based applications because the server component usually does not fully track the client's state, and instead, relies more on parameters like object IDs, that are sent from the client to decide which objects to access. The server response is usually enough to understand whether the request was successful.	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.

Is the API Vulnerable?

Object level authorization is an access control mechanism that is usually implemented at the code level to validate that a user can only access the objects that they should have permissions to access.

Every API endpoint that receives an ID of an object, and performs any action on the object, should implement object-level authorization checks. The checks should validate that the logged-in user has permissions to perform the requested action on the requested object.

Failures in this mechanism typically lead to unauthorized information disclosure, modification, or destruction of all data.

Comparing the user ID of the current session (e.g. by extracting it from the JWT token) with the vulnerable ID parameter isn't a sufficient solution to solve Broken Object Level Authorization (BOLA). This approach could address only a small subset of cases.

In the case of BOLA, it's by design that the user will have access to the vulnerable API endpoint/function. The violation happens at the object level, by manipulating the ID. If an attacker manages to access an API endpoint/function they should not have access to - this is a case of Broken Function Level Authorization (BFLA) rather than BOLA.

Example Attack Scenarios

Scenario #1

An e-commerce platform for online stores (shops) provides a listing page with the revenue charts for their hosted shops. Inspecting the browser requests, an attacker can identify the API endpoints used as a data source for those charts and their pattern: `/shops/{shopName}/revenue_data.json`. Using another API endpoint, the attacker can get the list of all hosted shop names. With a simple script to manipulate the names in the list, replacing `{shopName}` in the URL, the attacker gains access to the sales data of thousands of e-commerce stores.

Scenario #2

An automobile manufacturer has enabled remote control of its vehicles via a mobile API for communication with the driver's mobile phone. The API enables the driver to remotely start and stop the engine and lock and unlock the doors. As part of this flow, the user sends the Vehicle Identification Number (VIN) to the API. The API fails to validate that the VIN represents a vehicle that belongs to the logged in user, which leads to a BOLA vulnerability. An attacker can access vehicles that don't belong to him.

Scenario #3

An online document storage service allows users to view, edit, store and delete their documents. When a user's document is deleted, a GraphQL mutation with the document ID is sent to the API.

```
POST /graphql
{
  "operationName":"deleteReports",
  "variables":{
    "reportKeys":["<DOCUMENT_ID>"]
  },
  "query":"mutation deleteReports($siteId: ID!, $reportKeys: [String]!) {
    {
      deleteReports(reportKeys: $reportKeys)
    }
  }"
}
```

Since the document with the given ID is deleted without any further permission checks, a user may be able to delete another user's document.

How To Prevent

- 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.

Authorization Cheat Sheet¶

Introduction¶

Authorization may be defined as "the process of verifying that a requested action or service is approved for a specific entity" (NIST). Authorization is distinct from authentication which is the process of verifying an entity's identity. When designing and developing a software solution, it is important to keep these distinctions in mind. A user who has been authenticated (perhaps by providing a username and password) is often not authorized to access every resource and perform every action that is technically possible through a system. For example, a web app may have both regular users and admins, with the admins being able to perform actions the average user is not privileged to do so, even though they have been authenticated. Additionally, authentication is not always required for accessing resources; an unauthenticated user may be authorized to access certain public resources, such as an image or login page, or even an entire web app.

The objective of this cheat sheet is to assist developers in implementing authorization logic that is robust, appropriate to the app's business context, maintainable, and scalable. The guidance provided in this cheat sheet should be applicable to all phases of the development lifecycle and flexible enough to meet the needs of diverse development environments.

Flaws related to authorization logic are a notable concern for web apps. Broken Access Control was ranked as the most concerning web security vulnerability in OWASP's 2021 Top 10 and asserted to have a "High" likelihood of exploit by MITRE's CWE program. Furthermore, according to Veracode's State of Software Vol. 10, Access Control was among the more common of OWASP's Top 10 risks to be involved in exploits and security incidents despite being among the least prevalent of those examined.

The potential impact resulting from exploitation of authorization flaws is highly variable, both in form and severity. Attackers may be able to read, create, modify, or delete resources that were meant to be protected (thus jeopardizing their confidentiality, integrity, and/or availability); however, the actual impact of such actions is necessarily linked to the criticality and sensitivity of the compromised resources. Thus, the business cost of a successfully exploited authorization flaw can range from very low to extremely high.

Both entirely unauthenticated outsiders and authenticated (but not necessarily authorized) users can take advantage of authorization weaknesses. Although

honest mistakes or carelessness on the part of non-malicious entities may enable authorization bypasses, malicious intent is typically required for access control threats to be fully realized. Horizontal privilege elevation (i.e. being able to access another user's resources) is an especially common weakness that an authenticated user may be able to take advantage of. Faults related to authorization control can allow malicious insiders and outsiders alike to view, modify, or delete sensitive resources of all forms (databases records, static files, personally identifiable information (PII), etc.) or perform actions, such as creating a new account or initiating a costly order, that they should not be privileged to do. Furthermore, if logging related to access control is not properly set-up, such authorization violations may go undetected or a least remain unattributable to a particular individual or group.

Recommendations¶

Enforce Least Privileges¶

As a security concept, Least Privileges refers to the principle of assigning users only the minimum privileges necessary to complete their job. Although perhaps most commonly applied in system administration, this principle has relevance to the software developer as well. Least Privileges must be applied both horizontally and vertically. For example, even though both an accountant and sales representative may occupy the same level in an organization's hierarchy, both require access to different resources to perform their jobs. The accountant should likely not be granted access to a customer database and the sales representative should not be able to access payroll data. Similarly, the head of the sales department is likely to need more privileged access than their subordinates.

Failure to enforce least privileges in an application can jeopardize the confidentiality of sensitive resources. Mitigation strategies are applied primarily during the Architecture and Design phase (see CWE-272); however, the principle must be addressed throughout the SDLC.

Consider the following points and best practices:

- During the design phase, ensure trust boundaries are defined. Enumerate the types of users that will be accessing the system, the resources exposed and the operations (such as read, write, update, etc) that might be performed on

those resources. For every combination of user type and resource, determine what operations, if any, the user (based on role and/or other attributes) must be able to perform on that resource. For an ABAC system ensure all categories of attributes are considered. For example, a Sales Representative may need to access a customer database from the internal network during working hours, but not from home at midnight.

- Create tests that validate that the permissions mapped out in the design phase are being correctly enforced.
- After the app has been deployed, periodically review permissions in the system for "privilege creep"; that is, ensure the privileges of users in the current environment do not exceed those defined during the design phase (plus or minus any formally approved changes).
- Remember, it is easier to grant users additional permissions rather than to take away some they previously enjoyed. Careful planning and implementation of Least Privileges early in the SDLC can help reduce the risk of needing to revoke permissions that are later deemed overly broad.

Deny by Default¶

Even when no access control rules are explicitly matched, the application cannot remain neutral when an entity is requesting access to a particular resource. The application must always make a decision, whether implicitly or explicitly, to either deny or permit the requested access. Logic errors and other mistakes relating to access control may happen, especially when access requirements are complex; consequently, one should not rely entirely on explicitly defined rules for matching all possible requests. For security purposes an application should be configured to deny access by default.

Consider the following points and best practices:

- Adopt a "deny-by-default" mentality both during initial development and whenever new functionality or resources are exposed by the app. One should be able to explicitly justify why a specific permission was granted to a particular user or group rather than assuming access to be the default position.

- Although some frameworks or libraries may themselves adopt a deny-by-default strategy, explicit configuration should be preferred over relying on framework or library defaults. The logic and defaults of third-party code may evolve over time, without the developer's full knowledge or understanding of the change's implications for a particular project.

Validate the Permissions on Every Request¶

Permission should be validated correctly on every request, regardless of whether the request was initiated by an AJAX script, server-side, or any other source. The technology used to perform such checks should allow for global, application-wide configuration rather than needing to be applied individually to every method or class. Remember an attacker only needs to find one way in. Even if just a single access control check is "missed", the confidentiality and/or integrity of a resource can be jeopardized. Validating permissions correctly on just the majority of requests is insufficient. Specific technologies that can help developers in performing such consistent permission checks include the following:

- Java/Jakarta EE Filters including implementations in Spring Security
- Middleware in the Django Framework
- .NET Core Filters
- Middleware in the Laravel PHP Framework

Thoroughly Review the Authorization Logic of Chosen Tools and Technologies, Implementing Custom Logic if Necessary¶

Today's developers have access to vast amounts of libraries, platforms, and frameworks that allow them to incorporate robust, complex logic into their apps with minimal effort. However, these frameworks and libraries must not be viewed as a quick panacea for all development problems; developers have a duty to use such frameworks responsibly and wisely. Two general concerns relevant to framework/library selection as relevant to proper access control are misconfiguration/lack of configuration on the part of the developer and vulnerabilities within the components themselves (see A6 and A9 for general guidance on these topics).

Even in an otherwise securely developed application, vulnerabilities in third-party components can allow an attacker to bypass normal authorization controls. Such concerns need not be restricted to unproven or poorly maintained projects, but affect even the most robust and popular libraries and frameworks. Writing complex, secure software is hard. Even the most competent developers, working on high-quality libraries and frameworks, will make mistakes. Assume any third-party component you incorporate into an application *could* be or become subject to an authorization vulnerability. Important considerations include:

- Create, maintain, and follow processes for detecting and responding to vulnerable components.
- Incorporate tools such as Dependency Check into the SDLC and consider subscribing to data feeds from vendors, the NVD, or other relevant sources.
- Implement defense in depth. Do not depend on any single framework, library, technology, or control to be the sole thing enforcing proper access control.

Misconfiguration (or complete lack of configuration) is another major area in which the components developers build upon can lead to broken authorization. These components are typically intended to be relatively general purpose tools made to appeal to a wide audience. For all but the simplest use cases, these frameworks and libraries must be customized or supplemented with additional logic in order to meet the unique requirements of a particular app or environment. This consideration is especially important when security requirements, including authorization, are concerned. Notable configuration considerations for authorization include the following:

- Take time to thoroughly understand any technology you build authorization logic upon. Analyze the technology's capabilities with an understanding that *the authorization logic provided by the component may be insufficient for your application's specific security requirements*. Relying on prebuilt logic may be convenient, but this does not mean it is sufficient. Understand that custom authorization logic may well be necessary to meet an app's security requirements.
- Do not let the capabilities of any library, platform, or framework guide your authorization requirements. Rather, authorization requirements should be

decided first and then the third-party components may be analyzed in light of these requirements.

- Do not rely on default configurations.
- Test configuration. Do not just assume any configuration performed on a third-party component will work exactly as intended in your particular environment. Documentation can be misunderstood, vague, outdated, or simply inaccurate.

Prefer Attribute and Relationship Based Access Control over RBAC¶

In software engineering, two basic forms of access control are widely utilized: Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC). There is a third, more recent, model which has gained popularity: Relationship-Based Access Control (ReBAC). The decision between the models has significant implications for the entire SDLC and should be made as early as possible.

- RBAC is a model of access control in which access is granted or denied based upon the roles assigned to a user. Permissions are not directly assigned to an entity; rather, permissions are associated with a role and the entity inherits the permissions of any roles assigned to it. Generally, the relationship between roles and users can be many-to-many, and roles may be hierarchical in nature.
- ABAC may be defined as an access control model where "subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject, assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions" (NIST SP 800-162, pg. 7). As defined in NIST SP 800-162, attributes are simply characteristics that can be represented as name-value pairs and assigned to a subject, object, or the environment. Job role, time of day, project name, MAC address, and creation date are but a very small sampling of possible attributes that highlight the flexibility of ABAC implementations.
- ReBAC is an access control model that grants access based on the relationships between resources. For instance, allowing only the user who created a post to edit it. This is especially necessary in social network

applications, like Twitter or Facebook, where users want to limit access to their data (tweets or posts) to people they choose (friends, family, followers).

Although RBAC has a long history and remains popular among software developers today, ABAC and ReBAC should typically be preferred for application development. Their advantages over RBAC include:

- **Support fine-grained, complex Boolean logic**. In RBAC, access decisions are made on the presence or absence of roles; that is, the main characteristic of a requesting entity considered is the role(s) assigned to it. Such simplistic logic does a poor job of supporting object-level or horizontal access control decisions and those that require multiple factors.
    - ABAC greatly expands both the number and type of characteristics that can be considered. In ABAC, a "role" or job function can certainly be one attribute assigned to a subject, but it need not be considered in isolation (or at all if this characteristic is not relevant to the particular access requested). Furthermore, ABAC can incorporate environmental and other dynamic attributes, such as time of day, type of device used, and geographic location. Denying access to a sensitive resource outside of normal business hours or if a user has not recently completed mandatory training are just a couple of examples where ABAC could meet access control requirements that RBAC would struggle to fulfill. Thus, ABAC is more effective than RBAC in addressing the principle of least privileges.
    - ReBAC, since it supports assigning relationships between direct objects and direct users (and not just a role), allows for fine-grained permissions. Some systems also support algebraic operators like AND and NOT to express policies like "if this user has relationship X but not relationship Y with the object, then grant access".
- **Robustness**. In large projects or when numerous roles are present, it is easy to miss or improperly perform role checks (OWASP C7: Enforce Access Controls). This can result in both too much and too little access. This is especially true in RBAC implementations where a role hierarchy is not present and multiple role checks must be chained to have the desired impact (i.e. (`if(user.hasAnyRole("SUPERUSER", "ADMIN", "ACCT_MANAGER")`)).

- **Speed**. In RBAC, "role explosion" can occur when a system defines too many roles. If users send their credential and roles through means like HTTP headers, which have size limits, there may not be enough space to include all of the user's roles. A viable workaround to this problem is to only send the user ID, and then the application retrieves the user's roles, but this will increase the latency of every request.
- **Supports Multi-Tenancy and Cross-Organizational Requests**. RBAC is poorly suited for use cases where distinct organizations or customers will need access to the same set of protected resources. Meeting such requirement with RBAC would require highly cumbersome methods such as configuring rule sets for each customer in a multi-tenant environment or requiring pre-provisioning of identities for cross-organizational requests (OWASP C7; NIST SP 800-162). By contrast, as long as attributes are consistently defined, ABAC implementations allow access control decisions to be "executed and administered in the same or separate infrastructures, while maintaining appropriate levels of security" (NIST SP 800-162, pg. 6).
- **Ease of Management**. Although the initial setup for RBAC is often simpler than ABAC, this short-term benefit quickly vanishes as the scale and complexity of a system grows. In the beginning, a couple of simple roles, such as User and Admin, may suffice for some apps, but this is very unlikely to hold true for any length of time in production applications. As roles become more numerous, both testing and auditing, critical processes for establishing trust in one's codebase and logic, become more difficult (OWASP C7). By contrast, ABAC and ReBAC are far more expressive, incorporate attributes and Boolean logic that better reflects real-world concerns, are easier to update when access-control needs change, and encourages the separation of policy management from enforcement and provisioning of identities (NIST SP 800-162; see also XACML-V3.0 for a standard that highlights these benefits)

Ensure Lookup IDs are Not Accessible Even When Guessed or Cannot Be Tampered With¶

Applications often expose the internal object identifiers (such as an account number or Primary Key in a database) that are used to locate and reference an object. This ID may be exposed as a query parameter, path variable, "hidden" form field or elsewhere. For example:

`https://mybank.com/accountTransactions?acct_id=901`

Based on this URL, one could reasonably assume that the application will return a listing of transactions and that the transactions returned will be restricted to a particular account - the account indicated in the `acct_id` param. But what would happen if the user changed the value of the `acct_id` param to another value such as `523`. Will the user be able to view transactions associated with another account even if it does not belong to him? If not, will the failure simply be the result of the account "523" not existing/not being found or will it be due to a failed access control check? Although this example may be an oversimplification, it illustrates a very common security flaw in application development - CWE 639: Authorization Bypass Through User-Controlled Key. When exploited, this weakness can result in authorization bypasses, horizontal privilege escalation and, less commonly, vertical privilege escalation (see CWE-639). This type of vulnerability also represents a form of Insecure Direct Object Reference (IDOR). The following paragraphs will describe the weakness and possible mitigations.

In the example above, the lookup ID was not only exposed to the user and readily tampered with, but also appears to have been a fairly predictable, perhaps sequential, value. While one can use various techniques to mask or randomize these IDs and make them hard to guess, such an approach is generally not sufficient by itself. A user should not be able to access a resource they do not have permissions simply because they are able to guess and manipulate that object's identifier in a query param or elsewhere. Rather than relying on some form of security through obscurity, the focus should be on controlling access to the underlying objects and/or the identifiers themselves. Recommended mitigations for this weakness include the following:

- Avoid exposing identifiers to the user when possible. For example it should be possible to retrieve some objects, such as account details, based solely on currently authenticated user's identity and attributes (e.g. through information contained in a securely implemented JSON Web Token (JWT) or server-side session).
- Implement user/session specific indirect references using a tool such as OWASP ESAPI (see OWASP 2013 Top 10 - A4 Insecure Direct Object References)

- Perform access control checks on *every* request for the *specific* object or functionality being accessed. Just because a user has access to an object of a particular type does not mean they should have access to every object of that particular type.

Enforce Authorization Checks on Static Resources¶

The importance of securing static resources is often overlooked or at least overshadowed by other security concerns. Although securing databases and similar data stores often justly receive significant attention from security conscious teams, static resources must also be appropriately secured. Although unprotected static resources are certainly a problem for websites and web applications of all forms, in recent years, poorly secured resources in cloud storage offerings (such as Amazon S3 Buckets) have risen to prominence. When securing static resources, consider the following:

- Ensure that static resources are incorporated into access control policies. The type of protection required for static resources will necessarily be highly contextual. It may be perfectly acceptable for some static resources to be publicly accessible, while others should only be accessible when a highly restrictive set of user and environmental attributes are present. Understanding the type of data exposed in the specific resources under consideration is thus critical. Consider whether a formal Data Classification scheme should be established and incorporated into the application's access control logic (see here for an overview of data classification).
- Ensure any cloud based services used to store static resources are secured using the configuration options and tools provided by the vendor. Review the cloud provider's documentation (see guidance from AWS, Google Cloud and Azure for specific implementations details).
- When possible, protect static resources using the same access control logic and mechanisms that are used to secure other application resources and functionality.

Verify that Authorization Checks are Performed in the Right Location¶

Developers must never rely on client-side access control checks. While such checks may be permissible for improving the user experience, they should never be the decisive factor in granting or denying access to a resource; client-side logic is often easy to bypass. Access control checks must be performed server-side, at the gateway, or using serverless function (see OWASP ASVS 4.0.3, V1.4.1 and V4.1.1)

Exit Safely when Authorization Checks Fail¶

Failed access control checks are a normal occurrence in a secured application; consequently, developers must plan for such failures and handle them securely. Improper handling of such failures can lead to the application being left in an unpredictable state (CWE-280: Improper Handling of Insufficient Permissions or Privileges). Specific recommendations include the following:

- Ensure all exception and failed access control checks are handled no matter how unlikely they seem (OWASP Top Ten Proactive Controls C10: Handle all errors and exceptions). This does not mean that an application should always try to "correct" for a failed check; oftentimes a simple message or HTTP status code is all that is required.
- Centralize the logic for handling failed access control checks.
- Verify the handling of exception and authorization failures. Ensure that such failures, no matter how unlikely, do not put the software into an unstable state that could lead to authorization bypass.
- Ensure sensitive information, such as system logs or debugging output, is not exposed in error messages. Misconfigured error messages can increase the attack surface of your application. (CWE-209: Generation of Error Message Containing Sensitive Information)

Implement Appropriate Logging¶

Logging is one of the most important detective controls in application security; insufficient logging and monitoring is recognized as among the most critical security risks in OWASP's Top Ten 2021. Appropriate logs can not only detect malicious activity, but are also invaluable resources in post-incident investigations, can be used to troubleshoot access control and other security related problems, and are useful in security auditing. Though easy to overlook during the initial

design and requirements phase, logging is an important component of holistic application security and must be incorporated into all phases of the SDLC. Recommendations for logging include the following:

- Log using consistent, well-defined formats that can be readily parsed for analysis. According to OWASP Top Ten Proactive Controls C9, Apache Logging Services is one example of a project that provides support for numerous languages and platforms
- Carefully determine the amount of information to log. This should be determined according to the specific application environment and requirements. Both too much and too little logging may be considered security weaknesses (see CWE-778 and CWE-779). Too little logging can result in malicious activity going undetected and greatly reduce the effectiveness of post-incident analysis. Too much logging not only can strain resources and lead to excessive false positives, but may also result in sensitive data being needlessly logged.
- Ensure clocks and timezones are synchronized across systems. Accuracy is crucial in piecing together the sequence of an attack during and after incident response.
- Consider incorporating application logs into a centralized log server or SIEM.

Create Unit and Integration Test Cases for Authorization Logic¶

Unit and integration testing are essential for verifying that an application performs as expected and consistently across changes. Flaws in access control logic can be subtle, particularly when requirements are complex; however, even a small logical or configuration error in access control can result in severe consequences. Although not a substitution for a dedicated security test or penetration test (see OWASP WSTG 4.5 for an excellent guide on this topic as it relates to access control), automated unit and integration testing of access control logic can help reduce the number of security flaws that make it into production. These tests are good at catching the "low-hanging fruit" of security issues but not more sophisticated attack vectors (OWASP SAMM: Security Testing).

Unit and integration testing should aim to incorporate many of the concepts explored in this document. For example, is access being denied by default? Does

the application terminate safely when an access control check fails, even under abnormal conditions? Are ABAC policies being properly enforced? While simple unit and integration tests can never replace manual testing performed by a skilled hacker, they are an important tool for detecting and correcting security issues quickly and with far less resources than manual testing.