Exploiting broken access control vulnerabilities

BY AYOUB · MARCH 20, 2026 · LAST UPDATED ON MARCH 21, 2026

Broken access control vulnerabilities have consistently remained at the top of the OWASP Top 10, and for a good reason. As web applications continue to grow in complexity, with the introduction of role-based access controls, multi-tenant support, and granular permission models, the likelihood of access control flaws increases significantly.

Unlike other vulnerability classes that often rely on insufficient input sanitization (such as XSS, SSRF, and XXE), broken access control vulnerabilities stem from the flawed or missing enforcement of authorization rules, allowing attackers to access resources and perform actions beyond their intended privileges.

In this article, we'll explore the fundamentals of access control, the various ways it can be broken, and how you can identify and exploit these vulnerabilities effectively to gain unauthorized access to application components and sensitive data.

Let's dive in!

What is authorization

Authorization is the process that determines what a user (whether authenticated or not) is allowed to do within an application. Once a user has proven their identity (=authentication), the application needs to enforce boundaries on which resources they can access and what actions they can perform within an application or service (=authorization).

For instance, a regular user on an e-commerce platform should be able to view and manage their own orders, but shouldn't be able to access another customer's order history or modify product listings reserved for administrators.

In practice, authorization is typically enforced through a combination of server-side checks that verify whether the requesting user has the necessary permissions to perform a specific action on a resource. When these checks are missing, improperly implemented, or inconsistently applied across the application, broken access control vulnerabilities arise, often resulting in unauthorized data access, privilege escalation, or even full account takeover.

What is authentication

Authentication, on the other hand, is the process of verifying a user's identity before allowing them access to the application. In most web applications, authentication is commonly implemented through a login mechanism where users provide their credentials, such as a username and password, to prove their identity. Upon successful verification, the application creates a session or issues a token (such as a JWT) to track the authenticated user across subsequent requests.

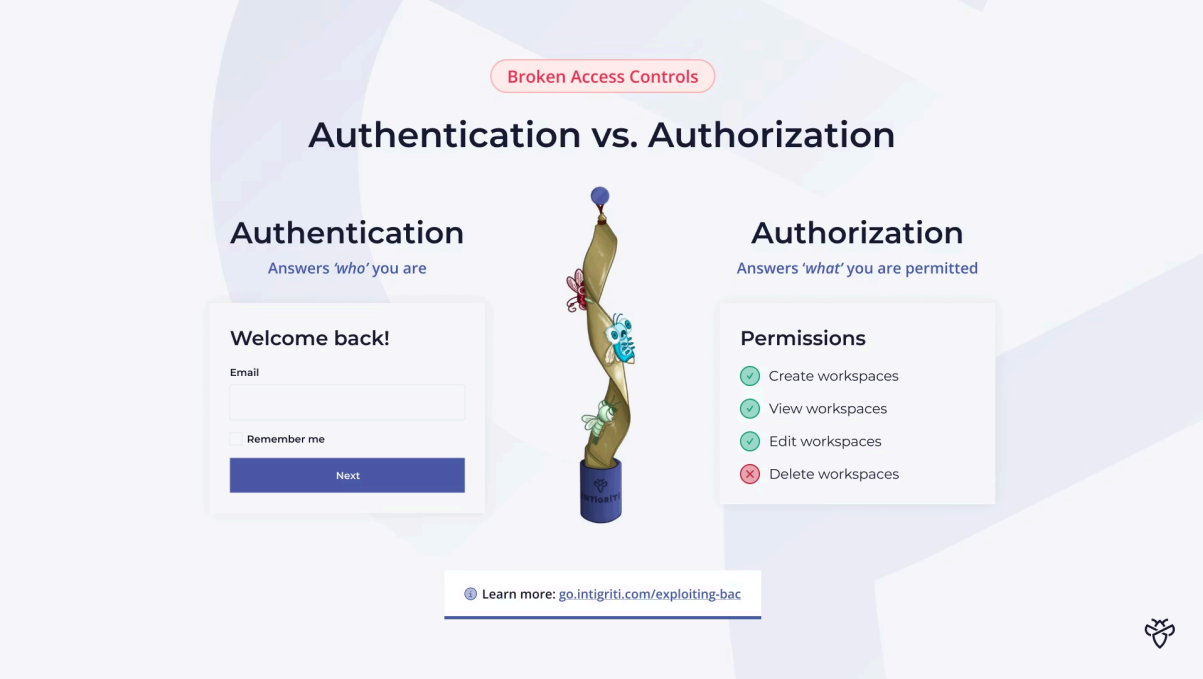
Exploiting broken authentication vulnerabilities

While authentication and authorization work closely together, it's important to understand that they serve fundamentally different purposes, as a flaw in one doesn't necessarily imply a flaw in the other.

Learn more about exploiting broken authentication vulnerabilities

Authentication vs. Authorization

As we've previously covered, authentication and authorization are two distinct security concepts that are often confused with each other. To put it simply, authentication is concerned with verifying who the user is, while authorization determines what that user is allowed to do.



Authentication vs. Authorization

Understanding this distinction is crucial when testing for access control vulnerabilities, as a fully functional authentication mechanism does not always imply that authorization is correctly enforced throughout the application or web service.

Common authorization implementations

Applications can implement authorization using different models. Understanding these models helps you identify missing or weak checks during testing. Let's cover the 3 most common types:

1. **Role-Based Access Control (RBAC)** is the most common authorization model you'll encounter. Users are assigned roles, such as admin, moderator, or regular user, and permissions are associated with those roles. For example, an admin role may grant access to user management features, while a regular user role restricts access to only their own profile and data.

2. **Attribute-Based Access Control (ABAC)** takes authorization a step further by basing access controls on attributes rather than predefined roles. These attributes can include user properties, resource

properties (such as the sensitivity or visibility of a resource), and even environment conditions (such as the time of day or the user's IP address). This model adds more flexibility, which also results in increased complexity.

3. **Discretionary Access Control (DAC)** enables resource owners to decide who can access their resources. The creator of a file or document controls who gets read, write, or delete permissions. This model is common in document and other types of resource-sharing platforms.

> **Tip!**
>
> It's worth noting that most bugs occur in RBAC and ABAC implementations where developers forget server-side access validation based on roles or attributes.
>
> Additionally, when RBAC and ABAC are combined to provide more granular access controls, it introduces increased complexity and the risk of flawed access controls.

What are broken or missing access control vulnerabilities

As we've learned in previous sections of this article, broken access controls stem from a lack of server-side validation checks. We've also learned that complexity is one of the major drivers of how broken access control vulnerabilities arise. As applications introduce more roles, permission levels, and granular access rules, the attack surface for possible access control flaws grows significantly.

Ultimately, access control flaws often result in allowing an unauthorized user to read or modify resources beyond their intended in-app permissions. However, it is worth noting that not all missing access control validations represent a security concern. To properly distinguish what can be considered an impactful broken access control issue from what's intended behavior, we must consider several factors. This is even more true when you're actively participating in bug bounty programs.

Let's have a look at a few examples.

Examples of impactful BAC/MAC flaws

Before reporting any type of security issue, it is always essential to demonstrate impact. The CIA triad can help us not only determine the impact but also the severity of our findings.

- **Accessing confidential data (PII).** Findings that result in accessing another user's data, such as PII in orders, messages, or billing data, or other types of in-app resources (e.g., account configuration, uploaded assets, etc.), are often treated as valid findings, as they directly impact the confidentiality (C) metric.

- **Introducing unauthorized changes.** Broken access control flaws that allow you to change or use resources or other users' data that benefits an attacker affect the integrity (I) metric. Findings of this nature are often treated as valid issues within most application contexts.

- **Ability to disrupt services.** Broken access control flaws that allow you to disrupt services, such as the ability to shut down production servers or permanently delete other people's resources within an application (resulting in denied access), are, in most cases, considered impactful as they directly affect the availability (A) metric.

Examples of non-impactful BAC/MAC flaws

On the other hand, some odd behavior may seem like an access control flaw, but actually isn't. This is often the case when, in no realistic scenario, a bad user can benefit from the identified issue. Let's have a look at a few examples.

- **Session swapping.** Swapping session tokens (e.g., cookies or authorization bearer tokens) between two accounts you control and observing that the application responds with the respective account's data is not a vulnerability, as this is expected behavior. Your target application is correctly identifying and serving data based on the authenticated session, which is exactly how session management is supposed to work.
- **Accessing public information through an API endpoint.** Accessing information that's already public (e.g., a web shop's catalog page) through an API endpoint that returns the same data but in a structured format (i.e., JSON or XML) cannot be considered a valid security flaw.
- **Theoretical vulnerabilities.** As with most cases, submissions that involve theoretical security issues with no realistic exploit scenario or attack surface, or issues that would require complex end-user interactions to be exploited, are usually unlikely to be considered actual security concerns. Think of IDORs that require multiple, non-guessable IDs for an attacker to exploit.

The list above was non-exhaustive, but the key is to always prove impact by triggering one of the CIA triad. Now that we understand what broken access control vulnerabilities are and what qualifies as an impactful finding, let's examine how to identify and test for these vulnerabilities effectively.

> **Impactful vs. non-impactful BAC flaws**
>
> To accurately distinguish impactful from non-impactful broken access control findings, you must at all times ask yourself whether the identified flaw allows an actual attacker to access, modify, or disrupt resources that are not intended for their privilege level.
>
> If a broken access control finding slightly violates the CIA triad and also requires a far-fetched or unlikely attack scenario, it may be worth noting the finding down for later reference. If it results in a direct impact, such as accessing another user's sensitive data or escalating to administrative privileges, it's best to gather the necessary evidence to prepare your report.

Identifying broken access controls

As with any other web-based vulnerability, the first step involves active enumeration. For broken access controls, you'll be required to enumerate the authorization model your target employs. This information will help us determine our testing approach and the number, including types of accounts, we'll need to set up.

For most targets, two regular user accounts are typically sufficient to test for broken access controls. However, if the application uses a role-based access control (RBAC) model, you'll want to create a test account for each available role. Additionally, if your target is a multi-tenant application, you may also want to test cross-tenant access controls.

Use email aliases!

Intigriti provides each researcher with an email alias. You can use this feature to create multiple test accounts without having to manage multiple mailboxes or rely on third-party email services.

Learn more

Next, it's time to map out all application routes and API endpoints by browsing through the application with each account. Pay close attention to any requests that reference user-controlled identifiers (such as user IDs, order IDs, or other types of identifiers), as these are common places where you should test for missing access controls.

Finally, it's time to test and try accessing resources that you shouldn't be able to. This includes data from other users as well as higher-privilege functionality. If the server does not properly deny access, you may have discovered a broken authorization flaw.

Testing for Broken Access Controls Methodology

Exploiting broken access control vulnerabilities

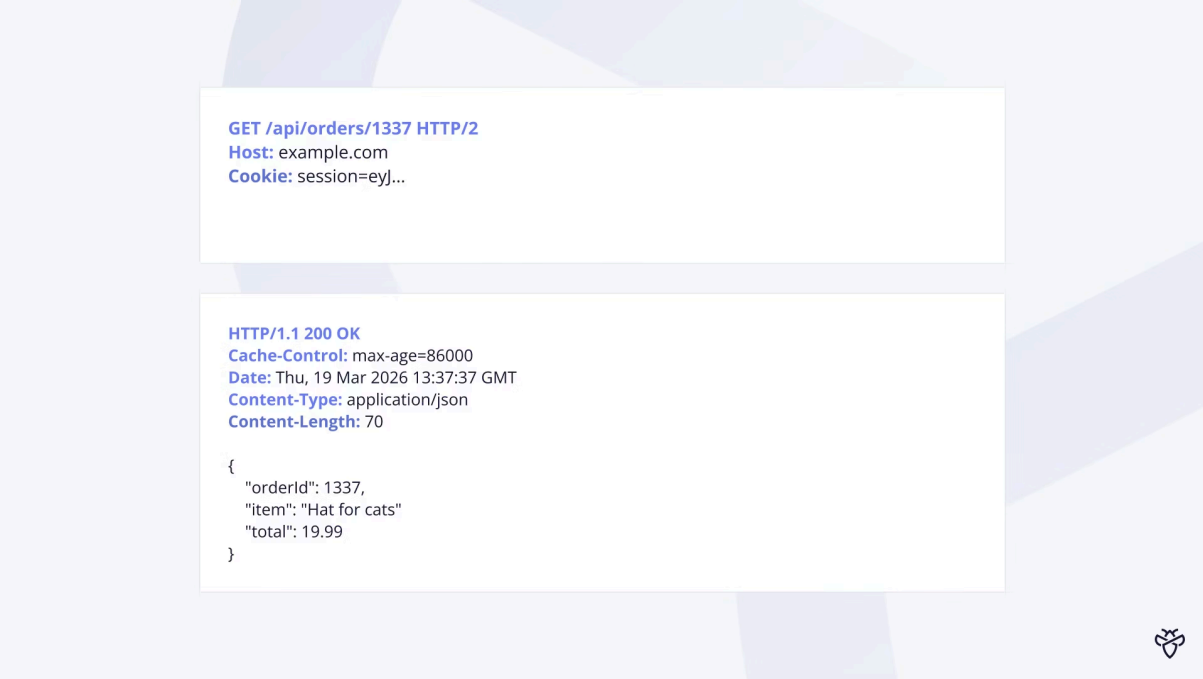
Broken access controls are, in most cases, much easier to exploit than to discover, merely because most vulnerabilities involve changing one identifier to another.

In this section, we'll dive into various exploitation scenarios, including some more advanced ones.

1. Exploiting simple broken access control flaws

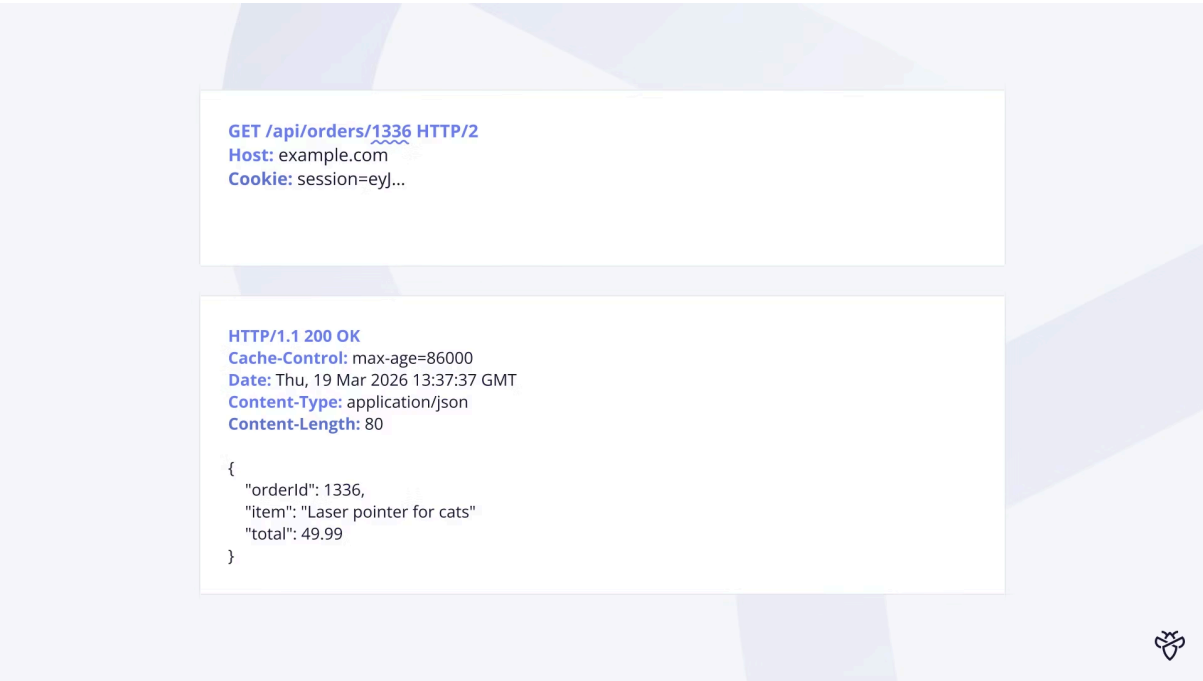
Insecure Direct Object References (IDORs) are the most straightforward type of broken access control vulnerability. They occur when an application uses user-controllable identifiers (such as user IDs, order IDs, or file names) to reference server-side objects without verifying whether the requesting user is authorized to access them. Let's take a look at a practical example.

Consider the following API endpoint that returns your order based on the order ID provided as a path parameter.



Exploiting simple broken access control (IDOR) vulnerabilities

Understanding how IDORs work, we can simply change the current order ID to any other order ID and retrieve another user's order details:



Exploiting simple broken access control (IDOR) vulnerabilities

Most cases of IDOR vulnerabilities that you will encounter involve restrictions that are only enforced on the client-side (for instance, by only displaying a user's own orders in the UI) but never validated on the

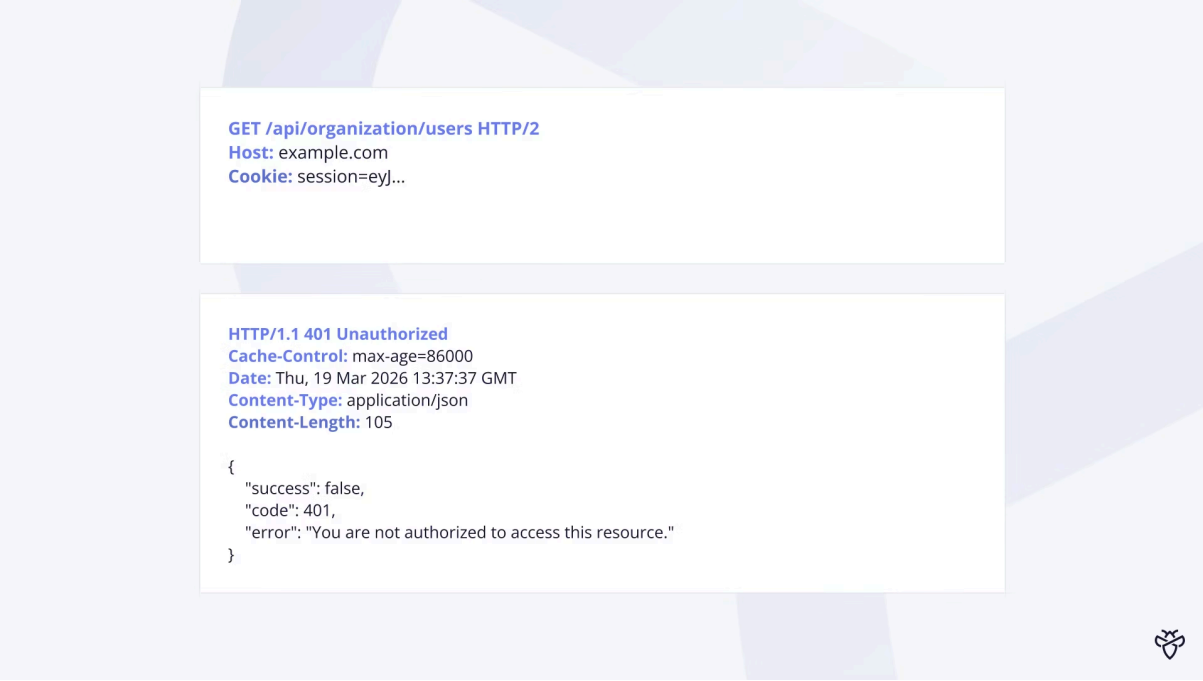
server-side. This means that by simply modifying the identifier in the request, an attacker can access or modify another user's resources.

Let's dive into the more advanced scenarios now.

2. Exploiting broken access control vulnerabilities via request method matching discrepancies

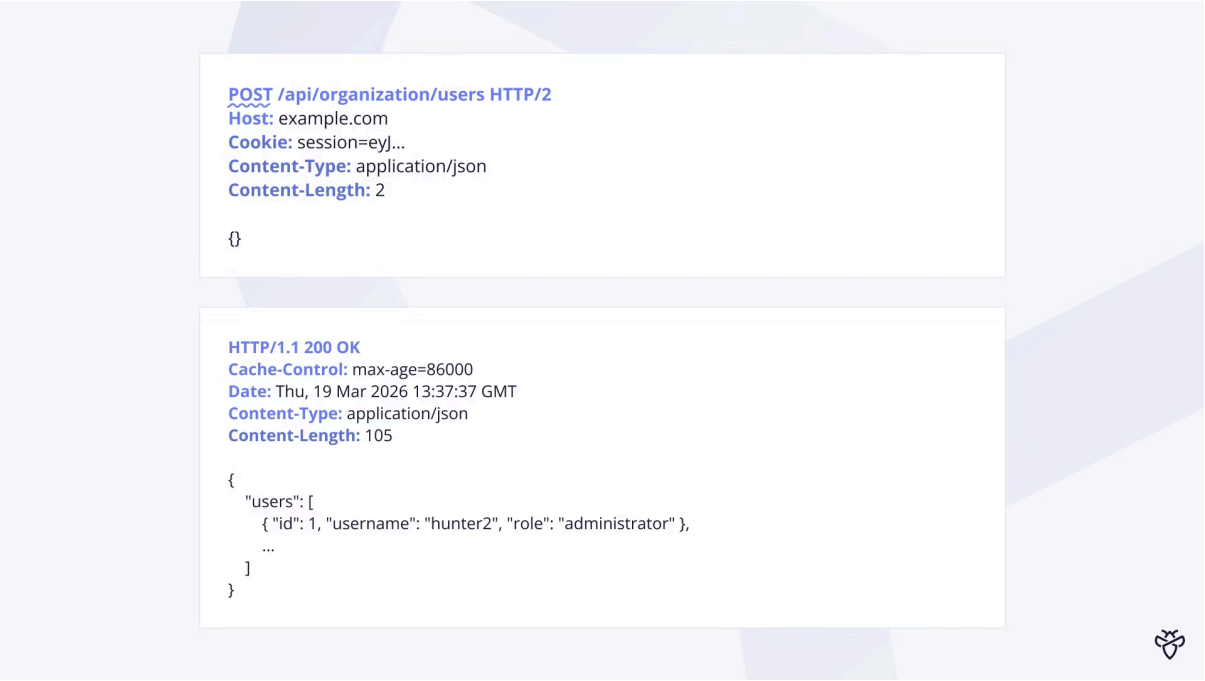
Another scenario you may observe is that access control checks are only enforced for specific HTTP methods. For instance, a `GET` request to an admin endpoint may correctly return a `403 Forbidden` response, but switching the request method to `POST` or `PUT` may bypass the access control check entirely.

Consider the following request where a regular user attempts to access the admin user management endpoint:



Exploiting broken access control vulnerabilities via request method matching discrepancies

As you can see, the server correctly validates access. However, by simply changing the request method from `GET` to `POST`, we can bypass the authorization check:



Exploiting broken access control vulnerabilities via request method matching discrepancies

This flaw commonly occurs when the application's routing or middleware only validates the user's access for a certain set of HTTP methods, while leaving others unvalidated. When you encounter a `401` or `403` response, always try alternative methods such as `POST`, `PUT`, `PATCH`, and even `DELETE` before moving on.

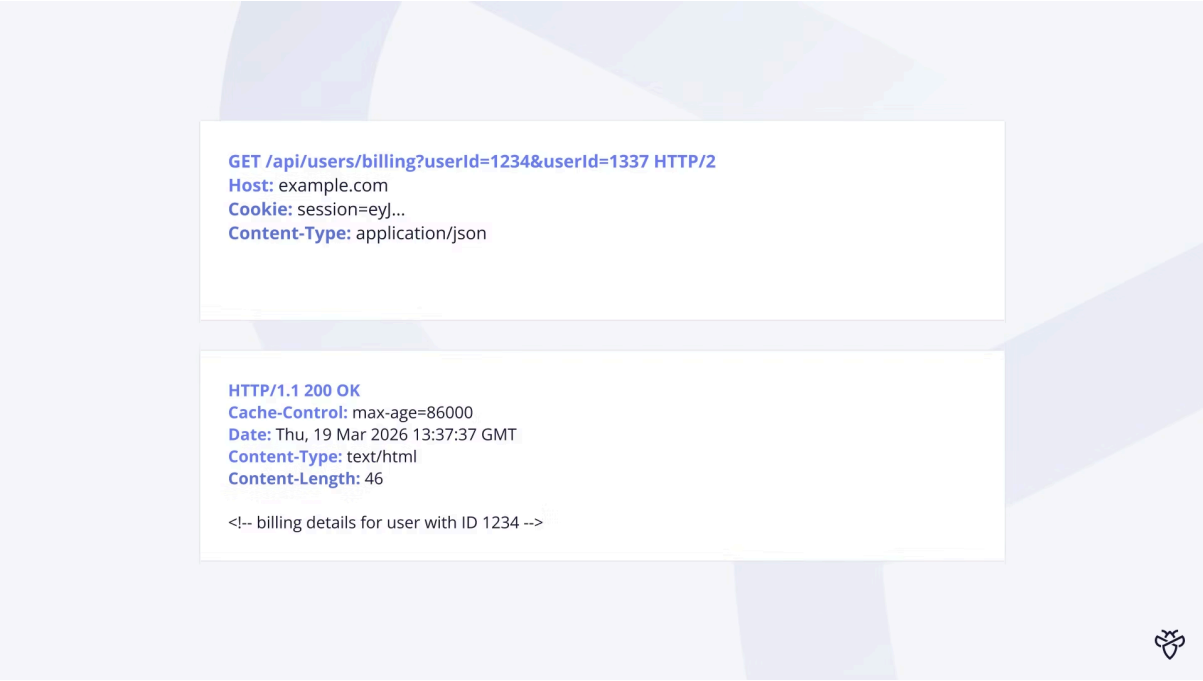
Fuzz content types!

Similarly, some APIs only respond to requests with a specific content type. If switching the HTTP method alone doesn't yield results, try fuzzing the Content-Type header as well, as some endpoints may only process requests sent with specific content types, such as `application/json`, `application/xml`, or `application/x-www-form-urlencoded`.

3. Exploiting broken access control vulnerabilities via HTTP parameter pollution

Another way to attempt to bypass authorization checks is by exploiting odd behavior in parameter parsing. This technique relies on inconsistencies between how authorization logic processes requests and how the application actually handles them.

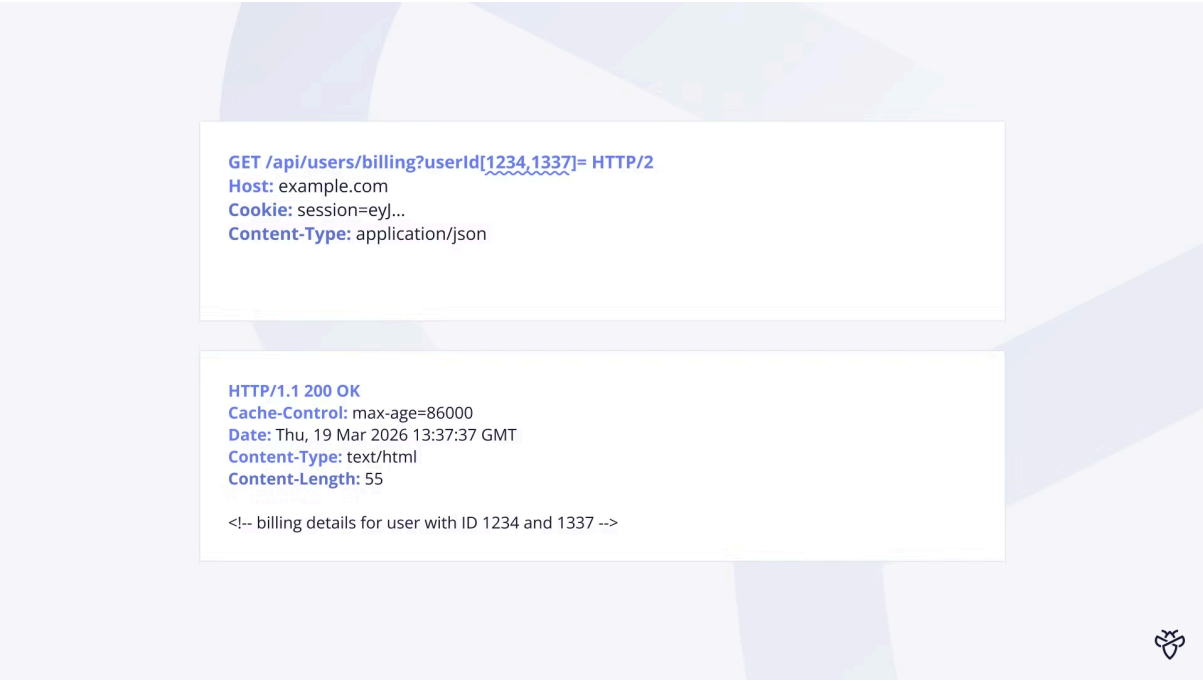
HTTP Parameter Pollution (HPP) occurs when you send the same parameter multiple times in a request. Various frameworks handle this differently, and this can lead to confusion. For instance, an in-app middleware that handles all authorization checks might validate one value, while the underlying endpoint uses the other. Consider the following example:



Exploiting broken access control vulnerabilities via HTTP parameter pollution

As you can see, the application only processed the first parameter while ignoring the second occurrence. It also returned our billing details, which we're permitted to view. This means that the application handled this case correctly. However, in other instances, the application logic may fail and either verify access using the first value while returning the billing details of the last supplied ID. Let's have a look at another example.

Suppose we sent the following request:



Exploiting broken access control vulnerabilities via HTTP parameter pollution

Three distinct scenarios could emerge. One where the application simply fails to handle parameter arrays and refuses your request. Practically, this may mean that testing for HPP might be ineffective in this

instance. Two, the application handles arrays but only validates and returns the first occurrence. Lastly, the application fails to properly handle this case and retrieves both billing details, returning all the matching records. You can further experiment with how an application may behave by fuzzing with malicious characters, such as:

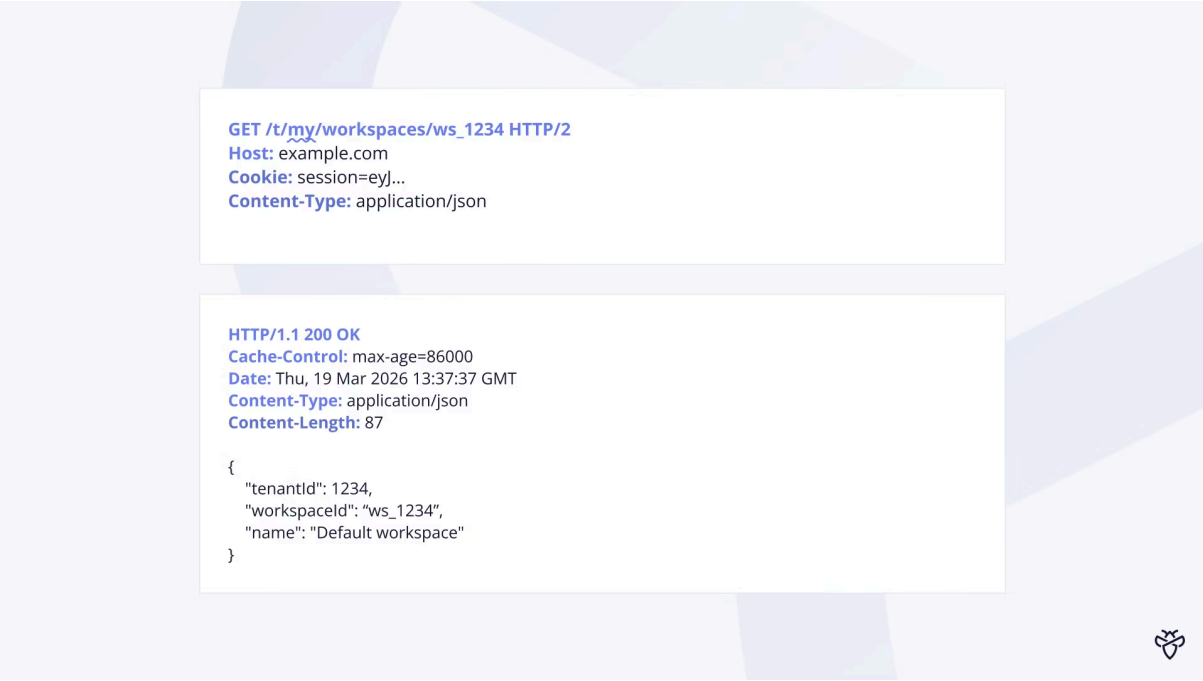
- **Adding comma characters.** Example: `/api/users/billing?userId=1234,1337`
- **URL encoding IDs.** Example: `/api/users/billing?userId=%31%33%33%37`
- **Injecting null byte characters.** Example: `/api/users/billing?userId=1337%00`
- **Injecting CR/LF characters.** Example: `/api/users/billing?userId=1337%0A`
- **Injecting a wildcard character.** Example: `/api/users/billing?userId=*` (be careful when applying this with any type of state-changing endpoint).
- **Injecting a boolean operator.** `Example: /api/users/billing?userId=true` (be careful when applying this with any type of state-changing endpoint).

All of the above could potentially break the application's logic and introduce behavior that could ultimately lead to a broken access control. Later in this article, we'll explore another advanced exploitation technique that relies solely on injecting malicious characters to bypass server-side validation.

4. Exploiting broken access control vulnerabilities in endpoints using static keyword references

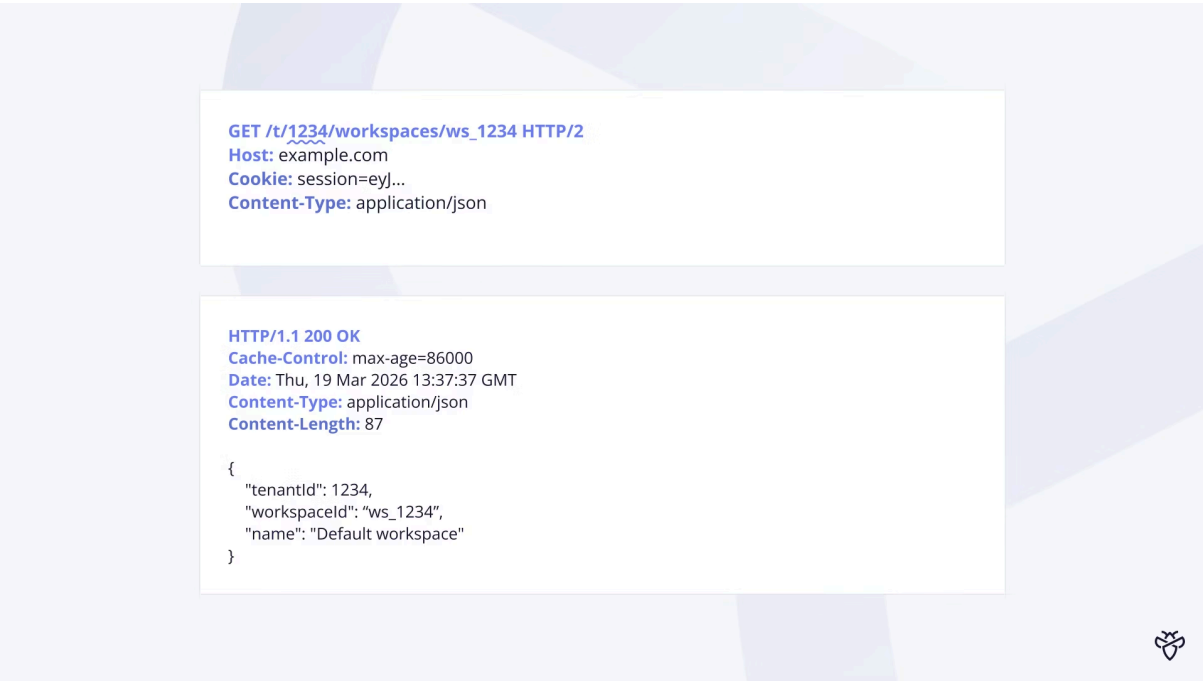
Some applications use keywords like `me`, `current`, or `my` as shorthand aliases to reference your current resource. While this may seem like a safe approach to avoid passing direct references, these keywords can sometimes be swapped with actual identifiers to access resources belonging to other users or tenants.

Consider a multi-tenant application that uses the following API endpoint to retrieve workspace details:



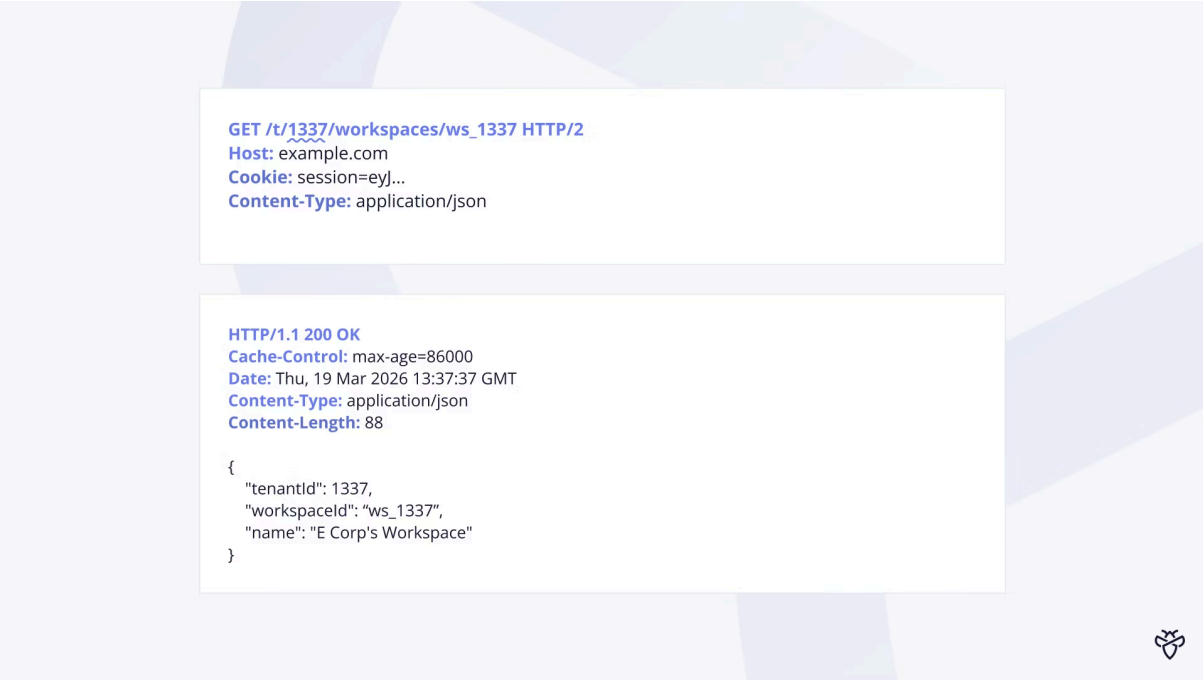
Exploiting broken access control vulnerabilities in endpoints using static keyword references

The `my` keyword resolves to the current user's tenant, possibly resolved on the server side. To verify this, we could attempt replacing the static `my` keyword with our tenant's identifier. In this instance, we can observe that the API also returns our own workspace data:



Exploiting broken access control vulnerabilities in endpoints using static keyword references

Understanding this, we can practically change the tenant ID with our victim's tenant ID and access other users' workspaces:



Exploiting broken access control vulnerabilities in endpoints using static keyword references

The scenario featured above illustrates a clear example of a cross-tenant broken access control vulnerability that directly impacts the confidentiality of another organization's data. Whenever you encounter alias keywords in API endpoints or request parameters, always try replacing them with actual identifiers to verify whether server-side authorization is properly enforced.

5. Exploiting broken access control vulnerabilities via JWT flaws

JSON Web Tokens are commonly used to handle authentication and maintain session state in modern web applications. Since JWTs carry authorization data in their payload, such as user roles, tenant IDs, and permission scopes, they can become a direct attack vector for broken access control vulnerabilities, especially when their claims can be tampered with.

Figure 1.1: Deconstructing a JSON Web Token (JWT)

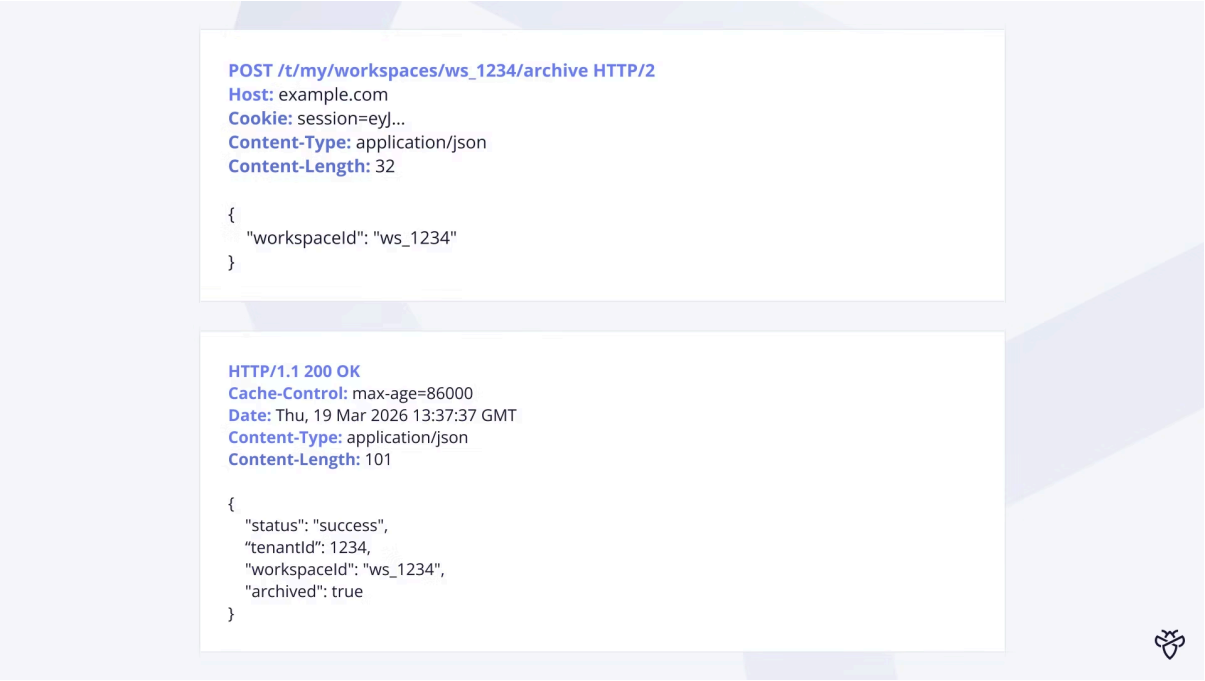
These issues arise when the application fails to properly validate the token's signature, uses weak signing secrets that can be brute-forced, or is susceptible to the `none` algorithm attack, where the target server accepts JWTs with no signature, indicating that any type of integrity verification is completely absent.

> **Learn more about JWT attacks!**
>
> Read our complete guide on exploiting JWT vulnerabilities if you wish to go through all common and advanced JWT exploitation techniques.

6. Exploiting broken access control vulnerabilities via logic flaws

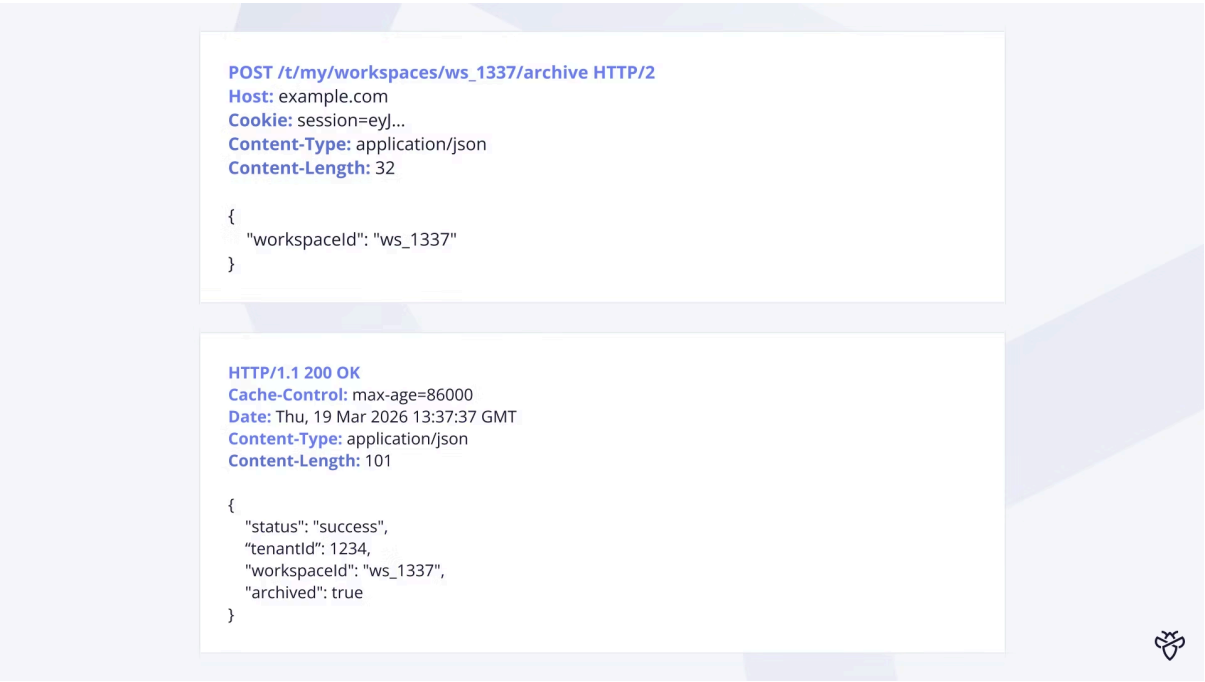
Broken access controls do not always stem from a single endpoint that lacks authorization controls. In some cases, they emerge from logic flaws within multi-step workflows or feature integrations where the application loses track of authorization between steps.

Let's take a look at an example. Consider the previous tenant-based application and suppose the application supports an archive feature to allow workspace administrators to archive workspaces they no longer need:



Exploiting broken access control vulnerabilities via logic flaws

By sending the request above, the application correctly archives our own workspace and places it under our archived section. However, if we decide to archive a workspace that belongs to another tenant user, the application logic may change the ownership to us, essentially giving us full read and write access to the workspace.



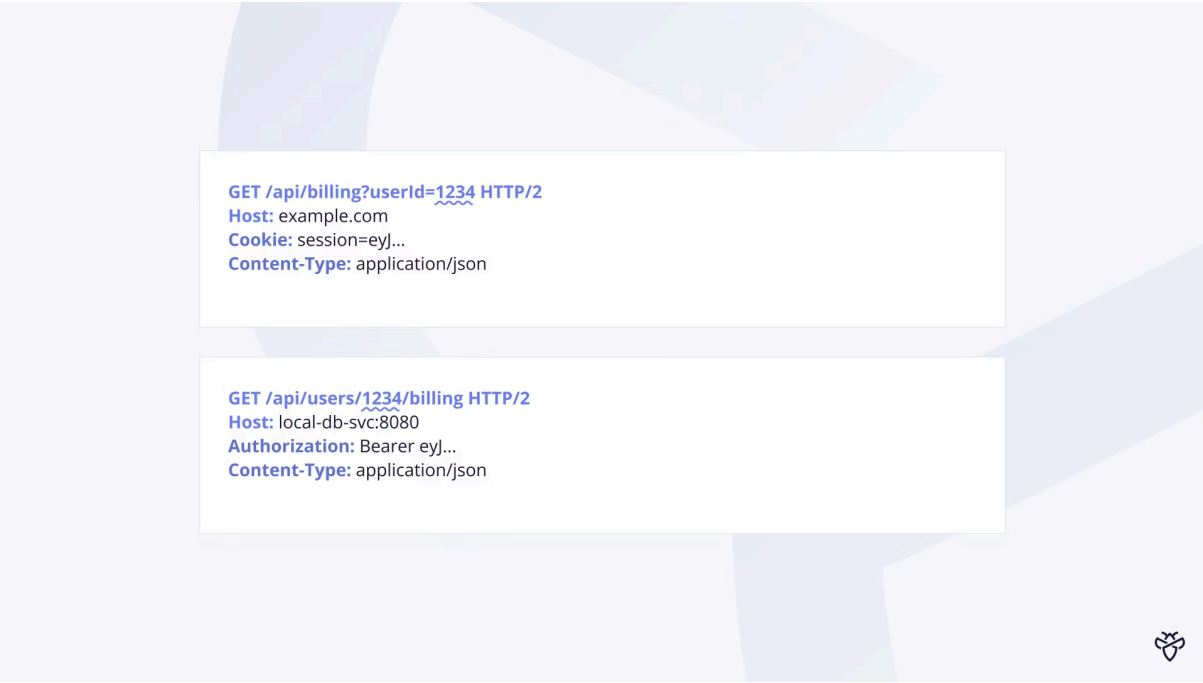
Exploiting broken access control vulnerabilities via logic flaws

The reasoning behind this is a subtle flaw that confused resource ownership. The application logic may have been configured to assume that the archiving is only done by the resource owner, but failed to validate that this is always the case. This is not the only scenario where logic flaws can lead to broken access control vulnerabilities. Let's have a look at a few more examples in the next section.

7. Exploiting broken access control vulnerabilities in second-order attacks

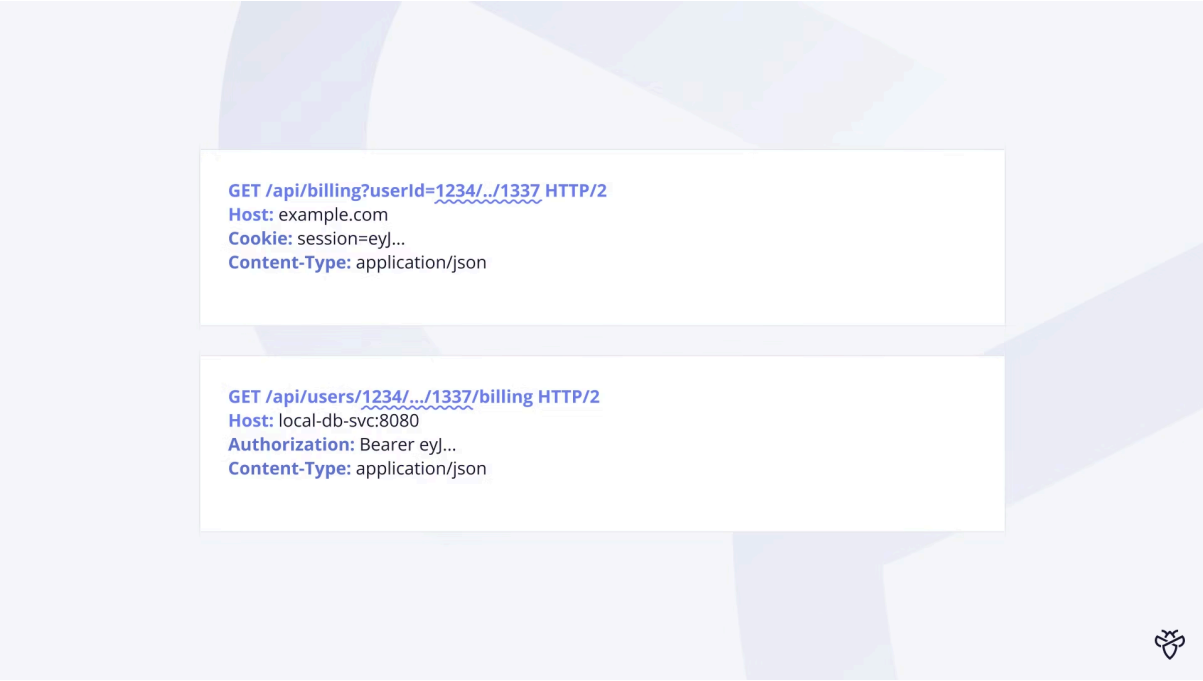
Second-order broken access control vulnerabilities occur when an application processes our input in a secondary context, such as an internal API call or a background process, where authorization checks are weaker or entirely absent. These are harder to spot because the initial request appears to be handled correctly, but the vulnerability arises in how the application forwards or reprocesses our data in other, secondary contexts.

Take the following target into consideration. Many modern applications use a middleware API that sits between the client and internal backend services. Sending a request to the API may be forwarded to an internal backend request. In this instance, the public API acts more as a middleware:



Exploiting broken access control vulnerabilities in second-order attacks

Notice what's happening here. The middleware authenticates to the internal service using a single, globally shared admin token rather than forwarding the original user's credentials. If the internal database API service doesn't perform any additional access control validations of its own, it will serve whatever the middleware asks for. By injecting a path traversal sequence into the **userId** parameter, the middleware or public API will again construct the following internal request:



Exploiting broken access control vulnerabilities in second-order attacks

This resolves to **/api/users/1337/billing**, returning the data of user 1337. To spot second-order vulnerabilities, a deeper understanding of your target is required. For instance, in our described scenario, the attack requires three conditions to be met:

1. The middleware should perform weak or no path normalization on user input.
2. The internal API must be accessed using a shared, global and privileged token instead of the user's own credentials.
3. Lastly, the internal service must not perform any access control validations of its own.

Let's examine another example of a second-order broken access control vulnerability.

Tip!

Don't limit yourself to testing only for IDORs. Second-order attacks can lead to various vulnerabilities, including injection attacks such as SQLi & SSRF!

Session poisoning

Similar to JWT attacks, session poisoning can be another form of second-order attack where one application feature overwrites session variables set by another. In server-side code, developers use session variables to store information about the authenticated user, such as the **session.userId**, **session.email**, and **session.isAuthenticated**.

These variables are set during login and referenced throughout the application to identify the current user. The issue arises when other features, such as a forgot password flow, reuse or overwrite these same session variables without forcing a logout.

Consider the following scenario. We log in to our account, and the server sets the **isAuthenticated** property within our session to **true**, while setting the email property to our account's email address.

While still signed in, we navigate to the forgot password page and enter another user's email in the reset form. If the application is flawed, it may temporarily set the email property within our session to the value we provided in order to look up the account.

Now, if we navigate back to our account page and refresh, the application may fail to revalidate our session cookie and load the account details with the victim's email. Practically, this means we've effectively taken over the victim's account without ever knowing the password.

This occurs because the developers reused the same session variable across two different features and didn't force a logout or revalidation when the password reset flow was initiated. The initial forgot password request appeared harmless, but the second-order effect of poisoning the session variable resulted in a complete authentication bypass.

Below is a similar example shared by @the_IDORminator, where he successfully poisoned his session cookie to load another profile's details, ultimately leading to a full account takeover.

Conclusion

Broken access control vulnerabilities are often straightforward to exploit, however, discovering them requires a thorough understanding of the target's authorization model and a methodical approach to testing. This is why reconnaissance is always essential. Mapping out all available roles, permissions, and endpoints before diving into exploitation will always yield better results than blindly fuzzing for broken access control vulnerabilities. Tooling such as Burp Suite's Autorize extension and Firefox Multi-Account Containers can significantly help simplify the testing process, especially when you're tasked with systematically identifying all authorization flaws across a feature-rich application.

So, you've just learned something new about exploiting broken access control vulnerabilities... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs and CTFs or... browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe earn a bounty on your next submission!

START HACKING ON INTIGRITI TODAY

AUTHOR

Ayoub

Senior security content developer