Broken Access Control and Injection: When the Backend Believes You

Wed, Sep 23, 10:00 AM EDT · https://scottslab.io/posts/injection-access-control-and-input-handling

> **TL;DR**
>
> Four common vulnerability classes, broken access control, SQL injection, buffer overflow, and directory traversal, share one root cause: the backend trusts something it received from the client instead of checking it. Fix that root cause with server-side validation on every input and an authorization check on every object, and most of the individual fixes below fall out for free.

The pattern underneath four different bugs

These four bugs look unrelated. One is a missing permission check, one is a database query, one is a memory bounds issue, one is a filesystem path. What they share is where the failure lives: the backend received something from the client and acted on it without verifying it first. Once you see that pattern, the four "different" vulnerabilities start reading like variations on the same mistake.

Broken access control: the backend knows who you are, not what you're allowed to see

Broken access control happens when the backend authenticates a request, confirms who you are, but doesn't authorize it, confirms you're allowed to touch the specific record you asked for. The classic case is an insecure direct object reference: you're logged into your own account, you notice the URL includes your account number, and you change one digit. If the backend serves whatever record matches the number in the URL without checking that the number belongs to the session making the request, you're looking at somebody else's data.

The fix isn't "add a login page." Most apps with this bug already have one. The fix is checking ownership on every request that touches a specific object, not just once at the front door. A session being valid tells you who's asking. It says nothing about what they're allowed to ask for.

SQL injection: the backend can't tell code from data

SQL injection happens when user input gets concatenated straight into a query string, so a value the backend expected to be data becomes part of the query's logic. A single quote in an input field is enough to escape out of the string the developer intended and start writing SQL of your own. In a demo I ran through, a single-quote escape in a form field let a second command ride along and change a password on an account that wasn't mine, using nothing but the input box.

The defense is parameterized queries: bind the input as a value, never splice it into the query text, and the database engine itself refuses to interpret it as SQL no matter what characters it contains. Stored procedures done correctly get you the same protection, and the reason is worth stating plainly: a procedure that accepts an argument, say a lookup that takes a state and returns every customer in it, plugs that argument into SQL that was already compiled ahead of time. The statement itself is fixed before any input ever arrives, so once it's precompiled there's no query text left for an attacker's input to rewrite. Server-side input validation matters too, but it's a second layer, not the fix. Client-side validation isn't a layer at all: it's a user-experience feature that catches an honest typo, and it stops mattering to an attacker the moment they disable JavaScript in the browser or point a proxy tool straight at the request and skip the browser's check entirely.

Buffer overflow: the backend didn't check how much it was given

A buffer overflow happens when input exceeds the memory the program reserved for it, and the excess spills into adjacent memory instead of being rejected. What that adjacent memory holds is not the attacker's business, which is exactly the problem. In one demo, a hotel booking form's room number field had no length limit. Feeding it a 4,097-digit number overran the buffer and dumped every guest record in the system back to the requester, not just the one room they'd asked about.

The defense is strict, server-side length and type validation before the value ever reaches whatever's allocating memory for it. That's the direct fix. The deeper architectural fix is choosing memory-safe languages and bounds-checked libraries where the option exists, so a missed validation check doesn't turn into an out-of-bounds write by default.

Directory traversal: the backend trusted a path

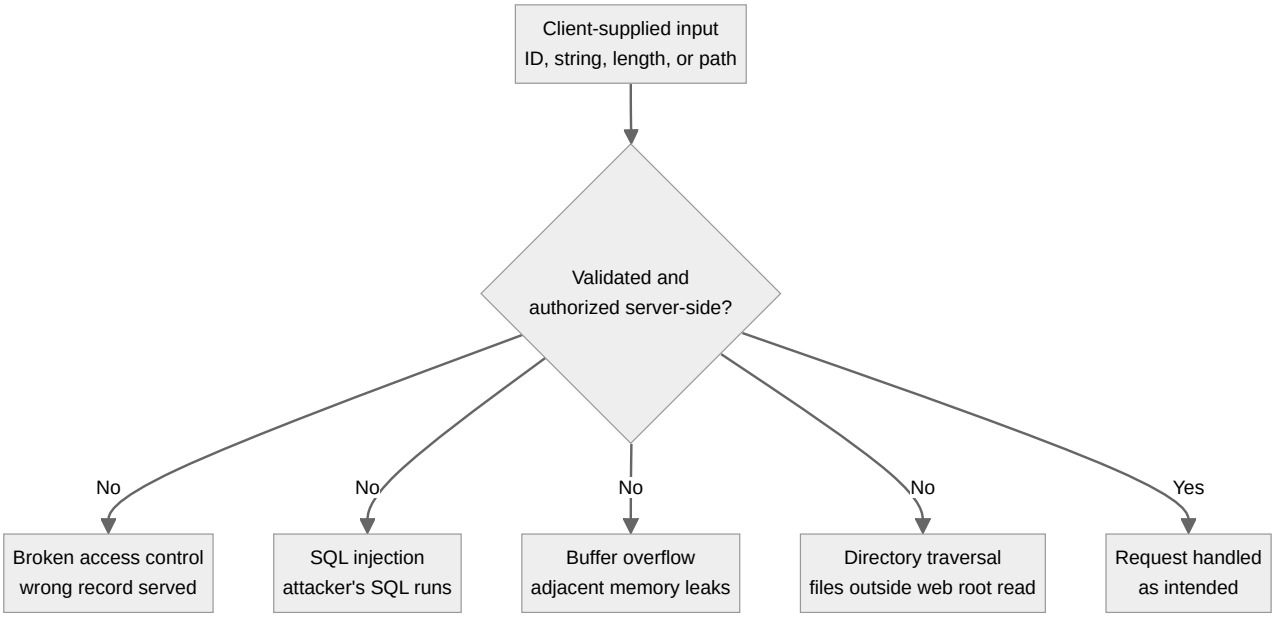
Directory traversal uses sequences like ../ inside a filename or path parameter to walk back out of the directory an application is supposed to be confined to. In a WebGoat exercise run through OWASP ZAP, four levels of ../ in a file parameter were enough to escape the web root and read tomcat-users.xml straight off the server, a file that can hold credentials for Tomcat's own admin console, which makes it a genuinely dangerous file to expose rather than just an embarrassing one.

The defense has two parts. Reject periods and path separators in any user-supplied filename or path input before it's used, rather than trying to sanitize your way around every encoding of ../ an attacker might try. And set strict filesystem permissions for the account the web server runs as, so that even a validation check that gets bypassed doesn't hand over files the process was never permitted to read in the first place.

The fix is the same shape every time

Server-side validation, parameterized handling of anything that becomes a query or a path, and an authorization check scoped to the specific object rather than the endpoint. Every one of the four bugs above is a version of the backend skipping one of those and taking the client's word for it instead.

That server-side validation splits into two formal approaches, and picking between them matters. Whitelisting defines what's allowed, an exact set of accepted characters, formats, or values, and rejects everything outside that spec. Blacklisting defines what's disallowed instead and lets everything else through. Blacklisting looks more flexible on day one, since you don't have to enumerate every legitimate input up front, but that flexibility is also its failure mode: an exhaustive blacklist has to anticipate every way an attacker might encode, case-shift, or otherwise vary the same underlying attack, and missing one variant is enough to defeat the whole check. Whitelisting is the more effective approach for exactly that reason. It doesn't need to predict what an attack looks like, only what a legitimate input looks like, which is a far smaller and more stable target to define.



The fix is the same shape every time

Believing less

None of these four defenses are exotic. They're all a version of "check it yourself, on the server, every time," which is unglamorous enough that it gets skipped under deadline pressure more often than it should. The backend that believes what the client sends it isn't insecure because nobody thought about security. It's insecure because someone assumed the check at the front door was enough, and the front door was never the part these bugs walk through.