# Insecure Design

**Insecure Design** refers to **missing or weak security controls** resulting from **flawed application design**, not from implementation bugs or misconfigurations. It focuses on **security logic, business workflows, trust assumptions, and design patterns** — the foundation of how an app works.

# Key difference from implementation bugs:

This is a **design flaw**, not a coding error. Fixing requires **re-architecting or redesigning**, not just patching.

# Core Characteristics of Insecure Design

| Area | Example |
| --- | --- |
| Missing threat modeling | Application never assessed for abuse cases |
| Trust-by-default | Trusting user input or third-party data without checks |
| No rate limiting | Allowing unlimited login attempts or API calls |
| Reuse of vulnerable workflows | Repeating flawed patterns in multiple modules |
| Business logic abuse | Allowing cancellation of other users’ orders by manipulating IDs |
| Hardcoded workflows | Impossible to apply access control due to rigid flows |
| Unsafe defaults | Feature exposed by default (debug, test, verbose APIs) |

# Impact

| Impact Type | Description |
| --- | --- |
| Authentication/Access Bypass | No access control logic or weak rules |
| Business Logic Exploitation | Abusing design to cancel orders, transfer funds, etc. |
| Privilege Escalation | No checks on role transitions or elevated access |
| Data Leakage | Export/download flows missing security logic |
| Abuse at Scale | No throttling = automated attacks (e.g., mass reset, brute force) |
| Denial of Trust | Weak verification in workflows (e.g., password reset via guessable link) |

# Examples of Insecure Design

| Scenario | Design Flaw |
| --- | --- |
| API lacks rate limiting | Allows brute force login attempts |
| App allows role=admin in POST | No design check on role escalation |
| Password reset flow works without email validation | No ownership verification |
| E-commerce: order\_id in URL can be modified | No object-level access control |
| Logout doesn’t invalidate session token | No session lifecycle design |
| Verbose error messages show stack traces | Debug mode exposed in design |

# Difference: Insecure Design vs. Insecure Implementation

| Insecure Design | Insecure Implementation |
| --- | --- |
| No access control defined | Access control logic present but misconfigured |
| No rate limiting concept | Rate limiting exists but broken in code |
| Password reset via GET only | Poor implementation of token logic |
| Logic lets user delete any object | Broken function-level auth implementation |

# Testing Insecure Design

| Method | Description |
| --- | --- |
| Threat Modeling | Identify where attacker can misuse flows |
| Business Logic Abuse | Attempt to cancel other user’s order, change roles |
| Workflow Tampering | Modify JSON/URL parameters to test privilege escalation |
| Disable JavaScript / Break Client-Side Controls | See if security is enforced server-side |
| Analyze Error Messages | Look for unhandled cases, debug info |
| Bypass Security Steps | Test what happens if steps are skipped (e.g., skip email verification) |

# Mitigation

## 1. Secure by Design Principles

* Security requirements documented during planning
* Build security into **user stories and acceptance criteria**
* Use **secure architectural patterns** (Zero Trust, Least Privilege)

## 2. Threat Modeling

* Use **STRIDE**, **PASTA**, or **LINDDUN**
* Analyze **abuse cases, misuses**, and **trust boundaries**
* Revisit models during feature additions

## 3. Business Logic Controls

* Validate every state transition (e.g., order → cancel → refund)
* Enforce access control **by role and object**
* Never rely only on the frontend for critical workflows

## 4. Workflow Resilience

* Implement **rate limiting, CAPTCHA**, and **fail-safe defaults**
* Require **step-wise validation** (e.g., confirm email → verify token → reset password)
* Secure **session lifecycle**: login, refresh, revoke, logout

## 5. Developer Training

* Train teams to **design for abuse prevention**
* Review with architects before development
* Integrate secure design reviews into the SDLC

# Real-World Examples

| Org | Design Flaw |
| --- | --- |
| GitHub (2022) | Logic allowed repo takeover via renamed usernames |
| Tesla (bug bounty) | No check on VIN → allowed command injection into any car |
| Facebook | IDOR + poor design allowed mass access to private content |
| PayPal | Flawed "guest checkout" logic allowed bypassing fraud checks |

# Points

“Insecure Design is **not a coding mistake**, it's a **flawed architectural decision** — the app was built insecure from the ground up.”

“Assess insecure design by **thinking like an attacker** during **feature planning**, not just pen testing after release.”

“Controls like **MFA, rate-limiting, secure session flows, and access control** should be **designed-in**, not bolted on.”