# JSON Injection

**JSON Injection** is a vulnerability that arises when **user-controlled input is inserted into a JSON structure without proper sanitization or escaping**, allowing the attacker to:

* Inject **arbitrary JSON content**
* Modify API request/response behavior
* Exploit **client-side code** using JavaScript parsers
* Chain into **XSS**, **logic manipulation**, or **injection attacks**

# Where JSON Injection Happens

| Context | Example |
| --- | --- |
| JSON in HTTP Request Body | API endpoint receives JSON data |
| JSON in HTTP Response | Client-side app parses and displays JSON |
| JSON Serialization/Deserialization | Backend encodes/decodes user input |
| JSON used in JavaScript | JSON is passed to eval(), JSON.parse(), or DOM directly |

# Impact

| Area | Description |
| --- | --- |
| Data Tampering | Modify or override request structure |
| Privilege Escalation | Inject higher roles in serialized JSON |
| Stored XSS | Inject JS/HTML inside JSON interpreted by frontend |
| Deserialization Attacks | Craft JSON objects that trigger insecure deserialization |
| Prototype Pollution | Inject keys like \_\_proto\_\_ or constructor |
| SQL/Command Injection Chain | Injected JSON passed to SQL or command without sanitation |

# Example

## 1: API Input Injection

Vulnerable Request:  
{

"username": "attacker",

"role": "user"  
  
Attacker Payload:  
{

"username": "attacker",

"role": "admin"

}  
If backend trusts JSON input blindly → attacker gets elevated privileges.

## 2: Injection in JSON Response

Server Response: { "message": "Welcome back, Abhinav!" }  
If AB is attacker-controlled, payload: { "message": "Welcome back, </script><script>alert(1)</script>" }  
**Reflected in HTML** → XSS via JSON injection.

# Advanced Exploits

## Prototype Pollution (Client-side JavaScript)

Payload:

{

"\_\_proto\_\_": {

"isAdmin": true

}

}

If app merges this into a JS object without validation, attacker can:

* Override object behavior
* Escalate privileges
* Manipulate frontend logic

## Command/SQL Injection via JSON

If backend parses: { "cmd": "ls" }

Attacker injects: { "cmd": "ls; rm -rf /" }

If used in shell execution → leads to RCE.

# Detection Techniques

| Method | Description |
| --- | --- |
| Manual Fuzzing | Modify JSON keys/values in requests and responses |
| Reflection Testing | Inject special chars (", <, >, /, {}) into fields |
| Prototype Payloads | Try \_\_proto\_\_, constructor, toString in objects |
| Intercept JSON | Use Burp Suite, Postman to tamper with body |
| Observe Behavior | Do unexpected keys cause logic changes? |

# Tools

| Tool | Purpose |
| --- | --- |
| Burp Suite | Intercept and modify JSON payloads |
| Postman | Manual API testing with crafted JSON |
| ZAP | Passive scanning of JSON endpoints |
| JSON-Bomb / Prototype Pollution Tools | Test object pollution and logic bypass |
| jq | Parse and analyze JSON response structure in CLI |

# Mitigation

## 1. Sanitize and Validate All Inputs

* Only accept expected **data types and structures**
* Block unexpected keys like \_\_proto\_\_, constructor, etc.

## 2. Escape JSON Output

* Properly encode user data before inserting into JSON:
  + Escape ", \, <, >, /
* Never concatenate strings directly into JSON

## 3. Use Safe Parsers and Libraries

* Use JSON.parse() and **avoid eval()** in JavaScript
* In Node.js, use Object.create(null) to prevent prototype pollution

## 4. Implement Whitelisting

* Whitelist allowable keys, values, object structures
* Reject JSON with unknown or extra fields

## 5. Secure Deserialization

* Avoid insecure use of jsonpickle, GSON, or unsafe unpickling
* Limit which classes/types can be restored

# Points

“JSON Injection is often overlooked because JSON is **data**, not code — but when combined with JavaScript or unsafe deserialization, it can be deadly.”

“Test for JSON Injection by **tampering with keys, injecting nested structures, and probing for response reflection or desync**.”

“Mitigation is mostly about **strict schema validation**, **encoding**, and **never trusting client input even in APIs**.”

# Real-World Cases

| App | Vulnerability |
| --- | --- |
| Shopify (Bug Bounty) | JSON input allowed unauthorized parameter injection |
| PayPal | Prototype pollution via JSON API led to client-side control bypass |
| Node.js Modules | Multiple CVEs in popular libs from unsafe merges of JSON |