Medium Search

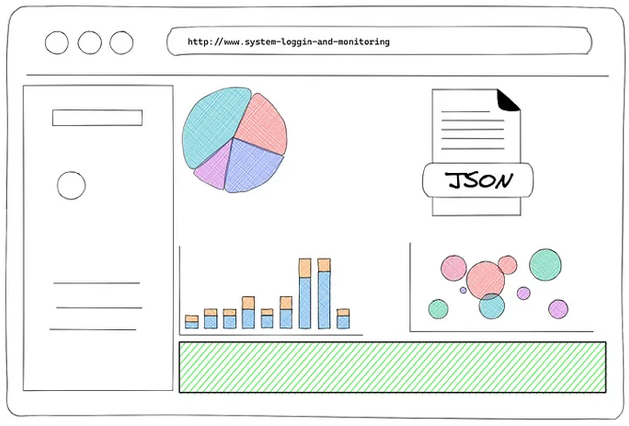
To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

Write Sign up Sign in

10 API Security Vulnerabilities You Need To Be Aware Of (Along with REST API Overview)

Santosh Shinde · Follow
Published in JavaScript in Plain English · 11 min read · May 15, 2021

85 1



REST API overview & API Security Vulnerabilities

What is an API?

An API (or Application Programming Interface) provides a method of interaction between two systems.

What is a RESTful API?

REST stands for REpresentational State Transfer, first defined in 2000 by computer scientist Dr. Roy Fielding.

REST is a set of architectural constraints, not a protocol or a standard. It is an architecture style for designing loosely coupled applications over HTTP, that is often used in the development of web services.

REST does not enforce any rule regarding how it should be implemented at a lower level, it just puts high-level design guidelines and leaves you to think of your own implementation.

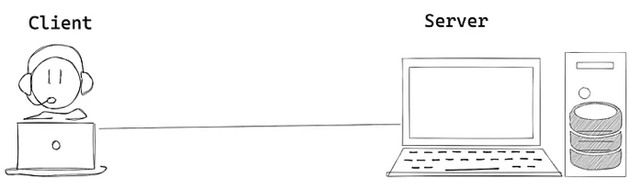
REST Design Principles or Architectural Constraints

1. Client-Server
2. Stateless
3. Cache
4. Uniform Interface
5. Layered System
6. Code-On-Demand (optional)

Client-Server

The principle behind this constraint is the separation of concern as the user interface is totally segregated from concerns regarding data storage.

Essentially it means client and server applications evolve without dependency on each other.



Client-Server

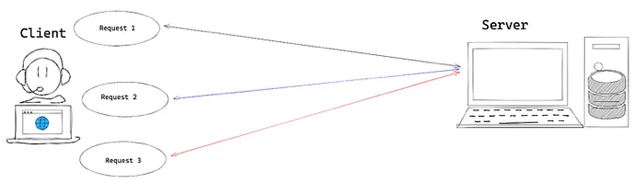
Stateless

In REST applications, each request must contain all of the information necessary to be understood by the server, rather than be dependent on the server remembering prior requests.

Storing the session state on the server violates the stateless constraint of the REST architecture. So the session state must be handled entirely by the client.

This constraint induces the properties of visibility, reliability, and scalability

- **Visibility** — Every request contains all context necessary to understand it. Therefore, looking at a single request is sufficient to visualize the interaction.
- **Reliability** — Since a request stands on its own, failure of one request does not influence others.
- **Scalability** — The server does not have to remember the application state, enabling it to serve more requests in a shorter amount of time.



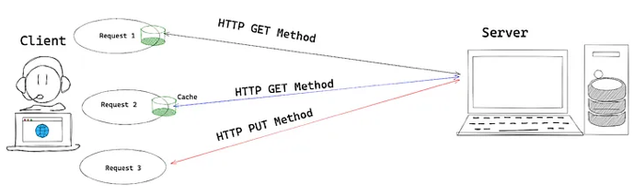
Stateless

Cache

Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

Caching is the ability to store copies of frequently accessed data in several places along the request-response path.

We can use **Expires** and **Cache-Control** HTTP response headers to control caching behavior



cache

Uniform Interface

The uniform interface constraint is fundamental to the design of any REST service. The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently.

The four guiding principles of this interface are:

- **Identification of resources** — You use the URI (IRI) standard to identify a resource. In this case, a resource is a web document.
- **Manipulation of resources through these representations** — You use the HTTP standard to describe communication. So for example GET means that you want to retrieve data about the URI-identified resource. You can describe an operation with an HTTP method and a URI.
- **Self-descriptive messages** — You use standard MIME types and (standard) RDF vocabs to make messages self-descriptive. So the client can find the data by checking the semantics, and it doesn't have to know the application-specific data structure the service uses.
- **Hypermedia as the engine of application state** — You use hyperlinks and possibly URI templates to decouple the client from the application-specific URI structure. You can annotate these hyperlinks with semantics e.g. IANA link relations, so the client will understand what they mean.

Uniform Interface

Layered System

In order to improve behavior for Internet-scale requirements layer system constraints come into the picture.

The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.

REST allows you to use a layered system architecture where you deploy the APIs on server 1, and store data on server 2 and authenticate requests in Server 3.

Layered System

Code-On-Demand

REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts.

This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility.

However, it also reduces visibility and thus is only an optional constraint within REST.

Code-On-Demand

. . .

API Security Vulnerabilities (OWASPS-Top-10)

1. **Broken Object Level Authorization**

APIs tend to expose endpoints that handle object identifiers, creating a wide attack surface Level Access Control issue.

The lack of proper authorization checks allows attackers to access the specified resource.

Broken Object Level Authorization

Use Cases

1.Genuine request from an authenticated user to get the user information

Case 1 - Genuine request from an authenticated user

2. User tries to access the resources of another user by guessing the right parameter

case 2 — guess right parameter

3. User tries to access resources of another user by guessing the wrong parameter

Case 3 — guess wrong parameters

> *Broken Object Level Authorization can only be tested manually. You can use the same tools with which you usually test APIs like Postman, Fiddler, ReadyAPI.*

. . .

2. **Broken User Authentication**

Broken user authentication flaws happen mainly due to loopholes in session management and credential management.

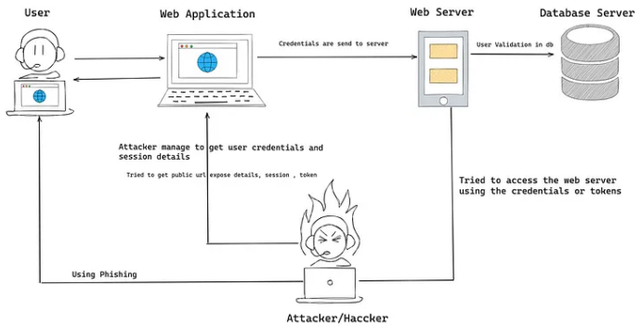
Attackers mainly take the advantage of the weakness of the authentication mechanism to get the user session id, details, and user credentials.

In this vulnerability, there are so many cases due to not implementing best practices while developing and deploying API.

Broken Authentication attacks aim to take over one or more accounts giving the attacker the same privileges as the attacked user. Authentication is broken when attackers are able to compromise passwords, keys or session tokens, user account information, and other details to assume user identities.

Use Cases

1. Username and password are easy to guess using fuzzing or brute force.
2. Password Does not match Password complexity policy or best practices.
3. Enumeration of username/password at the authentication failure response invalid username or an invalid password.
4. Login credentials are not protected when stored and lacking hashing and salt.
5. Transmission of username and password over an unencrypted channel such as HTTP
6. Session ID exposes in URL.
7. User session or authentication tokens are not timeouts after user logout.



Broken User Authentication

> *Developers must ensure authentication mechanisms are correctly set and secured. Several automated tools may help you test the most common authentication patterns. For example, for basic authentication, security tools like Acunetix or Burp Suite can verify the token is encrypted and the hash is correct. Such tools will provide you with a basic report, which you then must analyze carefully.*

. . .

3. **Excessive Data Exposure**

This vulnerability is related to exposing the data from API and which will render or display over the UI.

For example, we have an interface where we want to display three fields from user objects like name, address, and profile photo. But at the same time from API we are getting more data in user objects which is irrelevant in the case of the current user interface.

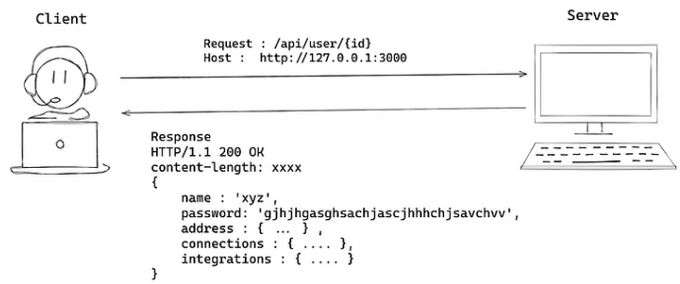
To prevent this we have returned the relevant fields in API response and not to shared sensitive data over API.

Use cases

1. The API returns full data objects as they are stored in the backend database.
2. The client application filters the responses and only shows the data that

the users really need to see.

3. Attackers call the API directly and get also the sensitive data that the UI would filter out.



Excessive Data Exposure

> *We have to test for Excessive Data Exposure manually. We can validate the API response fields via functional testing.*

. . .

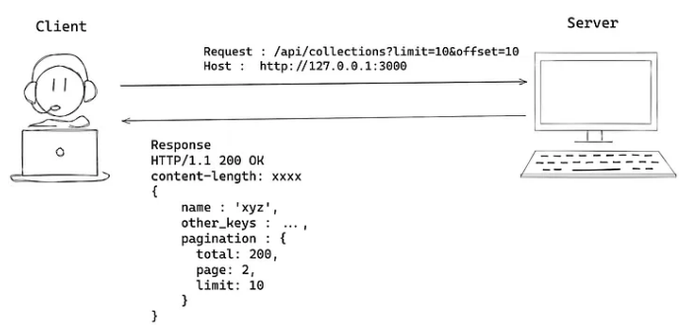
4. **Lack of Resources & Rate Limiting**

Improper rate limiting is a type of vulnerability that occurs when an API has no limit on the number of requests it sends to another API or a server.

The API is not protected against an excessive amount of calls or payload sizes. Attackers can use this for Denial of Service (DoS) and authentication flaws like brute force attacks.

Use Cases

1. The attacker may overload the API by sending unnecessary more request
2. Not having proper validation over the requested size of record for the per request



Lack of Resources & Rate Limiting

> *To find rate limiting vulnerabilities, you can use different fuzzing tools like JBroFuzz or Fuzzapi. Or, you can use the same tools with which you analyze traffic.*
>
> *Using 429 (Too Many Requests) status code we can implement rate limiting over overloaded unnecessary API requests.*
>
> *We can add proper validation over requested record size from API to return the limited record in response*

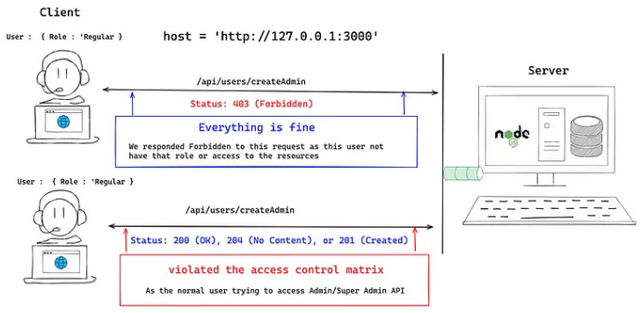
. . .

5. **Broken Function Level Authorization**

Complex access control policies with different hierarchies, groups, and

roles, and an unclear separation between administrative and regular functions, tend to lead to authorization flaws.

This vulnerability is concerned with vertical levels of authorization — The user attempting to gain more access rights than allowed. For example, a regular user trying to become an admin.



Broken Function Level Authorization

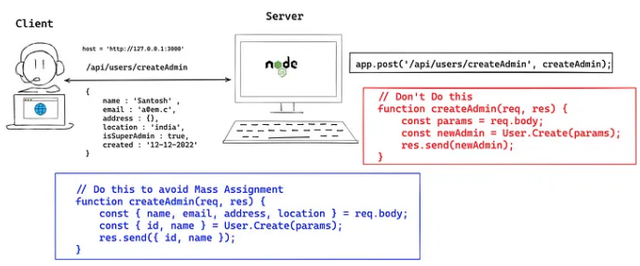
> *To find this vulnerability, we have to understand how various roles and objects in the application are connected and the access matrix implemented in the application*

. . .

6. **Mass Assignment**

The API takes data that the client provides and stores it without proper filtering for whitelisted properties.

Implement the validation middleware to get the parameters from client API requests and extract the particular fields which are needed to serve that request.



Mass Assignment

> *Use the `readOnly` property set to `true` in object schemas for all properties that can be retrieved through APIs but should never be modified.*
>
> *Precisely define the schemas, types, and patterns you will accept in requests at design time and enforce them at runtime.*

. . .

7. **Security Misconfiguration**

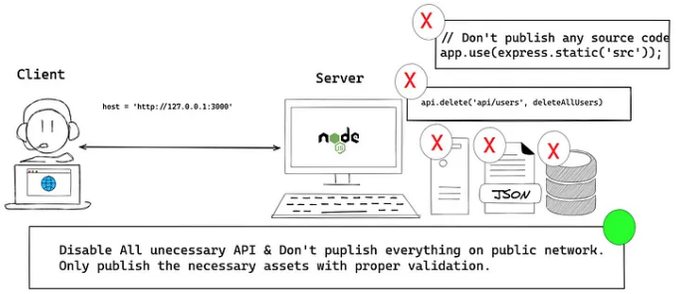
This vulnerability is related to the misconfiguration of your web servers or API.

Poor configuration of the API servers allows attackers to exploit them.

All unnecessary HTTP methods must be disabled on the server. Do not show any unnecessary user errors at all.

Do not pass technical details of the error to the client. If your application uses Cross-Origin Resource Sharing (CORS), that is, if it allows another application from a different domain to access your application's cookies, then these headers must be appropriately configured to avoid additional vulnerabilities.

Any access to internal files must also be disabled. There are special security headers, like Content-Security-Policy, that you can also implement in your applications to increase the security level.



Security Misconfiguration

> *If our application uses Cross-Origin Resource Sharing (CORS), that is, if it allows another application from a different domain to access our application's cookies, then these headers must be appropriately configured to avoid additional vulnerabilities. Any access to internal files must also be disabled. There are special security headers, like Content-Security-Policy, that we can also implement in your applications to increase the security level.*

. . .

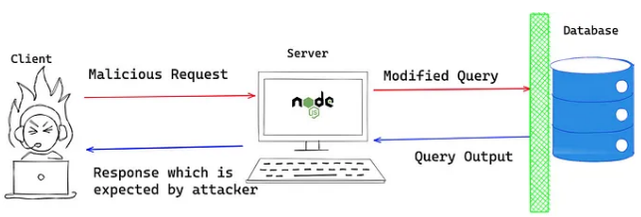
8. **Injection**

Attackers construct API calls that include SQL, NoSQL, LDAP, OS, or other commands that the API or the backend behind it blindly executes.

We can use the object-relational mapping model to avoid SQL injection.

This does not mean that we need to forget about injections at all.

Such problems are still possible throughout a huge number of old sites and systems. Besides XSS and SQL, we should look for XML injections, JSON injections, and so on.



Injection

> *We can test for injections using different tools. For example, ReadyAPI provides a paid tool for automatic scanning. Others, like Burp Suite, are partially free. Or, if you use Postman on a project, you could perform basic injection tests using Postman and data-driven testing.*

. . .

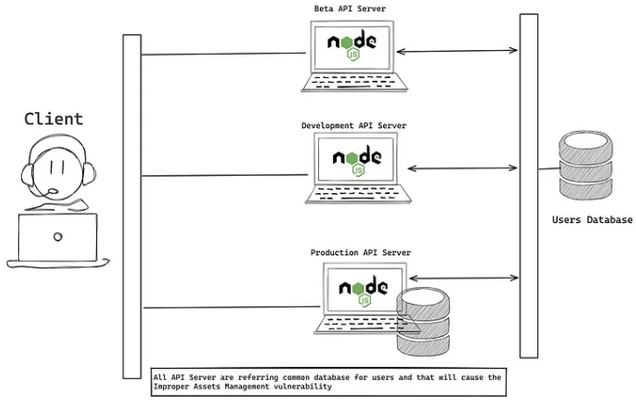
9. **Improper Assets Management**

This vulnerability is related to the mismanagement of your assets.

This flaw is growing as engineers adopt DevOps, continuous testing, and CI/CD pipelines.

From a security standpoint, it is essential to configure these CI/CD pipelines correctly.

Attackers find non-production versions of the API (for example, staging, testing, beta, or earlier versions) that are not as well protected as the production API, and use those to launch their attacks.



Improper Assets Management

> *When using virtual machines, containers may be created by the CI/CD pipeline, and microservices may be placed in a separate container. Throughout this process, make sure we don't have old containers hanging around that everyone has forgotten about — these could easily become additional access points.*

. . .

10. **Insufficient Logging & Monitoring**

This vulnerability has to do with insufficient logging and monitoring procedures.

The main idea here is that whatever happens to our application, we must be sure that we can track it. We should always have logs that show precisely what the attacker was trying to do.

Also, have systems in place to identify suspicious traffic, and so on.

Insufficient Logging & Monitoring

> *OWASP provides detailed checklists to reference to ensure your application is protected.*

. . .

Case Studies

Hackers can access the Nissan Leaf via insecure APIs

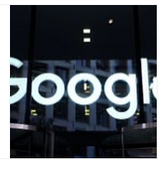
Two security researchers have demonstrated security vulnerabilities in the Nissan Leaf electric car by using mobile...

www.csoonline.com

Google+ Exposed Data of 52.5 Million Users and Will Shut Down in April

In October, Google dramatically announced that it would shut down Google+ in August 2019, because the company had...

www.wired.com



GitLab security update - API flaw could have exposed private events * DEVCLASS

GitLab released a slew of security updates yesterday, hard on the heels of the announcement of a vulnerability that...

devclass.com

Steam bug could have given you access to all the CD keys of any game | ZDNet

A Ukrainian vulnerability researcher has found a bug that would have allowed him to download all the activation keys...

www.zdnet.com

. . .

References

Architectural Styles and the Design of Network-based Software Architectures

The UNIVERSITY OF CALIFORNIA, IRVINE DISSERTATION submitted in partial satisfaction of the requirements for the degree of...

www.ics.uci.edu

OWASP API Security - Top 10 | OWASP

A foundational element of innovation in today's app-driven world is the API. From banks, retail, and transportation to...

owasp.org

Testing OWASP's Top 10 API Security Vulnerabilities | Nordic APIs |

Compared to web applications, API security testing has its own specific needs. Below, we cover the top vulnerabilities...

nordicapis.com

OWASP API Security Top 10

The Open Web Application Security Project (OWASP) is a non-profit, collaborative online community behind the OWASP Top...

apisecurity.io

Application Security Verification Standard

More content at ***plainenglish.io***

API | Security | JavaScript | Programming | Web Development

85 1

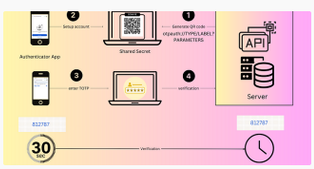
Written by Santosh Shinde

Follow

390 Followers · Writer for JavaScript in Plain English

Lead Software Engineer @Syngenta | #javascript #nodejs #typescript #reactjs #angular | #followme @ https://twitter.com/shindesan2012

More from Santosh Shinde and JavaScript in Plain English



Santosh Shinde in JavaScript in Plain English

Multi-factor authentication using Node JS and React JS

2FA TOTP implementation using Node.js, TypeScript, and React.js.

Apr 26 68

Afan Khan in JavaScript in Plain English

Microsoft is ditching React

Here's why Microsoft considers React a mistake for Edge.

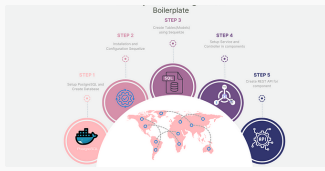
Jun 6 1.8K 47

Rehan Pinjari in JavaScript in Plain English

15 Time-Saving Websites Every Developer Needs

Ever thought that there aren't enough hours in the day for all your never-ending...

Apr 29 1.6K 20



Santosh Shinde in JavaScript in Plain English

How to Use Sequelize with Typescript, Node.js, and...

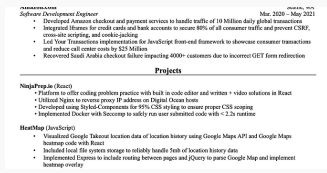
Getting started guide for Configure Sequelize with TypeScript by defining models,...

Mar 16 39 1

See all from Santosh Shinde

See all from JavaScript in Plain English

Recommended from Medium


Software Development Engineer
Mar. 2020 – May 2021
- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Braintree for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)
- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

HeatMap (JavaScript)
- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with Rust
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay

Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

Jun 1 10K 132

Ege Aytin in Permify Tech Blog

JWT vs PASETO: New Era of Token-Based Authentication

This article delves into a comprehensive comparison of Paseto and JWT, dissecting...

Jun 6 935 12

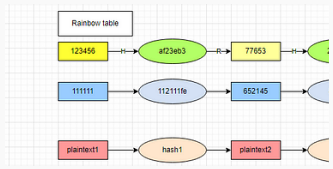
Lists

General Coding Knowledge
20 stories · 1337 saves

Coding & Development
11 stories · 677 saves

Stories to Help You Grow as a Software Developer
19 stories · 1167 saves

ChatGPT
21 stories · 694 saves



LORY

A basic question in security Interview: How do you store...

Explained in 3 mins.

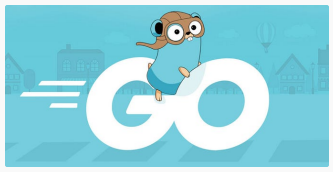
May 12 3.8K 39

Oliver Foster in Stackademic

What's the Difference Between localhost and 127.0.0.1?

My article is open to everyone; non-member readers can click this link to read the full text.

Feb 1 3.4K 22



CJ Hewett

Node.js is faster than Go

...when you write bad code. A lesson in connection pooling, and not simply copying...

Apr 1 504 9

Daniel Craciun in JavaScript in Plain English

Stop Using UUIDs in Your Database

How UUIDs can Destroy SQL Database Performance

May 16 2K 84

See more recommendations