“How Secure are you APIs?” Securing your APIs: OWASP API Top 10 2019, Case Study and Demo

Nathan Aw
https://www.linkedin.com/in/awnathan

24 June 2020

This Talk

- Background - Context - Problem Statement
- OWASP API Top 10
- Demo - “Broken Object Level Authorization” (Yes, a demo!)
- Real-Life Case Study

Opinions/views expressed in the talk are solely my own and do not express the views or opinions of my employer.

Who I am. Hello.

- Currently an **AppDevSec** Digital Solutions Architect and a Full-Stack Developer in the Financial Services Industry (FSI)
  - First a Full-Stack Cloud-Native Developer, then a Solutions Architect
  - Previously worked in a local bank as a Full-Stack Blockchain Engineer
- **Specialties** around API, Microservices that enables a Customer Journey Experience (CJX)
  - On “Hybrid-Multi” Cloud Native Platforms
  - On API, Microservices Security, Container Runtime Security and MITRE ATT&CK® for Kubernetes(K8S)
- Technology Stack: React, Kafka, Spring Boot, NodeJS, Apigee, Kong, Zuul, GraphQL, Azure Kubernetes Service (AKS), Elastic Kubernetes Service (EKS), Openshift, Service Mesh (Istio, Linkerd, Envoy), Cloud Foundry and many more…
- Designing, building and operating Scalable, Secure and Robust APIs and Microservices is my passion!
- https://www.linkedin.com/in/awnathan

Pop Quiz!

How many % of the internet traffic today is API Traffic?

A 9%

B 15%

C 83%

D 97%

Correct Answer is…. **83%!**

How many % of the internet traffic today is API Traffic?

A 9%

B 15%

[✓ C 83%]

D 97%

SOURCE:
Akamai - State of the Internet Security Report 2019
https://www.akamai.com/us/en/about/news/press/2019-press/state-of-the-internet-security-retail-attacks-and-api-traffic.jsp

Secure and Available APIs are extremely important!

Background - Context (1/3)

“The future of financial services is **<u>not</u>** technology… it is the relentless, unwavering and undying *focus* on the consumer.

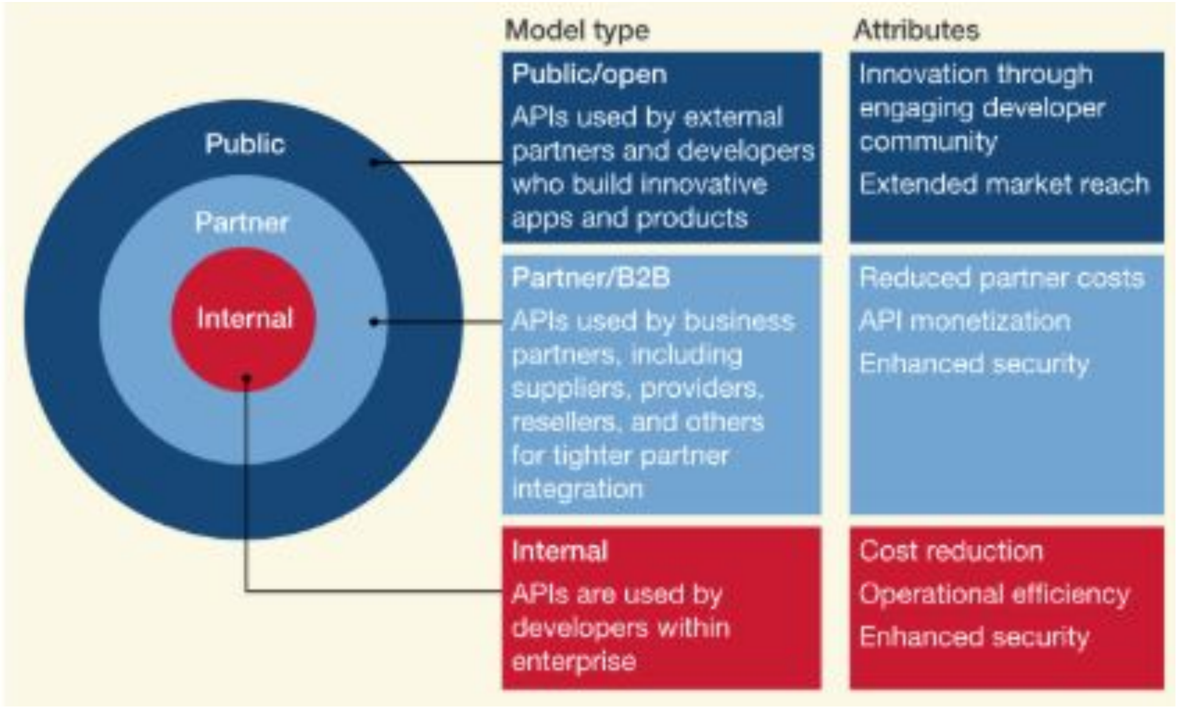
Technology, then, is merely the enabler of this customer focus and the very bridge to the consumer.”

- Nathan Aw

1. “APIs” as the “bridge” to your consumer
2. Embedding “APIs” in the consumer digital ecosystem

<u>Availability</u> and <u>Security</u> of APIs are therefore very crucial to serving the consumer.

Background - Context - Different Types of API (2/3)



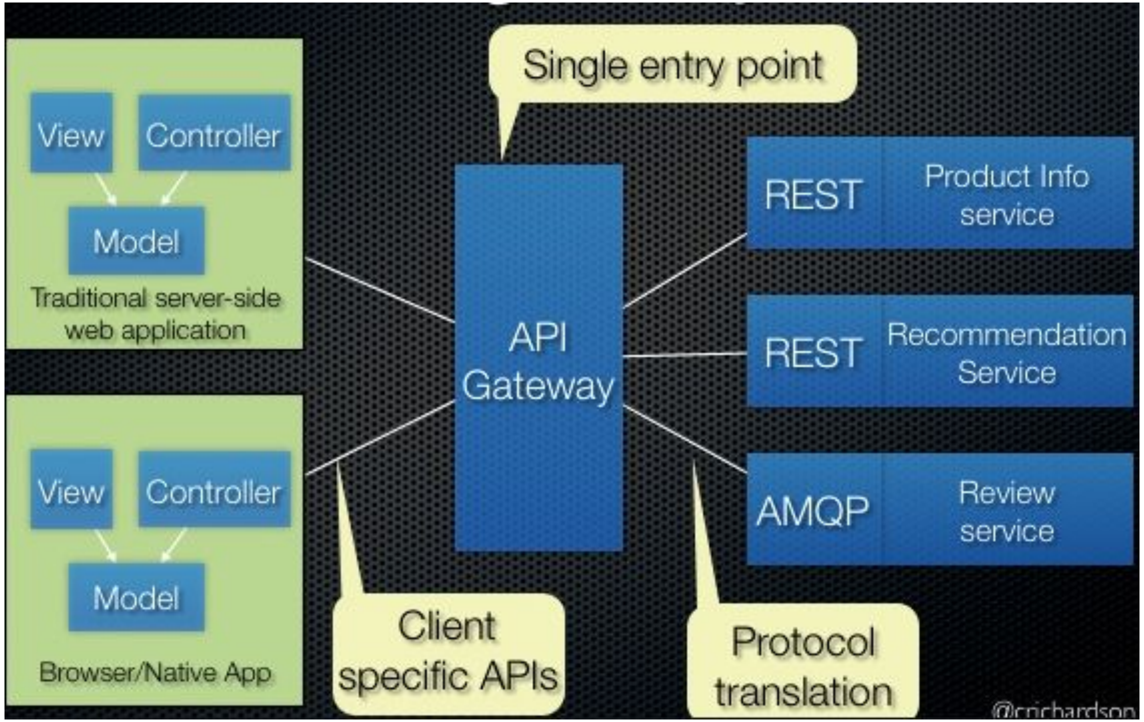
Model type	Attributes
Public/open APIs used by external partners and developers who build innovative apps and products	Innovation through engaging developer community Extended market reach
Partner/B2B APIs used by business partners, including suppliers, providers, resellers, and others for tighter partner integration	Reduced partner costs API monetization Enhanced security
Internal APIs are used by developers within enterprise	Cost reduction Operational efficiency Enhanced security

- Different Types of APIs Model
  - Public/Open
  - Partner/B2B
  - Internal

...Each Types Requires a Different Approach to Security

SOURCE: Mckinsey

Background - Context (3/3)



Software Architectures are rapidly changing with APIs as key building blocks/entry points for many Front Ends. Web Applications, Mobile App, etc

...APIs too need to be secured and pen tested!

OWASP API Top 10 2019: The Ten Most Critical API Security Risks

OWASP API Security Top 10 2019

The Ten Most Critical API Security Risks

OWASP API Top 10 2019

I read this, breathe this and **distilled** this so you don't have to!

SOURCE:
https://github.com/OWASP/API-Security/raw/master/2019/en/dist/owasp-api-security-top-10.pdf

OWASP API Top 10 2019: The Ten Most Critical API Security Risks

- 🟧 Broken Object Level Authorization
- 🟦 Broken User Authentication
- ⬜ Excessive Data Exposure
- 🟦 Lack of Resources & Rate Limiting
- ⬜ Broken Function Level Authorization
- ⬜ Mass Assignment
- 🟦 Security Misconfiguration
- ⬜ Injection
- ⬜ Improper Assets Management
- 🟦 Insufficient Logging & Monitoring

🟧 Demo Item

🟦 To be covered in greater depth

OWASP API Top 10 2019: The Ten Most Critical API Security Risks (1/4)

Broken Object Level Authorization	APIs tend to expose endpoints that handle **object identifiers**, creating a wide attack surface Level Access Control issue. Object level authorization checks should be considered in every function that accesses a data source using an input from the user.
Broken User Authentication	Authentication mechanisms are often implemented incorrectly, allowing attackers to **compromise authentication tokens** or to exploit implementation flaws to assume other user's identities temporarily or permanently. Compromising system's ability to identify the client/user, compromises API security overall.
Excessive Data Exposure	Looking forward to generic implementations, developers tend to **expose all object properties** without considering their individual sensitivity, relying on clients to perform the data filtering before displaying it to the user.

Demo Item

OWASP API Top 10 2019: The Ten Most Critical API Security Risks (2/4)

Lack of Resources & Rate Limiting

Quite often, APIs do not impose any restrictions on the size or number of resources that can be requested by the client/user. Not only can this impact the API server performance, leading to **Denial of Service (DoS)**, but also leaves the door open to **authentication flaws** such as brute force.

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. By exploiting these issues, attackers gain access to other users' resources and/or administrative functions.

OWASP API Top 10 2019: The Ten Most Critical API Security Risks (3/4)

Mass Assignment	Binding client provided data (e.g., JSON) to data models, without proper properties filtering based on a whitelist, usually lead to Mass Assignment. Either guessing objects properties, exploring other API endpoints, reading the documentation, or providing additional object properties in request payloads, allows attackers to modify object properties they are not supposed to.
Security Misconfiguration	Security misconfiguration is commonly a result of unsecure default configurations, incomplete or ad-hoc configurations, open cloud storage, misconfigured HTTP headers, unnecessary HTTP methods, permissive Cross-Origin resource sharing (CORS), and verbose error messages containing sensitive information.
Injection	Injection flaws, such as SQL, NoSQL, Command Injection, etc., occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's malicious data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

OWASP API Top 10 2019: The Ten Most Critical API Security Risks (4/4)

Improper Assets Management

APIs tend to expose more endpoints than traditional web applications, making proper and updated documentation highly important. Proper hosts and deployed API versions inventory also play an important role to mitigate issues such as deprecated API versions and exposed debug endpoints.

Insufficient Logging & Monitoring

Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems to tamper with, extract, or destroy data. Most breach studies demonstrate the time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

Broken Object Level Authorization ("BOLA")(1/2)*

*Demo

APIs tend to expose endpoints that handle object identifiers, creating a wide attack surface Level Access Control issue. Object level authorization checks should be considered in every function that accesses a data source using an input from the user.

What is it?	Attackers can exploit API endpoints that are vulnerable to broken object level authorization
How it is done?	By **manipulating the ID** of an object that is sent within the request.
Impact	This may lead to unauthorized access to sensitive data. **This issue is extremely common in API-based applications** because the server component usually does not fully track the client's state, and instead, relies more on parameters like object IDs, that are sent from the client to decide which objects to access. Unauthorized access can result in data disclosure to unauthorized parties, data loss, or data manipulation. Unauthorized access to objects can also lead to full account takeover. This has been the most common and impactful attack on APIs. Authorization and access control mechanisms in modern applications are complex and wide-spread. Even if the application implements a proper infrastructure for authorization checks, developers might forget to use these checks before accessing a sensitive object. Access control detection is not typically amenable to automated static or dynamic testing.

Broken Object Level Authorization (“BOLA”)(2/2)*

**Demo*

APIs tend to expose endpoints that handle object identifiers, creating a wide attack surface Level Access Control issue. Object level authorization checks should be considered in every function that accesses a data source using an input from the user.

Examples

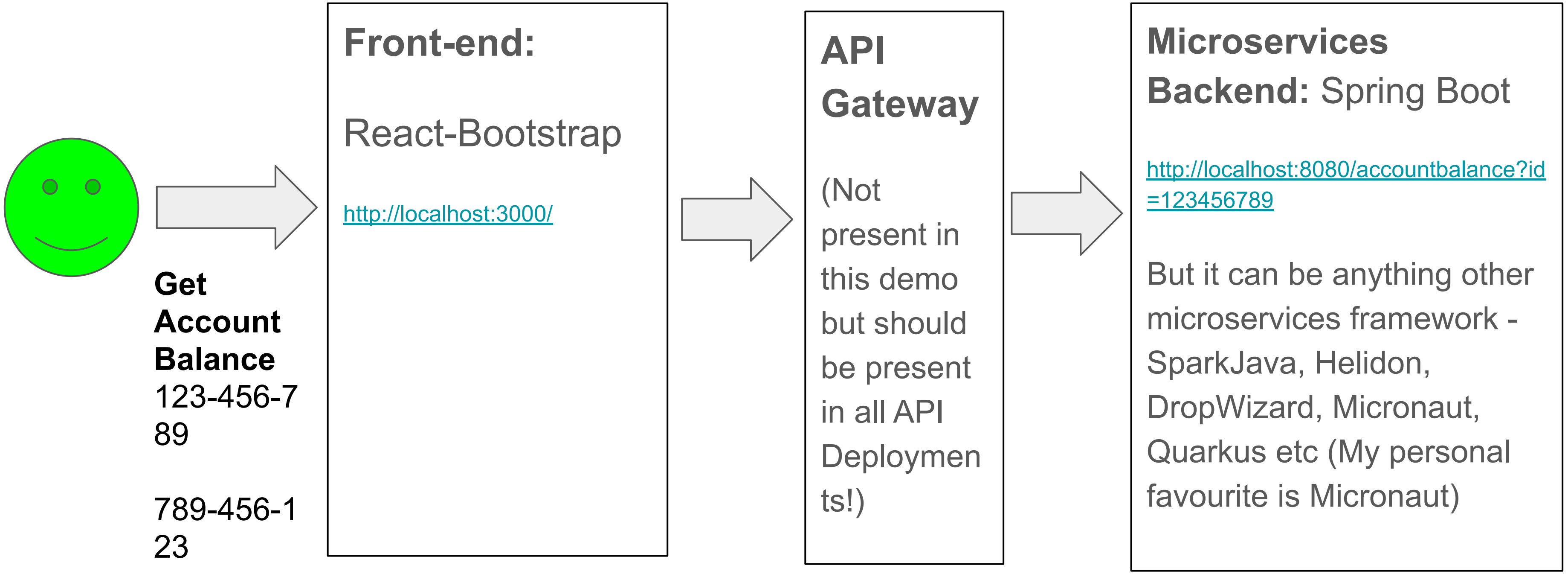
An e-commerce platform for online stores (shops) provides a listing page with the revenue charts for their hosted shops. Inspecting the browser requests, an attacker can identify the API endpoints used as a data source for those charts and their pattern /shops/{shopName}/revenue_data.json. Using another API endpoint, the attacker can get the list of all hosted shop names. With a simple script to manipulate the names in the list, replacing {shopName} in the URL, the attacker gains access to the sales data of thousands of ecommerce stores.

How to Prevent

Implement a proper authorization mechanism that relies on the user policies and hierarchy.

- Use an authorization mechanism to check if the logged-in user has access to perform the requested action on the record in every function that uses an input from the client to access a record in the database.
- Prefer to use random and unpredictable values as GUIDs for records’ IDs.
- Write tests to evaluate the authorization mechanism. Do not deploy vulnerable changes that break the tests.

Demo (“Broken Object Level Authorization”) - High Level Architecture (1/3)



Get Account Balance
123-456-789

789-456-123

Front-end:

React-Bootstrap

http://localhost:3000/

API Gateway

(Not present in this demo but should be present in all API Deployments!)

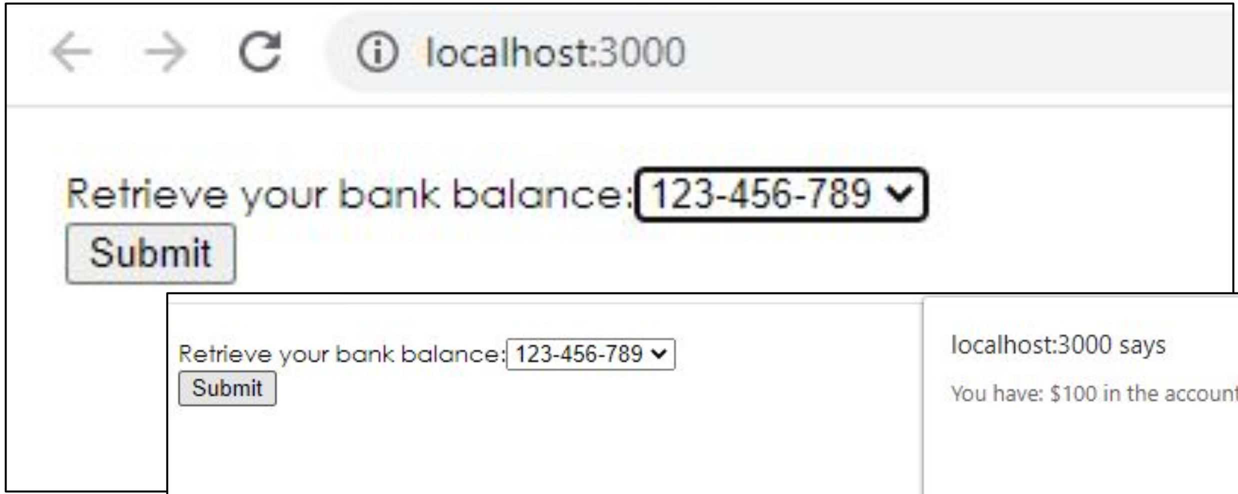
Microservices Backend: Spring Boot

http://localhost:8080/accountbalance?id=123456789

But it can be anything other microservices framework - SparkJava, Helidon, DropWizard, Micronaut, Quarkus etc (My personal favourite is Micronaut)

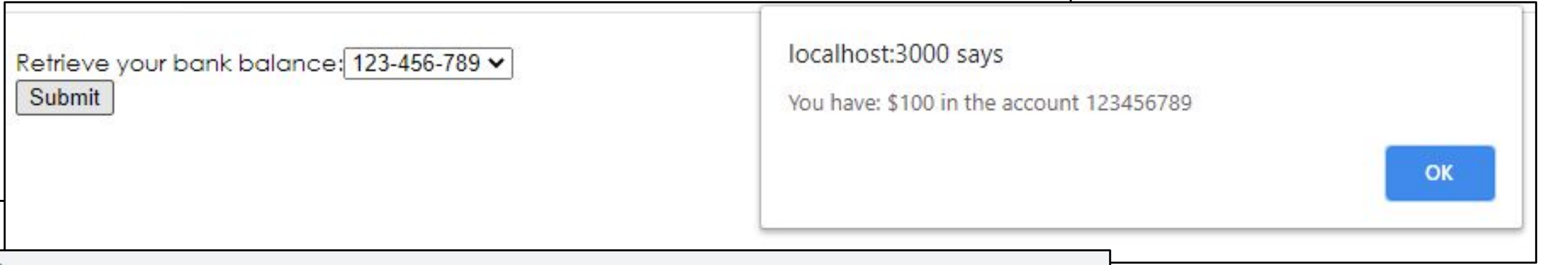
15 minutes to develop this demo so pardon the lack of aesthetics :)

Demo ("Broken Object Level Authorization") (2/3)



Retrieve your bank balance: 123-456-789

Submit

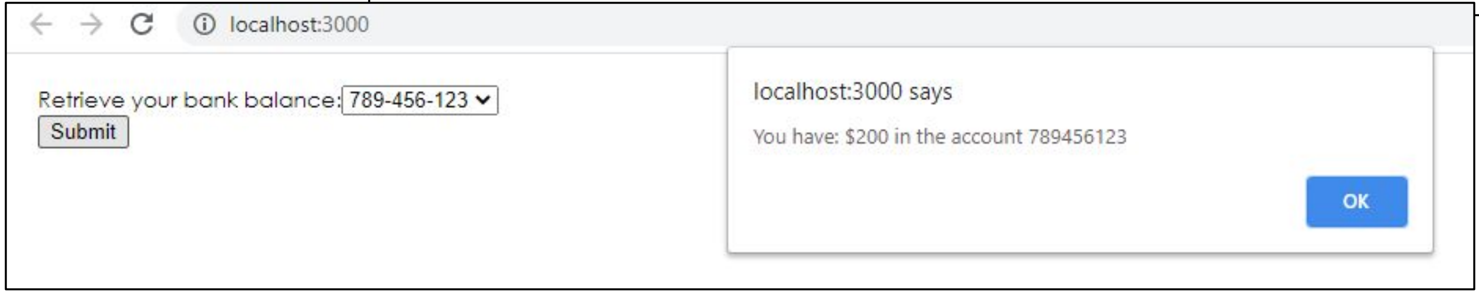


Retrieve your bank balance: 123-456-789

Submit

localhost:3000 says

You have: $100 in the account 123456789



Retrieve your bank balance: 789-456-123

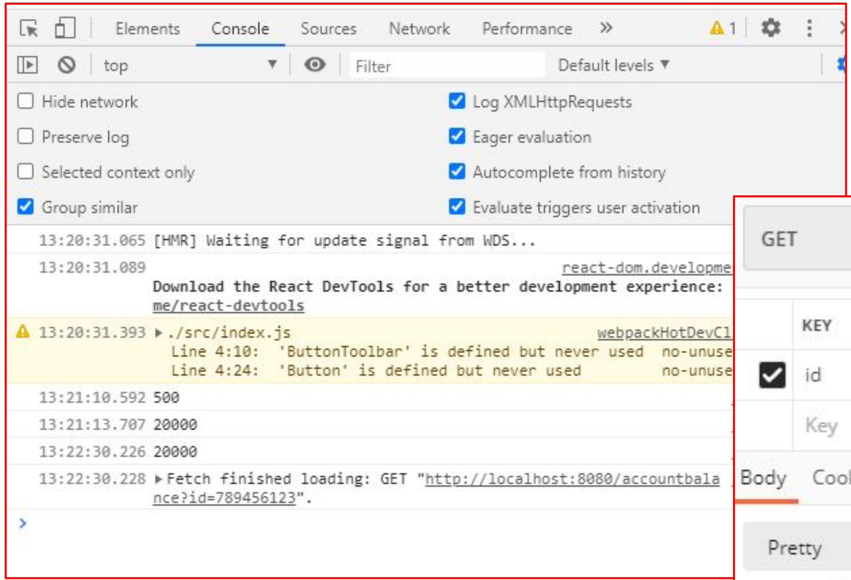
Submit

localhost:3000 says

You have: $200 in the account 789456123

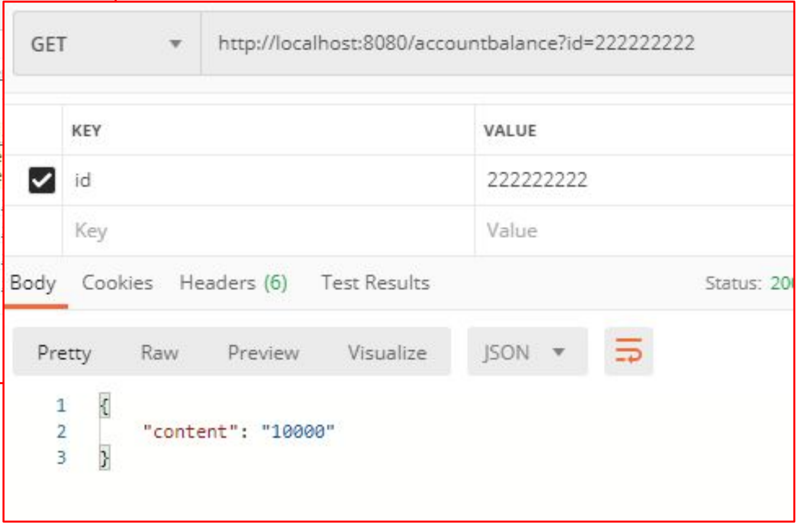
15 minutes to develop this demo so pardon the lack of aesthetics :)

Demo (“Broken Object Level Authorization”) (3/3)



- [ ] Hide network
- [ ] Preserve log
- [ ] Selected context only
- [x] Group similar
- [x] Log XMLHttpRequests
- [x] Eager evaluation
- [x] Autocomplete from history
- [x] Evaluate triggers user activation

```
13:20:31.065 [HMR] Waiting for update signal from WDS...
13:20:31.089                                          react-dom.developme
             Download the React DevTools for a better development experience:
             me/react-devtools
13:20:31.393 ▸./src/index.js                               webpackHotDevCl
               Line 4:10:  'ButtonToolbar' is defined but never used  no-unuse
               Line 4:24:  'Button' is defined but never used         no-unuse
13:21:10.592 500
13:21:13.707 20000
13:22:30.226 20000
13:22:30.228 ▸Fetch finished loading: GET "http://localhost:8080/accountbala
             nce?id=789456123".
```



GET http://localhost:8080/accountbalance?id=222222222

	KEY	VALUE
✓	id	222222222
	Key	Value

```
{
    "content": "10000"
}
```

The adversary could access other people bank account details by simply passing in the different account IDs which is not authorised to do so.

“Broken Object Level Authorization” - How to Prevent

How to Prevent

Implement a proper authorization mechanism that relies on the user policies and hierarchy.

• Use an authorization mechanism to check if the logged-in user has access to perform the requested action on the record in every function that uses an input from the client to access a record in the database.

• Prefer to use random and unpredictable values as GUIDs for records’ IDs. E.g., 82656a7f-3f8c-4926-a2a0-c38ef3459898

• Write tests to evaluate the authorization mechanism. Do not deploy vulnerable changes that break the tests.

Broken User Authentication (1/3)

What is it?

Authentication in APIs is a complex and confusing mechanism. Software and security engineers might have misconceptions about what are the boundaries of authentication and how to implement it correctly. In addition, the authentication mechanism is an easy target for attackers, since it's exposed to everyone. These two points makes the authentication component potentially vulnerable to many exploits.

How it is done?

1. Credential stuffing (using lists of known usernames/passwords), is a common attack. If an application does not implement automated threat or credential stuffing protections, the application can be used as a password oracle (tester) to determine if the credentials are valid.

2. For example, an attacker starts the password recovery workflow by issuing a POST request to **/api/system/verification-codes** and by providing the username in the request body. Next an SMS token with 6 digits is sent to the victim's phone. Because the API does not implement a rate limiting policy, the attacker can test all possible combinations using a multi-threaded script, against the **/api/system/verification-codes/{smsToken}** endpoint to discover the right token within a few minutes

Broken User Authentication (2/3)

Impact

Attackers can gain control to other users' accounts in the system, read their personal data, and perform sensitive actions on their behalf, like money transactions and sending personal messages.

How To Prevent (1/2)

• Ensure you know all the possible flows to authenticate to the API (mobile/ web/deep links that implement one-click authentication/etc.). Ask your engineers what flows you missed.

• Read about your authentication mechanisms. Make sure you understand what and how they are used. OAuth is not authentication, and neither is API keys.

• Don't reinvent the wheel in authentication, token generation, password storage. Use the standards.

• Credential recovery/forget password endpoints should be treated as login endpoints in terms of brute force, rate limiting, and lockout protections.

• Use the OWASP Authentication Cheatsheet.

Broken User Authentication (3/3)

How To Prevent (2/2)

• Where possible, implement multi-factor authentication.

• Implement anti brute force mechanisms to mitigate credential stuffing, dictionary attack, and brute force attacks on your authentication endpoints. This mechanism should be stricter than the regular rate limiting mechanism on your API.

• Implement account lockout / captcha mechanism to prevent brute force against specific users.

• Implement weak-password checks.

• API keys should not be used for user authentication, but for client app/project authentication.

Lack of Resources & Rate Limiting (1/2)

What is it?

Quite often, APIs do not impose any restrictions on the size or number of resources that can be requested by the client/user. Not only can this impact the API server performance, leading to Denial of Service (DoS), but also leaves the door open to authentication flaws such as brute force

How it is done?

Exploitation requires simple API requests. No authentication is required. Multiple concurrent requests can be performed from a single local computer or by using cloud computing resources. E.g., Hitting the /oauth endpoint repeatedly with the wrong credentials.

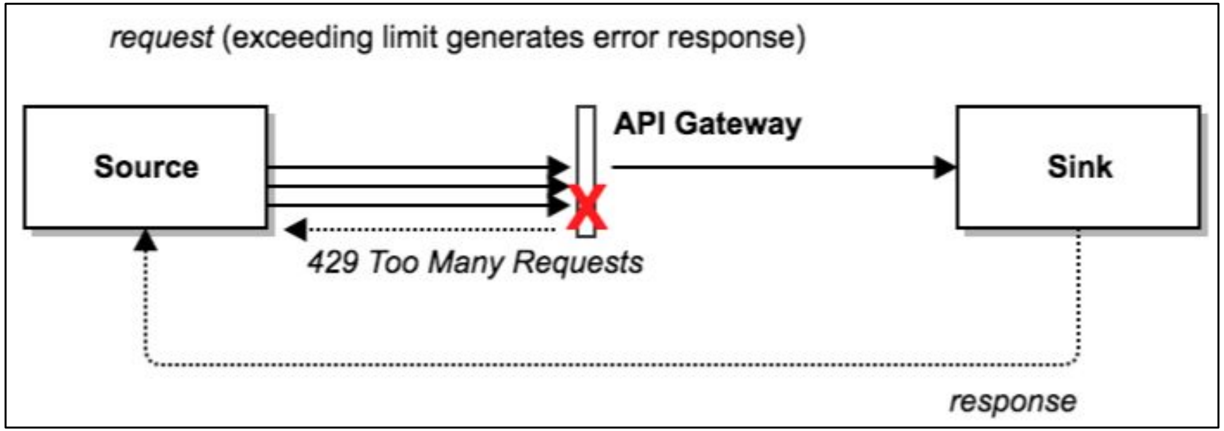
Scenario #1
An attacker uploads a large image by issuing a POST request to /api/v1/images. When the upload is complete, the API creates multiple thumbnails with different sizes. Due to the size of the uploaded image, available memory is exhausted during the creation of thumbnails and the API becomes unresponsive.

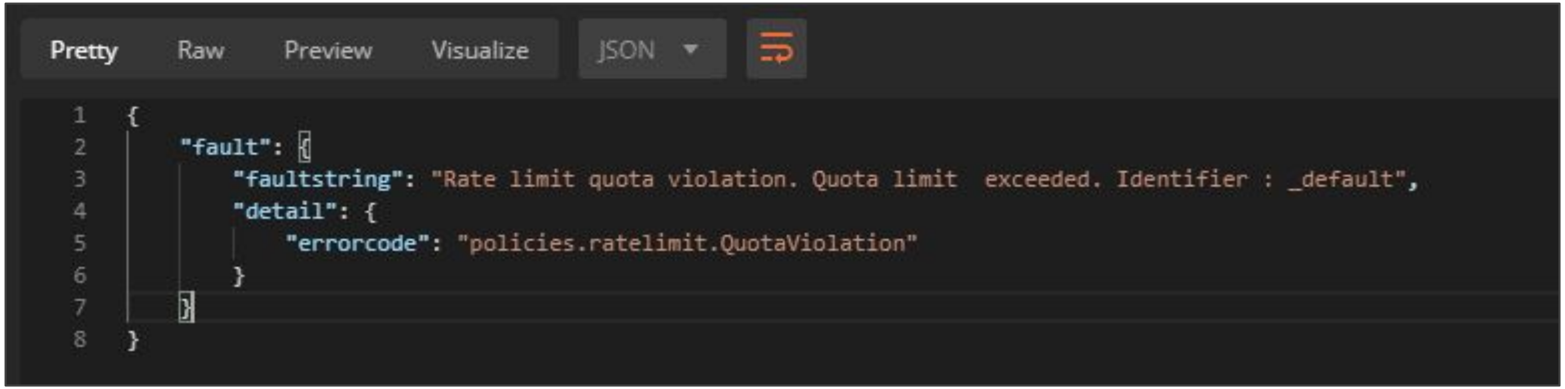
Scenario #2
We have an application that contains the users' list on a UI with a limit of 200 users per page. The users' list is retrieved from the server using the following query: /api/users?page=1&size=100. An attacker changes the size parameter to 200 000, causing performance issues on the database. Meanwhile, the API becomes unresponsive and is unable to handle further requests from this or any other clients (aka DoS). The same scenario might be used to provoke Integer Overflow or Buffer Overflow errors.

Lack of Resources & Rate Limiting (2/2)

Impact	Exploitation may lead to DoS, making the API unresponsive or even unavailable.
How To Prevent	• Implement a limit on how often a client can call the API within a defined timeframe - A rate limit is the number of API calls an app or user can make within a given time period. If this limit is exceeded or if CPU or total time limits are exceeded, the app or user will be throttled and API requests will fail. • Docker makes it easy to limit memory, CPU, number of restarts, file descriptors, and processes. • Notify the client when the limit is exceeded by providing the limit number and the time at which the limit will be reset. • Add proper server-side validation for query string and request body parameters, specifically the one that controls the number of records to be returned in the response. • Define and enforce maximum size of data on all incoming parameters and payloads such as maximum length for strings and maximum number of elements in arrays.

Lack of Resources & Rate Limiting





```
{
    "fault": {
        "faultstring": "Rate limit quota violation. Quota limit  exceeded. Identifier : _default",
        "detail": {
            "errorcode": "policies.ratelimit.QuotaViolation"
        }
    }
}
```

SOURCE: https://blog.getambassador.io/rate-limiting-for-api-gateways-892310a2da02 / https://docs.apigee.com/api-platform/develop/rate-limiting

Security Misconfiguration (1/3)

Security Misconfiguration

What is it?

Attackers will often attempt to find unpatched flaws, common endpoints, or unprotected files and directories to gain unauthorized access or knowledge of the system.

Security misconfiguration can happen at any level of the API stack, from the network level to the application level. Automated tools are available to detect and exploit misconfigurations such as unnecessary services or legacy options.

How it is done?

Scenario #1
An attacker finds the .bash_history file under the root directory of the server, which contains commands used by the DevOps team to access the API:

```
$ curl -X GET
'https://api.server/endpoint/' -H
'authorization: Basic Zm9vOmJhcg=='
```

An attacker could also find new endpoints on the API that are used only by the DevOps team and are not documented.

Scenario #2
To target a specific service, an attacker uses a popular search engine to search for computers directly accessible from the Internet. The attacker found a host running a popular database management system, listening on the default port. The host was using the default configuration, which has authentication disabled by default, and the attacker gained access to millions of records with PII, personal preferences, and authentication data.

Security Misconfiguration (2/3)

How it is done?

Scenario #3
Inspecting traffic of a mobile application an attacker finds out that not all HTTP traffic is performed on a secure protocol (e.g., TLS). The attacker finds this to be true, specifically for the download of profile images.

As user interaction is binary, despite the fact that API traffic is performed on a secure protocol, the attacker finds a pattern on API responses size, which he uses to track user preferences over the rendered content (e.g., profile images).

Impact

Security misconfigurations can not only expose sensitive user data, but also system details that may lead to full server compromise.

Security Misconfiguration (3/3)

Security Misconfiguration

How To Prevent

The API life cycle should include:

• A repeatable hardening process leading to fast and easy deployment of a properly locked down environment.

• A task to review and update configurations across the entire API stack. The review should include: orchestration files, API components, and cloud services (e.g., S3 bucket permissions).

• A secure communication channel for all API interactions access to static assets (e.g., images).

• An automated process to continuously assess the effectiveness of the configuration and settings in all Environments.

• To prevent exception traces and other valuable information from being sent back to attackers, if applicable, define and enforce all API response payload schemas including error responses.

• Ensure API can only be accessed by the specified HTTP verbs. All other HTTP verbs should be disabled (e.g. HEAD).

• APIs expecting to be accessed from browser-based clients (e.g., WebApp front-end) should implement a proper Cross-Origin Resource Sharing (CORS) policy.

Insufficient Logging & Monitoring (1/2)

What is it?	Attackers take advantage of lack of logging and monitoring to abuse systems without being noticed. Without logging and monitoring, or with insufficient logging and monitoring, it is almost impossible to track suspicious activities and respond to them in a timely fashion.
How it is done?	Scenario #1 Access keys of an administrative API were leaked on a public repository. The repository owner was notified by email about the potential leak, but took more than 48 hours to act upon the incident, and access keys exposure may have allowed access to sensitive data. Due to insufficient logging, the company is not able to assess what data was accessed by malicious actors. Scenario #2 A video-sharing platform was hit by a “large-scale” credential stuffing attack. Despite failed logins being logged, no alerts were triggered during the timespan of the attack. As a reaction to user complaints, API logs were analyzed and the attack was detected. The company had to make a public announcement asking users to reset their passwords, and report the incident to regulatory authorities.

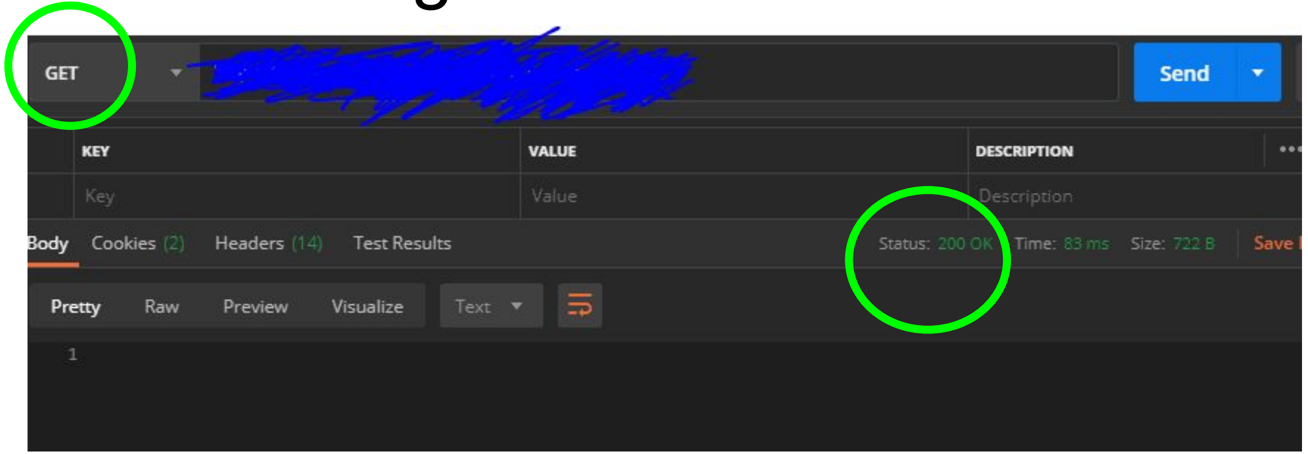
Insufficient Logging & Monitoring (2/2)

How To Prevent

- Log all failed authentication attempts, denied access, and input validation errors.
- Logs should be written using a format suited to be consumed by a log management solution, and should include enough detail to identify the malicious actor.
- Logs should be handled as sensitive data, and their integrity should be guaranteed at rest and transit.
- Configure a monitoring system to continuously monitor the infrastructure, network, and the API functioning.
- Use a Security Information and Event Management (SIEM) system to aggregate and manage logs from all components of the API stack and hosts.
- Configure custom dashboards and alerts, enabling suspicious activities to be detected and responded to earlier.

Case Study of Not Disabling Other HTTP Verbs

Security Misconfiguration - Ensure API can only be accessed by the specified HTTP verbs. All other HTTP verbs should be disabled (e.g. HEAD). In the following example, only GET is allowed. All other verbs should be rejected.

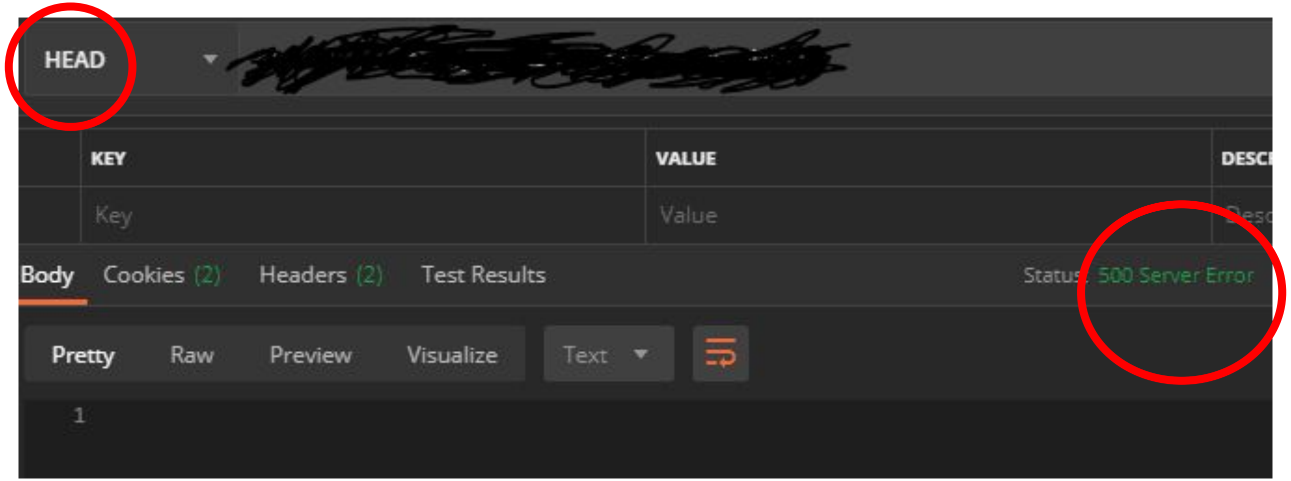


GET [redacted] Send

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies (2) Headers (14) Test Results Status: 200 OK Time: 83 ms Size: 722 B Save

Pretty Raw Preview Visualize Text



HEAD [redacted]

KEY	VALUE	DESC
Key	Value	Desc

Body Cookies (2) Headers (2) Test Results Status: 500 Server Error

Pretty Raw Preview Visualize Text

Diligence and Vigilance is the Key!

Diligence + **Vigilance**

Developer Responsibility + **Platform Responsibility**

Diligence and Vigilance is the key to ensuring comprehensively secure APIs

Both Developer Responsibility + Platform Responsibility

Q&A

Feel reach out to me @ https://www.linkedin.com/in/awnathan | nathan.mk.aw@gmail.com

- Currently an **AppDevSec** Digital Solutions Architect and a Full-Stack Developer in the Financial Services Industry (FSI)
  - First a Full-Stack Developer, then a Solutions Architect
  - Previously worked in a local bank as a Full-Stack Blockchain Engineer
- **Specialties** around API, Microservices that enables a Customer Journey Experience (CJX)
  - On "Hybrid-Multi" Cloud Native Platforms
  - On API, Microservices Security, Container Runtime Security and MITRE ATT&CK® for Kubernetes(K8S)
- Technology Stack: React, Kafka, Spring Boot, NodeJS, Apigee, Kong, Zuul, GraphQL, Azure Kubernetes Service (AKS), Elastic Kubernetes Service (EKS), Openshift, Service Mesh (Istio, Linkerd), Cloud Foundry and many more…
- Building Scalable, Secure and Robust APIs and Microservices is my passion!
- https://www.linkedin.com/in/awnathan
- Next Presentation on July 15th: "**Microservices Security, Container Runtime Security and MITRE ATT&CK® for Kubernetes (K8S)**"
- ***Opinions/views expressed here are solely my own and do not express the views or opinions of my employer.***

Next Presentation on 15th July: **“Microservices Security, Container Runtime Security and MITRE ATT&CK® for Kubernetes (K8S)”**

APIs are the gate to your microservices. In this session, we have learnt how to secure our APIs. How, then, do you protect your microservices?

We will explore in-depth in the next webinar: “Microservices Security, Container Runtime Security and MITRE ATT&CK® for Kubernetes (K8S)”

Join the next session on 15th July to find out:

https://meet.google.com/zxc-nmnj-zqv

References:

https://github.com/nathanawmk/owasp

https://www.akamai.com/us/en/about/news/press/2019-press/state-of-the-internet-security-retail-attacks-and-api-traffic.jsp

https://github.com/OWASP/API-Security/raw/master/2019/en/dist/owasp-api-security-top-10.pdf