CISO MAG
beyond cybersecurity

WHITEPAPER

API Security: Best Practices for Vulnerability Mitigation

Authored by:
Jagdish Mohite
Principal Security Consultant at Akamai Technologies
OSCP, OSWP, CRTP, CISSP, CISA, CEH, CHFI, PMP

ABSTRACT

API Security: Best Practices for Vulnerability Mitigation provides a hands-on approach to mitigate security vulnerabilities in APIs. The paper emphasizes the importance of implementing security measures that protect the API and underlying infrastructure. The paper identifies various security vulnerabilities that can arise in APIs and provides detailed guidelines for securing them. These guidelines cover authentication, authorization, input validation, output encoding, error handling, logging, and auditing. The paper discusses the OWASP top 10 API vulnerabilities 2023 release. The paper also provides practical examples and code snippets to illustrate implementing the recommended security measures in popular API frameworks such as Node.js. Furthermore, the paper outlines how to perform API security testing and monitoring to detect and address potential vulnerabilities. The paper provides a comprehensive and practical approach to securing APIs and mitigating security vulnerabilities, making it a valuable resource for developers and security professionals.

Contents

INTRODUCTION

APIs and applications form the backbone of many modern organizations, bridging the gap between customers and products, users, and databases. Thus, it is no surprise that they are attractive targets; they provide attackers with a means of infiltrating an organization. As businesses invest in digital innovation, threat actors also invest in developing and evolving their attack campaigns to compromise online assets.
APIs provide a wide range of functionality and form the foundation for innovation and digital transformation. APIs have also become the de facto standard for building and connecting modern applications, especially as we move toward microservices-based architectures. APIs act as the digital glue that binds disparate systems and partner ecosystems, enabling digital and omnichannel customer experiences and exposing them to the same risks as traditional web applications. APIs are a crucial element for Digital Transformation. By streamlining development and generating 38% of the total organization's revenue, they help to achieve this objective. A clear API integration strategy is in place for 93% of enterprises, ahead of their expected digital transformation progress (2023 Connectivity Benchmark Report, 2023). Gartner 2021 report predicted API security as a significant and emerging threat in 2022 (How To Address Growing API Security Vulnerabilities In 2022, 2022), which proves true as we experienced many CVEs related to API vulnerabilities disclosed, the Atlassian Confluence vulnerability (CVE-2022-26134), ProxyNotShell vulnerability (CVE-2022–41040), and Spring4Shell/SpringShell (CVE-2022-22965) was considered as significant vulnerability disclosure among others (Akamai SOTI Report, 2023).

The proliferation of APIs in our day-to-day life gives rise to more sophisticated applications that improve and strengthen our capabilities. Still, at the same time, they expose us to more significant risks. As we depend more on APIs for critical tasks, our vulnerability becomes even more exposed when they get an attack. To mitigate the risks to the APIs, organizations need to adopt a different approach and have a clear mindset, not just focusing on design but dedicating more resources to security. Privacy and data protection legal requirements compel companies to protect users' data with heavy penalties if protections are inadequate. This document discusses the latest OWASP Security top 10 APIs vulnerabilities, best practices for securing APIs, testing methodology, and continuous monitoring of API security posture.

Common API Security Vulnerabilities:

According to the Akamai Threat Report H1 2022, there has been a considerable surge in attacks targeting web applications and APIs worldwide, with over 9 billion attack attempts year to date, an estimated three-fold increase compared to H1 2021. This report further identifies three primary attack vectors, namely local file inclusion (LFI), Structured Query Language injection (SQLi), and cross-site scripting (XSS). (Akamai Web Application and API Threat Report, 2022). APIs are machine-to-machine calls, while applications are consumed by humans. Organizations must adopt different mitigation models for Application and API security; they are separate disciplines (Dionisio Zumerle, 2022) Traditional application security mitigation controls are insufficient to protect APIs; they are more focused on securing apps. In contrast, API security controls protect APIs connecting different applications to exchange data.

The Open Web Application Security Project (OWASP) is a non-profit organization dedicated to improving software security. OWASP's Top 10 project has been the primary list of knowledge on Web Application vulnerabilities and protection. Evaluation of API is radically changed the security landscape, which demands an innovative mitigation approach; OWASP launched its effort in 2019 focusing on API's Top 10 security threats. (OWASP API Security Project, 2019) In February 2023, OWASP published the Top 10 security threats, an updated version of 2019, which addresses emerging attack vectors and changing threat landscape; the 2023 version addresses new attack vectors that have surfaced since the last version was released.

	2019	2023
1	Broken Object Level Authorization	Broken Object Level Authorization
2	Broken User Authentication	Broken User Authentication
3	Excessive Data Exposure ▼	Broken Object Property Level Authorization ▲
4	Lack of Resources & Rate Limiting ▼	Unrestricted Resource Consumption ▲
5	Broken Function Level Authorization	Broken Function Level Authorization
6	Mass Assignment ▼	Server-Side Request Forgery ▲
7	Security Misconfiguration	Security Misconfiguration
8	Injection ▼	Lack of Protection from Automated Threats ▲
9	Improper Assets Management	Improper Assets Management
10	Insufficient Logging & Monitoring ▼	Unsafe Consumption of APIs

Fig. 1: OWASP API top 10, 2019 Vs. 2023: ▼ **Sun Set** ▲ **New Addition**

Three companies, namely Salesforce, eBay, and Amazon, sensed the need for API during the 2000-2002 online shopping boom. In 2002 Salesforce released its first API, followed by eBay and Amazon (Hawkins, 2020). The first comprehensive law for sharing personal information, known as the General Data Protection Regulation (GDPR), was passed in 2018, which brings new challenges to many organizations using APIs that would need to produce and store users' personal information. The Cambridge Analytica Scandal, which exposed the data of up to 87 million Facebook users, revealed a dark side of API and their free information sharing of data leaving organizations scrambling to develop a system to track and monitor the API they use (Chang, 2018).

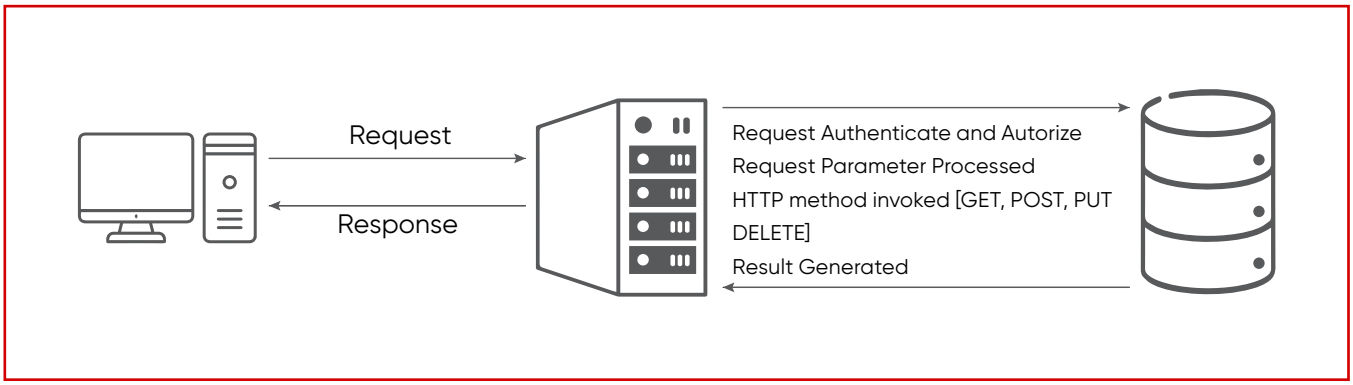


Fig.2: Client Server communication during API Calls

1. Broken Object Level Authorization:

Broken Object Level Authorization (BOLA) stands number one vulnerability in both versions of OWASP API top 10. When users access a resource they are not authorized to access, this occurs when the API endpoint does not have access level controls in place; the unprivileged user can freely access a resource on the server which he is not allowed to. BOLA vulnerability is easy to exploit and common among API-based applications because server-side code relies on parameter objects and does not track the client state. (OWASP Top 10 API, 2023).

Related CVE: CVE-2022-34770

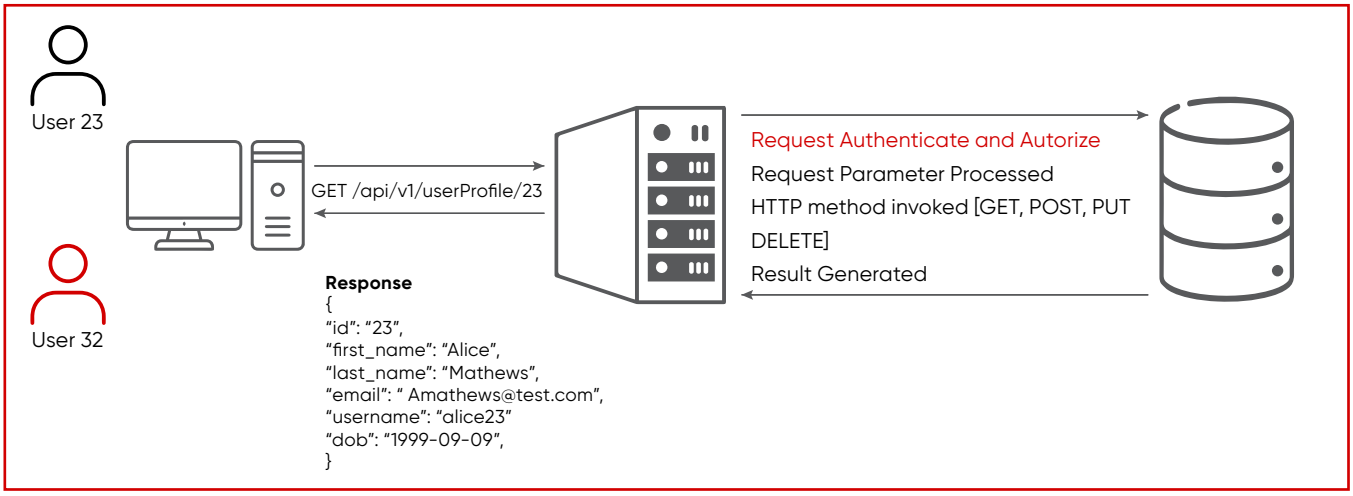


Fig.3: Broken Object Level Authorization: User32 can access User23's user profile

2. Broken User Authentication

Any weakness in the API authentication process is a Broken User Authentication. Broken User Authentication comes in many forms, for example, lack of authentication, lack of rate limiting applied to several authentication attempts, use of a single key or token created for all requests, insufficient token entropy, and several misconfigured JWT tokens. These weaknesses are most common when an API provider does not implement industry-standard strong authentication protection or implements it by reinventing the wheels via proprietary code (OWASP Top 10 API, 2023).

Related CVE: CVE-2023-22501

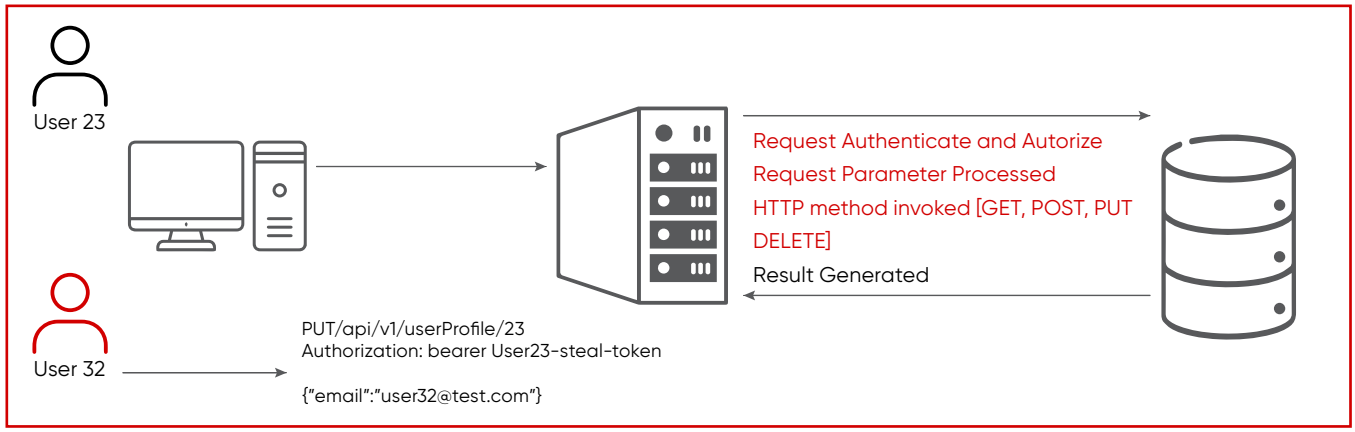


Fig.4: Broken User Authentication: Account takeover process, User32 can steal User23's token and update his mail in a password reset scenario.

3. Broken Object Property Level Authorization

Broken Object, Property Level Authorization, now includes two categories, excessive data exposure and mass assignment from the 2019 OWASP top 10. Server-side code fails to validate if the user can access specific properties within the object (OWASP Top 10 API, 2023). Users can change the value of object properties they are not supposed to access, resulting in data disclosure, manipulation, and data loss to an unauthorized user.

Related CVE: CVE-2022-29090, CVE-2020-24940

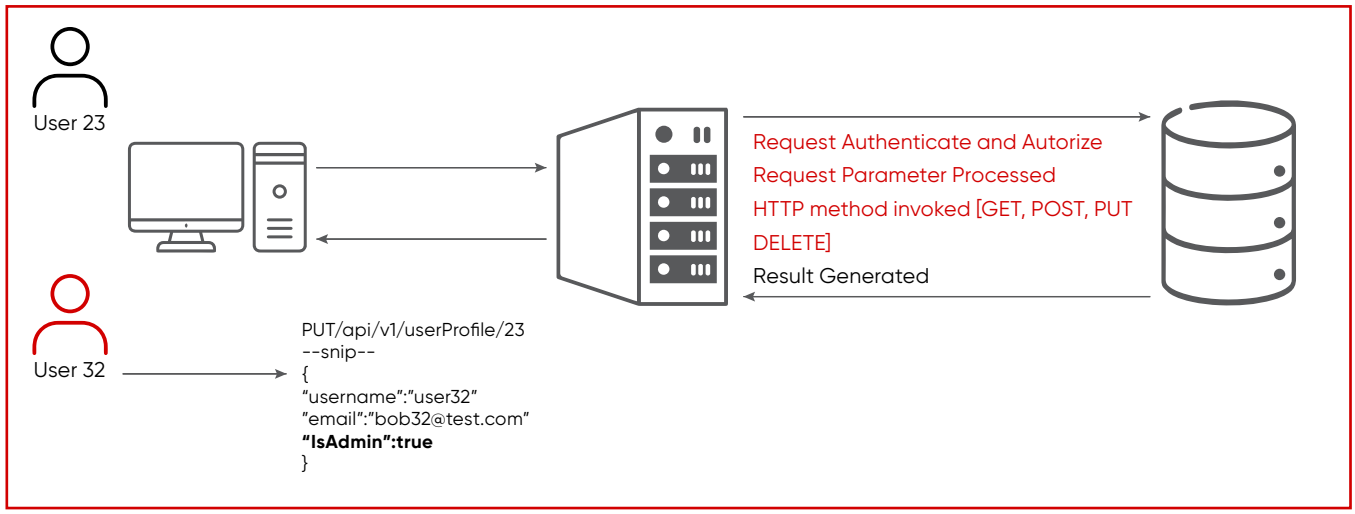


Fig 5: Unauthorized user32 can make himself admin.

4. Unrestricted Resource Consumption

Most automation uses calls to APIs, and failure to control unlimited usage could expose web services and applications to Denial of Service (DoS) attacks. An attacker can send multiple concurrent requests to the API endpoint to exhaust resources on the server; password brute-force, web scrapping, and scanning are some examples that fall into this category; this not only causes server starvation but affects the API provider's billing. This exploit is possible when the server-side code lacks controls to check the rate limit (2023 Connectivity Benchmark Report, 2023).

Related CVE: CVE-2023-1558, CVE-2022-1698

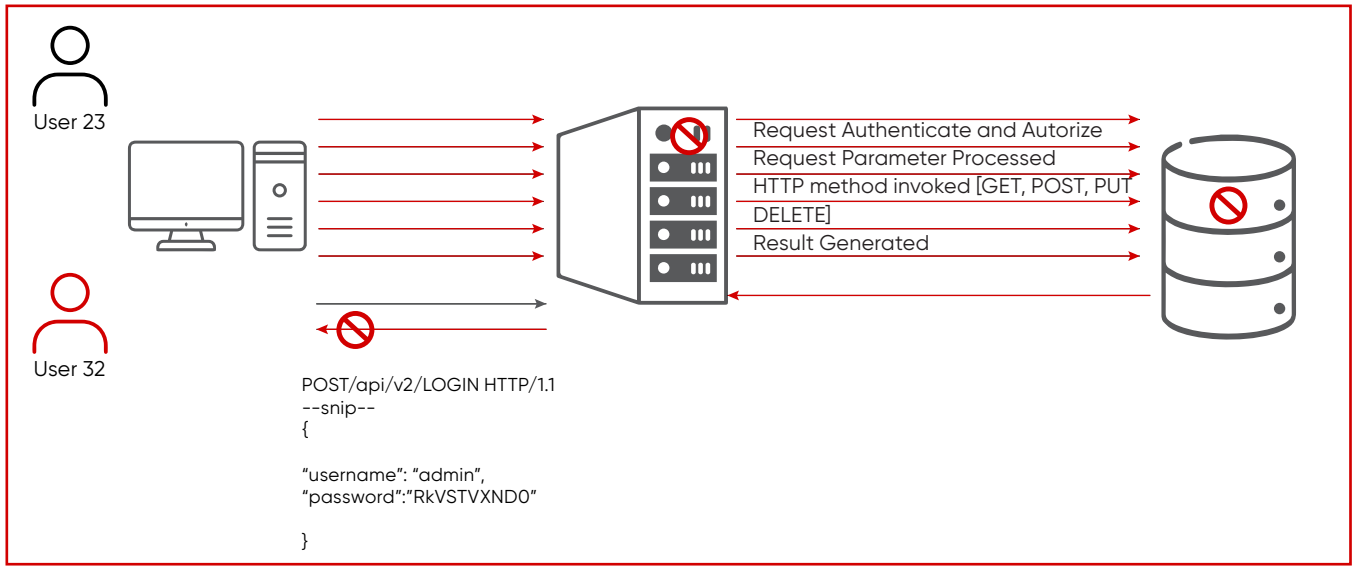


Fig 6: Denial of Service for legit user

5. Broken Function Level Authorization

A Broken Function Level Authorization (BFLA) vulnerability occurs when a user of one role or group has access to the API functionality of another role or group. BFLA is related to BOLA in that it involves an authorization problem for executing actions rather than an authorization problem for accessing resources; for example, when the DELETE function is only available to the superuser, a regular user can make a call to the DELETE function when the API endpoint is not validating authorization (OWASP Top 10 API, 2023). An unprivileged user can use the functionality of another privileged user if BFLA is present.

Related CVE: CVE-2021-21389

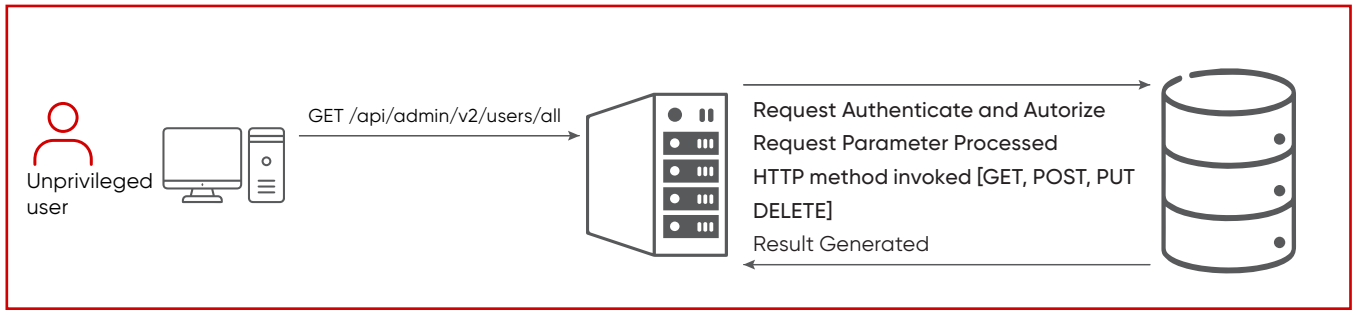


Fig 7: User32, an unprivileged user, can access admin functions.

6. Server-Side Request Forgery

Server-Side Request Forgery [SSRF] entered at number six in OWASP top 10 2023, replacing earlier Mass Assignment vulnerability. This vulnerability triggers when an API retrieves a remote resource without validating the user-supplied URL; it enables an attacker to force an application to submit a crafted request to an unexpected destination, even if a firewall or a VPN protects it. Based on the business use cases, it is impossible to eliminate SSRF risk, but applying the necessary mitigation developer can control this risk provided all business risks are addressed (OWASP Top 10 API, 2023).

To mitigate SSRF risk, the developer should implement input data validations to ensure that client-supplied input data follows the required format. Allow lists should be kept up to date so that only trusted requests/calls are executed, and HTTP redirections should be turned off (OWASP Top 10 API, 2023).

Related CVE: CVE-2022-28117

7. Security Misconfiguration

Security misconfigurations encompass any errors developers may make within an API's supported security setups. A significant security misconfiguration can expose crucial information or result in a total system takeover. For example, if the API's supported security settings show an unpatched vulnerability, an attacker might use a publicly available exploit and take down the system (Ball, 2022). Developer should pay attention to below best practices below recommended in OWASP top 10 2023 release,

- Harden system with best security practices pay close attention to permissions setup on cloud services.
- Keep all security patches up to date.
- Disable Unnecessary features (e.g., HTTP verbs, logging features)
- Verify that Transport Layer Security (TLS) is implemented.
- Verify Cross-Origin Resource Sharing (CORS) policy is appropriately configured.
- Verify Error messages, including stack traces, not exposing sensitive information. (OWASP Top 10 API, 2023)

Related CVE: CVE-2022-45139

8. Lack of Protection from Automated Threats

Lack of Protection from Automated Threats is a new entry in OWASP's top 10 2023. Innovative bot operators are the source of these threats, directly impacting business revenue. Bot operators can override
rate limit mitigation by accessing API from many locations/IPs worldwide in a fraction of a second (OWASP Top 10 API, 2023). A typical example could be an online ticket sale for a Super Bowl event; a malicious bot operator can run the automated script to purchase a maximum number of tickets, selling them at higher prices in the black market after the sale event.
To address these threats, Businesses should have a solution to determine whether a request is coming from a human or bots. Google Captcha and device fingerprinting can aid in thwarting this risk (OWASP Top 10 API, 2023)

9. Improper Assets Management

Often organizations need complete visibility into API inventory, including third-party API their application use. Outdated or unattended API documentation worsens the situation when finding and fixing API vulnerabilities. Due to a lack of asset inventory and retirement policies, unpatched systems are used, resulting in sensitive data loss. Because modern concepts like microservices make applications easy to deploy and independent, it is usual to find excessively exposed API hosts (OWASP Top 10 API, 2023).

Clear understanding and proper documentation are essential to mitigate this vulnerability. All details regarding API hosts, API environment, Network access, API version, Integrated services, redirections, rate limitation, and CORS policy should be carefully documented and updated. The general best practice is that every tiny detail should be documented and authorized access be granted to these records. System owners need to safeguard the exposed API version alongside the production version. A risk analysis is recommended when newer API versions are available (OWASP Top 10 API, 2023).

10.Unsafe Consumption of APIs

Unsafe Consumption of APIs is a newly added vulnerability at number ten in OWASP top 10 2023, replacing insufficient logging and monitoring. Developers trust data collected from third-party APIs more than user input, especially from well-known companies' APIs. As a result, developers need to pay closer attention to input validation and sanitization and apply stricter security controls for every API data collected. To mitigate this vulnerability, make sure API interaction over an encrypted channel, validate, and sanitize all data received, avoid blindly following redirection, implement a timeout, and put a limit on the number of resources available to process third-party service responses (OWASP Top 10 API, 2023).

Best Practices for API Security:

In a typical software development project, a team of developers, testers, and domain experts works on developing a critical feature. Performance, security, maintainability, and usability are the main attributes of a project; quality takes top priority, time to market is critical, and the team needs to be within budget. Developers implement open-source security libraries in their code, and the team focuses more on implementing business functionality where money is and keeping stakeholders happy. The product rolled out into production with no security review; users started to use the product, and all was well until one day, the product was all over the news for exposing confidential user data.

The above situation is common in most organizations that need to follow Security Best Practices. Sound security is not an accident when managing an organization's network, designing an app, or organizing paper files. organizations must get past thinking about security as a set of features to be genuinely secure (Start with Security: A Guide for Business, n.d.). Product managers must consider security as a concern and not a feature to secure APIs by following security best practices guidelines,

API Threat Modeling:

The threat is an event or series of circumstances threatening API's security objectives. Threat modeling is an engineering technique that may be applied to identify threats, attacks, vulnerabilities, and countermeasures that could pose a threat to APIs; this can be used to design secure APIs to meet organizational security objectives and to reduce risks (Microsoft Threat Modeling, n.d.), below some guidelines proposed by Microsoft.

- The system diagram needs to include all logical components of API.
- Determine trust boundaries between the various parts of the system and highlight data flow between various parts of the system. Identify flows that cross trust boundaries.
- List out threats to ensure they are tracked and managed. (1.4.3 Environments and threat models, 2020)

Authentication:

Authentication is the process of ensuring that users and clients are who they say they are. API authentication endpoints need an additional layer of protection that must be treated differently than other endpoints. There is always confusion or misconception among developers on deciding boundaries of authentication and correct implementation, so make sure the team knows all possible flows of API authentication (OWASP Top 10 API, 2023). Use the industry's well-known standards while implementing authentication, and token generation, do not reinvent the wheels. API keys are not for user authentication; they are for API client authentication; use multi-factor authentication for user authentication. Protect login endpoints with anti-brute force controls, for example, rate controls and lockout protections (OWASP Top 10 API, 2023)

Authorization:

Control of who has access to whom and what actions they may take is generally required to protect the confidentiality and integrity of assets (Vincent C. Hu, 2014). Authorization is governed by users' roles and groups, where different permissions are assigned to each role and group; applying the principle of least privilege reduces the chances of data exposure. API providers must implement robust access control, ensuring every request made is appropriately authorized. OAuth is a standard protocol for transmitting authorization; API providers must add an OpenId Connect standard identity layer that supports OAuth 2.0 with ID tokens. Use of random and unpredictable values as GUIDs for records' IDs can help further to reduce attack surface (OWASP Top 10 API, 2023)

Encryption:

If API transmits sensitive information over the wire, all traffic needs strong encryption. Encryption ensures that no unauthorized persons can read data, either during the transit from an API endpoint to its client or after being stored in the database and file system; encryption also ensures that no attacker can change the data. Encryption of transit data can be achieved via Transport Layer Protocol (TLS) while encryption of data at rest via Advance Encryption Standard (AES) as recommended by NIST (Protection of Data at Rest, 2018). The organization needs to follow the best practice of TLS implementation.

Security Monitoring and Audit logs:

The monitoring of API endpoints is a practice where the availability of API service is continuously checked for the correctness of transactions. It also gives insight into the API's performance regarding response time to the requests and assessing queries with differing complexity. Always configure checks on returned data that could trigger alerts accordingly. Implement sound API version management, which helps keep track of all changes and deprecates new APIs. Audit every record of API transaction; audit logs ensure accountability. It is necessary by-laws that audit logs be protected from tampering. The protection of audit trail data is necessary because they should be made available for use, where appropriate, and not necessarily helpful if their accuracy needs to be corrected. Adopt automation to review audit records in real-time or with a manual combination at regular intervals (Audit Trails -Chapter 18, NIST, n.d.)

Zero Trust:

According to Forrester, Zero Trust is an architectural model for how security teams should redesign networks into secure Micro perimeters, increase data security through obfuscation techniques, reduce risks associated with them with excessive user privileges, and dramatically improve security detection and response through analytics and automation. The Zero Trust security model treats all applications as internet-facing and considers the entire network compromised and hostile; this assumes that the system is never trusted and delivers only applications and data to authenticated and authorized users. In addition, the system always verifies and never trusts any entities with full logging and behavioral analytics. (Bennett, 2017).

Core components of Zero Trust include:

- Ensure that all resources, regardless of location or hosting model, are securely accessible. (Protection of Data at Rest, 2018)
- Adopting a “least privilege” strategy and strictly enforcing access control
- Inspecting and logging all traffic for suspicious activity (Bennett, 2017)

Data Validation:

Security flaws often exist when an attacker can submit input that exceeds developer assumptions about how the code should work. Remote code execution (RCE), a well-known vulnerability, triggers when a malformed request tries to inject code into an API server-side code and causes it to execute. In this way, an attacker may conduct actions that he would not usually be able to do. Input validation guarantees that data transmitted into the API is valid and secure. Malformed data can be sent to an API by malicious users, resulting in security vulnerabilities. Developers can align their code with the below guidelines for data validation,

- Validate input data for type, length, format, and range.
- Use a whitelist approach to validate input.
- Avoid over-validating.
- Don’t trust user input.
- Sanitize output data. (Kirchoff, 2022)

API Gateways:

If API transmits sensitive information over the wire, all traffic needs strong encryption. Encryption ensures that no unauthorized persons can read data, either during the transit from an API endpoint to its client or after being stored in the database and file system; encryption also ensures that no attacker can change the data. Encryption of transit data can be achieved via Transport Layer Protocol (TLS) while encryption of data at rest via Advance Encryption Standard (AES) as recommended by NIST (Protection of Data at Rest, 2018). The organization needs to follow the best practice of TLS implementation.

Secure code Best Practices Node.js

Broken Object Level Authorization (BOLA):

A Broken Object issue arises when the server fails to correctly check whether the currently logged-in or logged-out user can read, update, or delete an object to which they do not have access.

Vulnerable code:

```
//vulnerable code

app.get("/testhealth.com/vaccine_record/user/download/:record_id", authenticateToken, (req, res) => {
    const { id } = req.params;

    const record = db.records.filter((a) => {
      return a.id === id;
    })[0];

    res.json(record);
  });
```

Code_1: Missing checks and unauthenticated users can download patient vaccine records.
(Source: https://www.stackhawk.com/blog/nodejs-broken-object-level-authorization-guide-examples-and-prevention/)

Secure code

```
//secure code
//Added user ID in the session object,req.session.userId, matches the user-provided value. If there is no match, then deny access and redirect the user.

  app.get("/testhealth.com/vaccine_record/user/download/:record_id", authenticateToken, (req, res) => {
    const { id } = req.params;
    const { user_id } = req.session.userId;

    const record = db.records.filter((a) => {
      return a.id == id && a.user_id == id;
    })[0];

    res.json(record);
  });
```

Code_2: Only authenticated user can download their vaccine records.
(Source: https://www.stackhawk.com/blog/nodejs-broken-object-level-authorization-guide-examples-and-prevention/)

Broken User Authentication:

Secure Code:

```
1
2
3   //Broken User Authentication
4
5   // *** Hashing to store passwords in the database ***
6
7   const bcrypt = require('bcrypt');
8
9   //Generate Hashed Passwords when saving them to db
10
11  userSchema.pre('save', async function(next) {
12    const salt = await bcrypt.genSalt();
13    this.password = await bcrypt.hash(this.password, salt);
14    next();
15  });
16
17  //Validate passwords using bcrypt on login
18  userSchema.statics.login = async function(email, password) {
19    const user = await this.findOne({ email });
20    if (user) {
21      const auth = await bcrypt.compare(password, user.password);
22      if (auth) {
23        return user;
24      }
25      throw Error('incorrect password');
26    }
27    throw Error('incorrect email');
28  };
29
```

Code_3: Secure data at rest.
(Source: https://www.stackhawk.com/blog/nodejs-broken-authentication-guide-examples-and-prevention/)

```
54  // Use Safe and Secure JWT With TTL
55
56  const jwt = require('jsonwebtoken');
57
58  const maxAge = 3 * 24 * 60 * 60;
59  const createToken = (id) => {
60    return jwt.sign({ id }, 'MySecre#$tsalt@#', {
61      expiresIn: maxAge
62    });
63  };
64
65  //login form
66
67  module.exports.LOGIN = async (req, res) => {
68      const { email, password } = req.body;
69
70      try {
71        const user = await User.login(email, password);
72        const token = createToken(user._id);
73        res.cookie('jwt', token);
74        res.status(200).json({ user: user._id });
75      }
76      catch (err) {
77        const errors = handleErrors(err);
78        res.status(400).json({ errors });
79      }
80
81    }
```

```
84    //Use an HttpOnly Cookie With TTL
85
86    res.cookie('jwt', token, { httpOnly: true, maxAge: maxAge * 1000 });
87
```

Code_4: Secure Data in transit.
(Source: https://www.stackhawk.com/blog/nodejs-broken-authentication-guide-examples-and-prevention/)

```
88  //Password strength validation
89  const owasp = require('owasp-password-strength-test');
90
91  //signup form to test password strength
92
93  module.exports.SIGNUP = async (req, res) => {
94      const { email, password } = req.body;
95
96      try {
97        const result = owasp.test(password);
98        if(!result.strong){
99          res.status(401).json({error:result.errors})
100       }
101       //...
102     }
103     catch(err) {
104       //...
105     }
106
107   }
```

Code_5: Strong Password
(Source: https://www.stackhawk.com/blog/nodejs-broken-authentication-guide-examples-and-prevention/)

Unrestricted Resource Consumption:

```
2
3   import rateLimit from 'express-rate-limit';
4
5   export const rateLimiterUsingThirdParty = rateLimit({
6     windowMs: 24 * 60 * 60 * 1000, // 24 hrs in milliseconds
7     max: 200,
8     message: 'You are max out your limit!!!',
9     standardHeaders: true,
10    legacyHeaders: false,
11  });
12
13
```

Code_6: Limiting users on making API calls max 200 in a 24h window.
(Source: https://blog.logrocket.com/rate-limiting-node-js/)

Broken Object Property Level Authorization:

```
//vulnerable code

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const userSchema = new Schema({

    first_name: String,
    last_name: String,
    email: String,
    password: String,
    isAdmin: {
        type: Boolean,
        protect: true,
        default: false
    }

});

const User = mongoose.model("User", userSchema);

module.exports = User;
```

```
//Create a new directory called routes and add the file routes.js to handle user details:

const express = require("express");
    const userModel = require("../models/newUser");

    const app = express();

    app.post("/add_user", async (request, response) => {
        const user = new userModel(request.body);

        try {
          await user.save();
          response.send(user);
        } catch (error) {
          response.status(500).send(error);
        }
    });

module.exports = app;
```

Code_7: A malicious user can make a curl request making himself an admin.
(Source: https://snyk.io/blog/avoiding-mass-assignment-node-js/
https://cheatsheetseries.owasp.org/cheatsheets/Mass_Assignment_Cheat_Sheet.html
https://knowledge-base.secureflag.com/vulnerabilities/inadequate_input_validation/mass_assignment_nodejs.html)

```
curl --location --request POST 'http://test.com/add_user' \
--header 'Content-Type: application/json' \
--data-raw '{
   "first_name": "Mars",
   "last_name": "Venus2023",
   "email": "marsvenus@maliciousmail.com,"
   "isAdmin":"true"
}'
```

Secure Code

```
1
2    const mongoose = require("mongoose");
3
4    // create a schema
5    var userSchema = new mongoose.Schema({
6        first_name: String,
7        last_name: String,
8        email: String,
9        password: String
10
11       //NB: there is no isAdmin field
12
13   });
14
15   const User = mongoose.model("User", userSchema);
16
17   module.exports = User;
18
```

```
22  //Use of pick function to specify the variables extracted from a POST request.
23  //This prevents an attacker from using the sensitive isAdmin field.
24
25  const express = require("express");
26  const userModel = require("../models/newUser");
27  var _ = require('underscore');
28
29  const app = express();
30
31  app.post("/add_user", async (request, response) => {
32      const user = new userModel( _.pick(request.body, 'first_name', 'last_name', 'email', 'password'));
33
34      try {
35        await user.save();
36        response.send(user);
37      } catch (error) {
38        response.status(500).send(error);
39      }
40  });
41
42  module.exports = app;
43
```

Code_8: No sensitive field
(Source: https://snyk.io/blog/avoiding-mass-assignment-node-js/
https://cheatsheetseries.owasp.org/cheatsheets/Mass_Assignment_Cheat_Sheet.html
https://knowledge-base.secureflag.com/vulnerabilities/inadequate_input_validation/mass_assignment_nodejs.html)

Data exposure:

```
//Securing Data in the Browser
//Disabling autocomplete on an entire form

<form method="post" action="/saveForm" autocomplete="off">
 ...
</form>

//Disabling autocomplete on selective form fields containing sensitive data

<form method="post" action="/saveForm">
 <input type="text" id="name" name="Name">
 <input type="text" id="cc" name="Credit Card" autocomplete="off">
 ...
</form>


//Prevent browsers from caching sensitive information

res.set('Cache-Control', 'no-store');
...
res.send(sensitiveData);
```

Code_9: Securing data in the browser

```
 //Securing Data in Transit,creating an HTTPS Node server

 var https = require('https');
 var fs = require('fs');
 var options = {
  key: fs.readFileSync('app/keys/key.pem'),
  cert: fs.readFileSync('app/keys/cert.pem')
 };

 https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end('hello world\n');
 }).listen(8000);

```

Code_9: Securing data in transit
(Source: https://learning.oreilly.com/library/view/securing-node- applications/9781491982426/ch06.html#idm45584120742552
https://github.com/ckarande/securing-node-apps-book-examples/tree/master/chapter6)

Broken Function Level Access Control

Attackers usually use URL manipulation to exploit this vulnerability. Consider the following URLs provided by an application:

test.com/account/view

test.com/account/delete

Although both require authenticated users, let us assume that the /delete endpoint should only be accessible to the admin user. If an unprivileged user can access /delete the endpoint, it is a missing function-level access control flaw.

```
//Broken Function level control Secure code


app.post("/delete", isLoggedIn, isAdmin,
  accountHandler.removeAccount);


//IsAdmin middleware

 module.exports = function isAdmin (req, res, next) {
   if (req.session.userId) {
     userDAO.getUserById(req.session.userId, function(err, user) {
      // Check isAdmin property on user object
      if(user && user.isAdmin) {
       // Allow invoking the action for an admin user
       next();
      } else {
       // Redirect a unprivileged user to the login route
       return res.redirect("/login");
      }
     });
    } else {
     // Redirect an unauthenticated user to the login route
     return res.redirect("/login");
    }
  };

```

Code_10: Using middleware to add access control, isAdmin middleware implementation.

(Source: Securing Node Application, Published by O'Reilly Media, Inc., Author: Chetan Karande)

SQL Injection:

The most common and dangerous vulnerability in web applications is injection vulnerability. Usually, when application code sends untrusted user input into the interpreter as part of a query or command, an injection vulnerability is triggered. Attackers take advantage to create malicious data which deceives the interpreter by performing unauthorized commands or accessing information that has not been adequately authorized.

Vulnerable code:

```
connection.query(
'SELECT * FROM passdb WHERE username ="'
+ req.body.username + '" AND password = "' + passwordHash + '"', function(err, rows, fields) {
  console.log("Result = " + JSON.stringify(rows));
});
```

Code_11: A dynamically constructed SQL query by appending the user-supplied request parameter username, an

An attacker can exploit by entering admin' -- as a username.

(Source: Securing Node Application, Published by O'Reilly Media, Inc., Author: Chetan Karande)

Secure code:

```
var mysql = require('mysql2');
var bcrypt = require('bcrypt-nodejs');
// Prepare query parameters
var username = req.body.username;
var passwordHash = bcrypt.hashSync(req.body.password,
    bcrypt.genSaltSync());
// Connection to the MySQL database
var connection = mysql.createConnection({
    host : 'localhost',
    user : 'superuser',
    password : 'suPerSecret',
    database : 'app_db'
});
connection.connect();
//Prepared statement with parameterized user inputs
var query = 'SELECT * FROM passdb WHERE username=? AND password=?';
connection.query(query, [username, passwordHash],
    function (err, rows, fields) {
    console.log("Results = " + JSON.stringify(rows));
});
connection.end();
```

Code_12: Use of parameterized query.

(Source: Securing Node Application, Published by O'Reilly Media, Inc., Author: Chetan Karande)

Command Injection:

```
child_process.exec(
'gzip ' + req.body.file_path, function (err, data) {
 console.log(data);
});

```

Code_13: Use of child_process.exec method making system calls.

(Source: Securing Node Application, Published by O'Reilly Media, Inc., Author: Chetan Karande

Secure code:

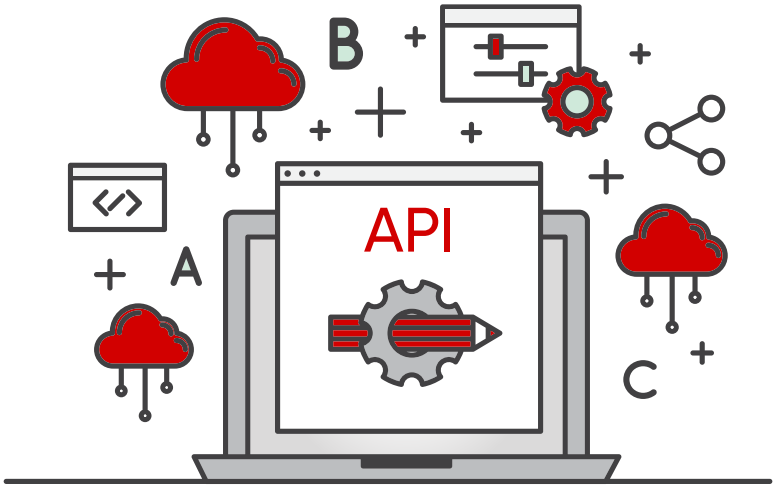
```
// Extract user input from request
var file_path = req.body.file_path;
// Execute gzip command
child_process.execFile( 'gzip',
[file_path],
function (err, data) {
 console.log(data);
});

```

Code_14: Use child_process.execFile instead of childproceess_exec.

(Source: Securing Node Application, Published by O'Reilly Media, Inc., Author: Chetan Karande)

Testing and Monitoring API Security:

Users have a Graphical User Interface for interacting with applications, whereas software uses APIs to interact with these applications and services. Applications will likely make APIs available to the rest of the world, mainly when hosted in a cloud and subsystems used for internal API communication. Developers/Testers have to test the APIs, just as they need to test the GUIs of applications. Addressing bugs at an early stage of the Software development Lifecycle has its advantage; this saves time, money, and possible embarrassment for the organization. Since the success or failure of the software application depends on the robustness of the API, the vendor needs to test all endpoints. The business must properly test all API endpoints. The API can be tested in various ways, each type being designed to test the different aspects of the API.



Unit Testing:

A unit test is a simple atomic test carried out by a developer in the early stages of software development; it is a part of the Continuous Integration and development (CI/CD). Unit tests are created simultaneously while API development is being done. However, they may be written in advance, particularly for organizations adopting test-driven development methodologies.

Integration Testing:

Integration tests, also known as system tests, are similar to unit tests, but instead of working against a single module, they work against actual implementations to check on all dependencies.

Functional Testing:

Functional testing aims to determine whether all application features work in accordance with the software requirements; it is used to execute scenarios from the user's point of view, simulating behavior. Functional tests also include negative tests to ensure functionality remains stable even if invalid values are provided, which identifies potential problems and ensures that the system is not abused.

Performance Testing:

The response time under load is addressed in the performance testing. If multiple requests for that API have been made simultaneously, API endpoints are expected to return a reply per the SLA agreed upon,

Security Testing:

The security test deals with unauthorized access to the API, for example, gaining access to an active session and changing parameters. No unidentified unauthorized user should be allowed to access data in the API. Developers/Testers also look for Error code and message testing to address incorrect input data and how API responds with the appropriate error code and message. (Types of API Testing, 2022)

API Testing Guidelines

OWASP, the Open Web Application Security Project, created the top 10 API Security issues; this section covers testing guidelines on matters mentioned in 2019 and 2023 top 10 API Security.

Authentication and authorization are critical components of many security-related API problems. The test environment with enough user data and mimicking real application permission settings are helpful. APIs can have different authentication implementations, such as HTTP Basic Authentication, where the client passes the username and password or API keys; another is OAuth2.0 bearer access token implementation. To check on API authentication, test all HTTP methods, including HEAD and OPTIONS, along with often used GET, POST, PUT, and DELETE. If the response from a server other than 401 is Unauthorized, something wrong in the code needs an immediate fix.

To test authorization of API is working correctly, the tester needs to create accounts with different access controls for each user, for example, admin with privileged rights and regular user. Check out API documentation on what each user's scope is. Carry out positive and negative tests to check what each user can do and what they are allowed to do and pay attention to error messages; 403 forbidden always gives hints encouraging attackers to brute force paths to confuse attackers, error message 404 Not Found always useful. Another critical consideration tester should look at is JSON structure in API response; Is API equipped with a selection filter, such as a query parameter named "fields," that allows you to pick the fields included in a response? The common problem is that role-based permissions remove an unlawful field from the JSON object, but including the name in the parameter query reveals that field again (Rosenstock, n.d.) If this is the case, please verify this setting as well.

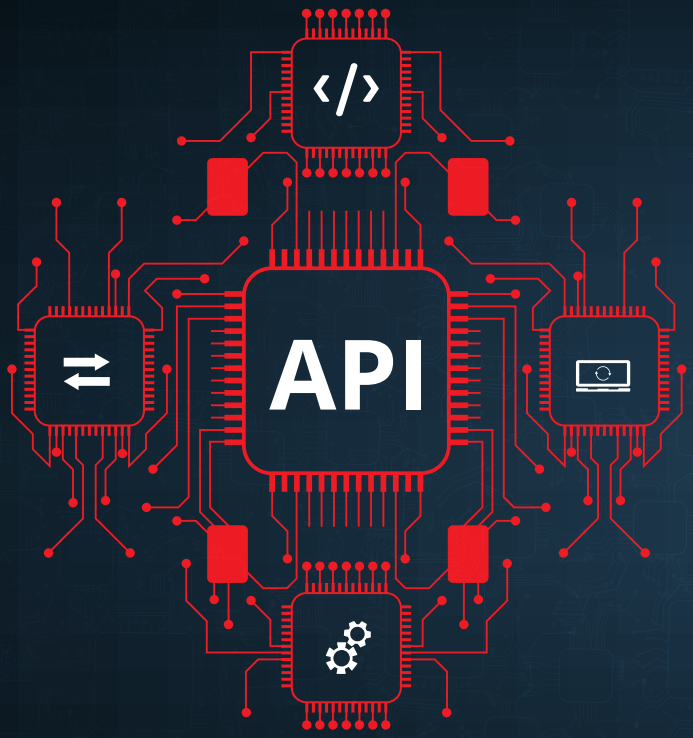
Input validation is determining whether an API satisfies requirements for how it works, how well it runs, how safe it is, and so on. While testing input validation, the tester should have test cases on the API response if a user sends additional fields in the request body besides the expected fields. Security vulnerability often exists in the code when an attacker submits requests that violate your assumption. If you get another HTTP 400 error response, fix the issue immediately. Test for constraints defined for the inputs in the API. For example, pay close attention to various data types and ranges. Carry out negative tests with a list of recommended queries or commands in the OWASP Cheatsheet; remember that the API injection issue is as critical as web applications.

Other Common testing guidelines

- Create a separate test environment that mimics the production environment.
- Carry out functional tests for the happy path first, then automate them with preferred tools.
- Create negative tests for edge scenarios that could lead to security concerns. Begin by checking authentication for a quick win.
- Create detailed documentation for all access control techniques, such as roles and groups. Create test users with a variety of permissions and access to secret resources. Then create test cases in which these users attempt to gain access to unlawful resources.
- Understand back-end architecture and the concerns it is sensitive to and create test cases accordingly to test different scenarios.
- Pay close attention to error responses; they might leak internal information.
- Start security testing in the early phase of the API development, and make sure performance testing is not breaking API security. Always adopt for fail-closed option (Rosenstock, n.d.)

Conclusion

API attacks will be the most common attack vectors in 2022, as predicted earlier by Gartner; there will be no respite from security incident-related API abuse and data breaches in 2023; this will double in 2024, according to Gartner (Talwalkar, 2023) With the increasing number of API attacks, enterprise security requires visibility into all APIs, including public-facing, internal, and unmanaged APIs. The sound practice of API vulnerability mitigations guarantees API compliance, detection, and prevention of API attacks. Organizations need to leverage a collaborative effort from stakeholders, including developers, application owners, and the security team, to understand API threat posture. A practical hands-on approach to understanding the public-facing API footprint to see what an attacker may see is always beneficial in the long run. A thorough inside-out API inventory, including all existing APIs and connections, should be verified with an outward view of APIs and related resources. Analyze existing and new APIs regularly to ensure compliance, maintain high coding quality, consistency, and governance, and scan the entire API inventory for threats; prevention is always better than detection. Last but not least, adopt a continuous testing approach with strong protection controls to secure APIs.



References

Environments and threat models. (2020). In N. MADDEN, API Security in Action. Shelter Island, NY 11964: Manning Publications Co. 2023 Connectivity Benchmark Report. (2023). Retrieved from https://resources.mulesoft.com/ty-report-connectivity-benchmark.html#loaded

Akamai SOTI Report. (2023, April). App and API SOTI report. Retrieved from State of Internet Report: https://www.akamai.com/resources/state-of-the-internet/slipping-through-the-security-gaps-the-rise-of-application-and-api-attacks

Akamai Web Application and API Threat Report. (2022). Retrieved from : https://www.akamai.com/resources/research-paper/akamai-web-application-and-api-threat-report

API Gateway overview. (n.d.). Retrieved from techdocs.akamai.com: https://techdocs.akamai.com/api-definitions/docs/api-gateway-ov

Audit Trails -Chapter 18, NIST. (n.d.). Retrieved from NIST Special Publication 800-12, Introduction to Computer Security: The NIST Handbook.: https://csrc.nist.gov/csrc/media/publications/shared/documents/itl-bulletin/itlbul1997-03.txt

Ball, C. (2022). Hacking APIs. No Starch Press.

Bennett, M. (2017). Zero Trust Security: A CIO's Guide To Defending Their Business From Cyberattacks. Forrester.

Chang, A. (2018, May 2). The Facebook and Cambridge Analytica scandal, is explained with a simple diagram. Retrieved from vox.com: https://www.vox.com/policy-and-politics/2018/3/23/17151916/facebook-cambridge-analytica-trump-diagram

Dionisio Zumerle, J. D. (2022, October 10). Innovation Insight for API Protection. Retrieved from Gartner.com: https://www.gartner.com/doc/reprints?id=1-2BUGX57W&ct=221129&st=sb&alild=eyJpljoiblJuT29wQ0tBM1wvZGhHSnUiLCJ0IjoidkFnSEoyS2FtOUlsN1QzR2RNeWJiZz09In0%253D

Hawkins, M. (2020, June 6). The History And Rise Of APIs. Retrieved from forbes.com: https://www.forbes.com/sites/forbestechcouncil/2020/06/23/the-history-and-rise-of-apis/?sh=1687f4d45c28

How To Address Growing API Security Vulnerabilities In 2022. (2022). Retrieved from https://www.forbes.com/sites/forbestechcouncil/2022/07/25/how-to-address-growing-api-security-vulnerabilities-in-2022/?sh=334878d95a9e

Kirchoff, J. (2022, November 29). 10 REST API Input Validation Best Practices. Retrieved from climbtheladder.com: https://climbtheladder.com/10-rest-api-input-validation-best-practices/

Microsoft Threat Modeling. (n.d.). Retrieved from Security Engineering: https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling

OWASP API Security Project. (2019). Retrieved from https://owasp.org/www-project-api-security/

OWASP Top 10 API. (2023, Feb). Retrieved from OWASP API Security Project: https://github.com/OWASP/API-Security/blob/master/2023/en/src/0xa1-broken-object-level-authorization.md

Protection of Data at Rest. (2018, 2 20). Retrieved from National Institute of Standard and Technology: https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp2089.pdf

Rosenstock, L. (n.d.). How to Test API Security: A Guide and Checklist. Retrieved from traceable.ai: https://www.traceable.ai/blog-post/how-to-test-api-security-a-guide-and-checklist

Start with Security: A Guide for Business. (n.d.). Retrieved from ftc.gov: https://www.ftc.gov/business-guidance/resources/start-security-guide-business

Talwalkar, A. (2023, January 19). 2023 Predictions: Staying One Step Ahead in API Protection. Retrieved from https://www.cequence.ai/blog/api-security/2023-predictions-staying-one-step-ahead-in-api-protection/

Types of API Testing. (2022). In J. Jain, Learn API Testing: Norms, Practices, and Guidelines for Building Effective Test Automation. Apress.

Vincent C. Hu, D. F. (2014, January). NIST Special Publication 800-162 Guide to Attribute Based Access Control (ABAC) Definition and Considerations. Retrieved from NIST Special Publication 800-162: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-162.pdf