Technical Advisory on Secure API Development

What is an API?

Application Programming Interfaces (APIs) are software interfaces that facilitate service communications between two or more applications, enabling developers to access application functionality and to send and receive data using familiar web technologies, particularly Hypertext Transfer Protocol (HTTP), JavaScript Object Notation (JSON), and Extensible Markup Language (XML).

Over the past decade, the growth of API deployments has increased significantly, with API calls accounting for over 80% of all web traffic. Businesses now rely heavily on APIs to build their products and services as APIs are rapidly establishing themselves as the preferred method for building modern applications, especially for mobile devices and the Internet of Things (IoT).

Why Is API Security Important?

APIs, as the gateway to data and systems, pose an ongoing security concern. This is particularly so with the increased adoption of APIs that has led to a widened attack surface, as evidenced by the increase in attacks on API endpoints.

In July 2022, a major social media platform reported a breach that occurred from late 2021 into 2022 and exposed Personally Identifiable Information (PII) of 5.4 million user accounts. The zero-day API vulnerability allowed attackers to submit an email address or phone number, verifying that it is associated with the platform's account, and retrieving the associated account ID. The retrieved account ID was then used to scrape for public information, leaving users vulnerable to hacking, targeted phishing and doxxing. Some of the collected data was sold on the dark web while others were allegedly released for free.

Purpose of Advisory

Developers play a critical role in API and applications security, and are responsible for building secure applications and services. The purpose of this advisory is to provide technical guidance to developers on the secure development of API. The advisory will cover three main sections, **(i) Secure API Design Guidelines**, **(ii) Secure API Development** and **(iii) API Security Testing**.

(i) Secure API Design Guidelines

API design involves the collection of planning and architectural decisions that are made when building an API. This section provides a list of security practices that developers can keep in mind when designing APIs. With the incorporation of security

considerations into the API software development lifecycle (SDLC), organisations can reduce the costs associated with responding to threats and patching security vulnerabilities due to security incidents.

Input Validation

Developers should validate input parameters and use prepared statements, parameterised queries or stored procedures, for each software function to mitigate potential attacks like cross-site scripting (XSS) and Structured Query Language (SQL) injection.

Authentication & Authorisation

Developers should use proven authentication and authorisation mechanisms such as JSON Web Tokens (JWT) to strictly control access to APIs, as they are critical entry points to an organisation's databases.

Encryption

Developers should use Hypertext Transfer Protocol Secure (HTTPS) to ensure all API traffic is encrypted using Transport Layer Security (TLS) to prevent Man-In-The-Middle (MITM) attacks, where any intercepted requests or responses are rendered useless to an attacker without the right decryption method.

Rate Limiting

Developers should conduct rate-limiting to prevent spam behaviours and protect the API from slow performance when too many services or machines are accessing the API for malicious purposes (i.e. Distributed Denial of Service).

Threat Modelling

Developers should conduct threat modelling to identify security vulnerabilities in the API. Threat modelling allows developers to understand and prioritise threats and vulnerabilities during the planning, design, and implementation of APIs. Developers can evaluate the risks associated with each threat based on the likelihood of it occurring, the impact on the system if it does occur, and the ease of exploitation. These potential vulnerabilities should be surfaced to be fixed.

Code Review

Developers should review the API source code to ensure that it adheres to secure coding best practices. Code review is essential for detecting and remediating security vulnerabilities and weaknesses that might be introduced during API development.

Logging and Monitoring Tools

Developers should deploy threat monitoring tools to help track API responses and performance in real-time to detect any anomalous behaviour that could lead to a security breach. The log data could also be used to understand the root cause of the security breach and help in the implementation of any hardening measure.

(ii) Secure API Development

Like any technology, APIs bring about their own unique security challenges and it is important for developers to consider the following during the development phase of APIs. In this section, developers can refer to examples of code snippets, for the section Secure API Design Guidelines.

Usage of JSON Web Token (JWT) for Authentication & Authorisation

Developers can use JWT which is a type of token that is used to authenticate users. It consists of three parts: header, payload, and signature.

- Header

The header contains information about the type of the token and the algorithm that is used to generate the signature. The Header parameters used consist of the hashing algorithm (i.e. HMAC SHA256 or RSA) and the type of the JWT as shown in the example below.

```
// Header
{
 “alg”: “HS256”,
 “typ”: “JWT”
}
```

Header Parameters

- Payload

The payload contains the actual data that is to be passed to the API endpoint. It contains statements about the entity (typically, the user) and additional entity attributes known as claims. It is important to include necessary claims such as “Iss”, “Exp” and “Admin” and validate them to ensure they are not incorrectly modified.

```
// Payload
{
  “Iss”: “Specter” // Contains the details about the logged in user
  “Exp”: 1661941009, // Expiry of token to be used
  “Admin”: false // Boolean describing the role of the user
}
```

Payload Claims

- Signature

The signature is used to verify the authenticity of the token and ensure that the message was not changed along the way. The signature is created by taking in the Base64-encoded header and payload, along with a secret, before being signed with the algorithm specified in the header.

```
// Signature
{
HMACSHA256(
      base64UrlEncode(header) + "." +
      base64UrlEncode(payload),
      secret)
}
```

Base64 Encoding

Input Validation

An example of a PreparedStatement, using Java's implementation of a parameterized query to execute a database query, is shown below:

```
// Prepared Statement
String username = request.getParameter("userName");
String query = "SELECT userName FROM user_data WHERE user_id = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, username);
ResultSet results = pstmt.executeQuery( );
```

Prepared Statements

Encryption

Developers can deploy HTTP with TLS encryption to encrypt/decrypt API requests and responses to/from web servers by creating Secure Sockets Layer (SSL) certificates and private keys using a certificate authority, as shown below:

```
sudo apt update
sudo apt install certbot
sudo certbot certonly --standalone -d httpsdemo.dummyserver.net
```

Create SSL Certificate

Developers can configure SSL Certificate for Node.js HTTPS Server as shown in the example below:

```
const https = require('https');
const fs = require('fs');
const certPath = '/etc/letsencrypt/live/httpsdemo.dummyserver.net';
https.createServer({
    key: fs.readFileSync(`${certPath}/privkey.pem`),
    cert: fs.readFileSync( `${certPath}/fullchain.pem`)
}, (req, res) => { res.end('<h1>Hello World!</h1>') }
).listen(443);
```

HTTPS Configuration

Rate Limiting

Developers can utilise rate limiting as a powerful yet simple solution to secure backend APIs from malicious attacks and handle spam requests from users. It controls the rate at which user requests are being taken in by the server.

An example of rate limiting in Node.js is shown below:

- Use the *express-rate-limit* npm package to limit API requests from a user

```
// src/middlewares/rateLimiter.js

import rateLimit from 'express-rate-limit';

export const rateLimiterUsingThirdParty = rateLimit({
  windowMs: 24 * 60 * 60 * 1000, // 24 hrs in milliseconds
  max: 200, // the number of allowed requests per window per user
  message: 'You exceeded 200 requests in 24 hrs limit!', // message when users exceeded the allowed limit
  standardHeaders: true,
  legacyHeaders: false,
});
```

express-rate-limit package

- Update *index.js* file in the middlewares folder to export the rate limiting middleware from the middlewares module

```
// src/middlewares/index.js

export { default as errorHandler } from './errorHandler';
export { rateLimiterUsingThirdParty } from './rateLimiter';
```

index.js

- Import *rateLimiterUsingThirdParty* middleware and apply it to all application routes:

```
// src/index.js
// Your code here

import { rateLimiterUsingThirdParty } from './middlewares';

// Your code here

app.use(rateLimiterUsingThirdParty);

// Your code here
```

rateLimiter routes

(iii) API Security Testing

API Testing is essential to identify potential vulnerabilities and bugs by checking for security and business logic flaws in an API. Here are three common types of modality testing:

- Unit testing validates each individual function of the API interface involving Create, Read, Update, Delete (CRUD) operations. The automation testing can be performed on JavaScript using the mocha library as shown in the example below:

```
import assert from 'assert';

describe('UserService', () => {

  describe('#create()', () => {
    it('should create a new User instance', () => {
        const email_account = 'testing_user@gmail.com'
        const testing_user = {
          email_account
        }
        assert.equal(testing_user.email, email_account);
    })
  })

})
```

Unit testing

- Functional testing focuses mainly on testing basic usability, mainline functions and ensures the API adheres to security best practices like authentication and authorisation.

```
import assert from 'assert';

describe('#getByUserEmail()', () => {
  it('query by user email', async () => {
      const res = await fetch('http://localhost:5000/api/testing_user?email_account=testing@gmail.com', {
        headers: {
          'Authorization': `Bearer ${wrong API token}`
        }
      })
      assert.equal(res.status, 401); // Unauthorized 401 Error Code
  })
})
```

Functional testing

- Load testing is seen as a subset of functional testing and emphasises the protection of data and resources from attackers. An example of load testing performed on JavaScript using the k6 library is shown below.

```
import http from 'k6/http';

import { sleep } from 'k6';

export default function () {

  http.get('http://localhost:5000/api/testing_user');

  sleep(1);

}
```

Load testing

An example of a k6 test is conducted with 10 virtual users for a duration of 5 seconds as shown below:

```
k6 run --vus 10 --duration 5s load_test.js
```

Running k6 test

Logging and Monitoring

Developers can monitor usage patterns to identify any unusual or suspicious usage patterns, such as repeated requests from a single IP address or an unusual volume of requests, by keeping a record of all API requests and responses. Developers can use popular logging libraries tools on web application frameworks (e.g. Node.js) to conduct activity logging and monitoring.

In the example below, separate loggers are created for different application services, and it is possible to store them on a “centralised.log” file. The configuration shown allows for error logs to be displayed to the console and log any activity that are classified as info level and below, including error and warn logs.

```
//logger.js configuration

const exampleLogger1 = createLogger({
  levels: config.syslog.levels,
   transports: [
       new transports.Console({ level: error }),
       new transports.File({ filename: 'centralised.log', level: 'info' })
    ]
});
const exampleLogger2 = createLogger({
  levels: config.syslog.levels,
   transports: [
       new transports.Console({ level: error }),
       new transports.File({ filename: 'centralised.log', level: 'info' })
       ]
});

module.exports = {
 exampleLogger1: exampleLogger1,
 exampleLogger2: exampleLogger2
};
```

Logging configuration

Penetration Testing

Developers can conduct penetration testing to evaluate API security posture. Penetration testing involves simulating real-world attacks against APIs and identifying security vulnerabilities and weaknesses that threat actors can exploit. This allows developers to remediate security vulnerabilities before they can be exploited by malicious actors. Developers can engage the assistance of white hat security experts to conduct penetration testing to ensure a thorough security assessment of APIs.

Conclusion

The implementation and assessment of API security is important for businesses when building their products and services. Strong API security measures will help deter attackers from infiltrating a system and reduce associated costs when responding to security incidents.

References:

https://thehackernews.com/2023/09/how-to-prevent-api-breaches-guide-to.html

https://www.akamai.com/our-thinking/the-state-of-the-internet

https://venturebeat.com/security/twitter-breach-api-attack/

https://threatpost.com/experian-api-leaks-american-credit-scores/165731/

https://onboardbase.com/blog/api-security-best-practices/