NAME: Justin Ann Bernante

Course/Yr/Sec: bsit -2B

**1.Describe the differences between authentication and authorization, and explain why they are both important in web application security.**

Authentication and authorization are two critical concepts in web application security that are often used together, but they have distinct differences and purposes.

Authentication refers to the process of verifying the identity of a user, device, or application. It confirms that the user is who they claim to be and ensures that the user is authorized to access the system. On the other hand, authorization refers to the process of determining what level of access a user has to a particular resource or function. Authorization is based on the identity of the user and the permissions associated with that identity. It defines what the user is allowed to do within the system once they have been authenticated.

**2.What are some common techniques used for authentication and authorization in Node.js applications? Provide examples of each.**

Authentication refers to the process of verifying a user's identity, while authorization refers to granting or denying access to specific resources or functionalities based on the authenticated user's permissions.Some common techniques used for authentication and authorization in Node.js applications include:

JSON Web Tokens (JWT) - JWT is a popular authentication technique that uses a JSON-based token to authenticate users. Once a user logs in successfully, a JWT token is generated and stored on the client-side. This token is then sent with each subsequent request to the server, and the server verifies its authenticity before allowing access to protected resources.

Example:

javascript

Copy code

const jwt = require('jsonwebtoken');

// Generate JWT token

const token = jwt.sign({userId: user.id}, 'secret\_key', { expiresIn: '1h' });

// Verify JWT token

jwt.verify(token, 'secret\_key', (err, decoded) => {

if (err) {

// Token is invalid

} else {

// Token is valid

}

});

Passport.js - Passport is a popular authentication middleware for Node.js that supports a wide range of authentication strategies, including local username and password, OAuth, OpenID, and more. It provides a simple, flexible, and modular approach to authentication and can be easily integrated into any Node.js application.

Example:

javascript

Copy code

const passport = require('passport');

const LocalStrategy = require('passport-local').Strategy;

passport.use(new LocalStrategy(

function(username, password, done) {

User.findOne({ username: username }, function (err, user) {

if (err) { return done(err); }

if (!user) { return done(null, false); }

if (!user.verifyPassword(password)) { return done(null, false); }

return done(null, user);

});

}

));

Role-based access control (RBAC) - RBAC is a common authorization technique that grants access to specific resources or functionalities based on a user's role or permission level. Each user is assigned a role or permission level, and access is granted or denied based on that role or permission level.

Example:

javascript

Copy code

const userRoles = {

ADMIN: 'admin',

USER: 'user',

};

// Middleware to check if user has admin role

function isAdmin(req, res, next) {

if (req.user.role === userRoles.ADMIN) {

next();

} else {

res.status(403).json({ message: 'Access denied' });

}

}

These are just a few examples of the many authentication and authorization techniques available for Node.js applications. The specific techniques used will depend on the application's requirements, security needs, and user base.

**3.Explain what JSON Web Tokens (JWTs) are and how they are used in Node.js applications. How do they improve security compared to other authentication methods**?

JSON Web Token (JWT) is a standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. The compact size makes the tokens easy to transfer through an URL, POST parameter, or inside an HTTP header.

Since tokens like JWT are stateless, only a secret key can validate it when received at a server-side application, which was used to create it. Hence they're considered the best and the most secure way of offering authentication.

**4.What is OAuth 2.0, and how is it used for authentication and authorization in Node.js applications? Describe the different OAuth 2.0 flows and how they are used in practice**.

OAuth 2.0 is an authorization protocol and NOT an authentication protocol. As such, it is designed primarily as a means of granting access to a set of resources, for example, remote APIs or user data. OAuth 2.0 uses Access Tokens.

Resource Owner: Entity that can grant access to a protected resource. Typically, this is the end-user.

Client: Application requesting access to a protected resource on behalf of the Resource Owner.

Resource Server: Server hosting the protected resources. This is the API you want to access.

Authorization Server: Server that authenticates the Resource Owner and issues Access Tokens after getting proper authorization. In this case, Auth0.

User Agent: Agent used by the Resource Owner to interact with the Client (for example, a browser or a native application).

**5.What is role-based access control (RBAC), and how is it used for authorization in Node.js applications? Provide an example of how you might use RBAC to control access to a specific resource or functionality.**

A Role-based access control (RBAC) is a security approach that restricts network access and assigns permissions to users based on their role within an organization.

In Node.js applications, RBAC can be implemented using middleware functions that check the user's role and permissions before allowing access to specific resources or functionality. These middleware functions can be added to the routes or endpoints that require authorization, and they can be used to restrict access to specific users or groups.

For example, let's say we have a Node.js application that allows users to create, read, update, and delete articles. We want to restrict access to the article deletion functionality to only certain users with the "admin" role. To implement this using RBAC, we can define a middleware function that checks the user's role and only allows access if the user has the "admin" role:

function requireAdmin(req, res, next) {

if (req.user.role === 'admin') {

next(); // allow access to the route handler

} else {

res.status(403).send('Access denied'); // return a 403 error for unauthorized users

}

}

We can then add this middleware function to the route handler for the delete article functionality:

go

Copy code

app.delete('/articles/:id', requireAdmin, deleteArticleHandler);

Now, only users with the "admin" role will be able to access the delete article functionality, while other users will receive a 403 error.

**6.Describe the different techniques used for secure password storage in Node.js applications, Secure Password Storage.**

To prevent hackers from stealing or compromising user credentials, secure password storage is essential. Hashing, which transforms a password into a fixed-length string of characters, is one of the safe password storage methods provided by Node.js. This method creates a distinct hash for each password using a one-way hashing algorithm, such as bcrypt or Argon2. The database stores the hash rather than the password in plain text. The system hashes the user's password when they log in and contrasts it with the previously saved hash. The user is authenticated if they match. Even if the database is compromised, attackers will not be able to recover the original passwords thanks to hashing.

Salting: Salting is a technique used to make it harder to guess passwords by adding random data (salt) to the password before hashing it. This random data is unique for each user, and it makes the generated hash different for every password, even if they are the same. Salted hashes make it more difficult for attackers to use pre-computed hash tables or rainbow tables to crack passwords.

Key stretching: Key stretching is a technique used to increase the time and resources required to brute force a password. It involves applying a computationally expensive function multiple times to the password hash, making it more difficult to crack. The bcrypt and Argon2 algorithms use key stretching as part of their hashing process.

**7.What is two-factor authentication (2FA), and how can it be implemented in Node.js applications? Explain the benefits and potential drawbacks of 2FA.**

Two-factor authentication (2FA) is an identity and access management security method that requires two forms of identification to access resources and data. 2FA gives businesses the ability to monitor and help safeguard their most vulnerable.

Implementing 2FA in Node.js applications involves integrating a secondary authentication factor into the login process. This can be done through various methods such as sending a one-time password (OTP) via SMS, email, or a mobile app, or using biometric authentication like fingerprint or facial recognition.

The benefits of implementing 2FA in Node.js applications are:

The main advantage of two-factor authentication is the increased login security. As for the shortcomings, the main two being the increase in the time of entry into the system and the risk of losing the physical media serving to pass one of the authentication steps (mobile phone, U2F key, OTP-token).

**8.How can session management be used to improve authentication and authorization in Node.js applications? Describe some common techniques used for session management, and explain why they are important for security.**

In order to ensure that only users with permission can access the program's protected resources, session management is a crucial component of web application security. The usage of session management in a Node.js application can enhance authentication and authorisation by allowing the server to recognize and validate the user's identity across numerous requests.

Using cookies to store session data on the client-side is one typical method of session management in Node.js. The server creates a session ID and delivers it to the client in a cookie when a user signs in. The server can then match the request with the appropriate session data because the client includes the session ID with each subsequent request. This strategy is popular because it's straightforward and These techniques are crucial for security since they prohibit unauthorized access to protected resources by confirming the user's identity and authorization status on each request. They are compatible with the majority of web browsers. As a result, the risk of session hijacking and other attacks is decreased. It also allows the server to end sessions if the user logs out or if there is suspicious behaviour.

**9.Describe a specific security vulnerability that could be exploited in a Node.js application that does not properly handle authentication or authorization. How might an attacker exploit this vulnerability, and what steps can be taken to prevent it?**

 "Session Fixation" attack is one specific security flaw that could be exploited in a Node.js application that improperly handles login or authorisation.

In a specific kind of attack known as session fixation, the attacker modifies the session ID of the victim to one that is familiar to them. If the Node.js application reuses the session ID after successful authentication rather than generating a new one, this can take place. The victim could be duped into using the attacker's session ID in order to take advantage of this vulnerability.Several steps can be done to prevent the session fixation vulnerability:

After successful user authentication, use a different session ID: The Node.js application should create a new session ID and invalidate the previous one. This would make sure that attackers couldn't hijack a user's session using a known session ID.

**10.Explain why it is important to regularly review and update the authentication and authorization mechanisms used in a Node.js application. What are some common issues that can arise if these mechanisms are not kept up-to-date?**

The security and integrity of the system must be regularly monitored and updated in a Node.js application's authentication and authorisation procedures. Mechanisms for authentication and permission aid in securing sensitive data and preventing unauthorized access to the system.