**Part 1 of IA 2**

Name: Rachit Shah Date: 31st March 2025

1) Identify common security threats for HTTP applications.

🡺

|  |
| --- |
| Common security threats for HTTP applications include   * Cross-Site Scripting (XSS) * SQL injection * Cross-Site Request Forgery (CSRF) * security misconfigurations * broken access control * vulnerable and outdated components * cryptographic failures |

2) Implement security measures such as HTTPS, CORS, and input validation in your HTTP service from Part 1.

🡺

|  |
| --- |
| **HTTPS (Hypertext Transfer Protocol Secure)**   * Encrypts data transmitted between the client and the server using **TLS (Transport Layer Security)**. * Prevents **man-in-the-middle (MITM) attacks**, eavesdropping, and data tampering. * Helps protect sensitive user information like passwords, payment details, and personal data. * Improves trustworthiness and SEO rankings.   **CORS (Cross-Origin Resource Sharing)**   * Controls which domains can access resources from your server, reducing **cross-origin attacks**. * Prevents **unauthorized access** to APIs from malicious websites. * Helps mitigate **Cross-Site Request Forgery (CSRF)** attacks when used with proper token-based authentication.   **Input Validation**   * Prevents **SQL Injection (SQLi), Cross-Site Scripting (XSS), and Command Injection** attacks. * Ensures that user input matches the expected format (e.g., email format, number ranges). * Reduces the risk of **buffer overflows and unexpected behaviour** in the application. * Helps maintain **data integrity** and improves overall system security.   1) Create and add certificates to system  2) .env //add this file in frontend  HTTPS=true  SSL\_CRT\_FILE=../backend/localhost.crt  SSL\_KEY\_FILE=../backend/localhost.key  3) Server.js  import express from "express";  import bcrypt from "bcryptjs";  import cors from "cors";  import { MongoClient } from "mongodb";  import fs from "fs";  import https from "https";  const app = express();  app.use(express.json());  app.use(cors());  // Load SSL certificate and key  const httpsOptions = {    key: fs.readFileSync("localhost.key"),    cert: fs.readFileSync("localhost.crt")  };  // MongoDB Connection  const mongoURI = "mongodb+srv://shreyjce22:shrey@cluster0.cnicu.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0";  const client = new MongoClient(mongoURI);  let usersCollection;  // Connect to MongoDB  async function connectDB() {    try {      await client.connect();      const db = client.db("test");      usersCollection = db.collection("users");      console.log("Connected to MongoDB Atlas");    } catch (error) {      console.error("MongoDB connection error:", error);    }  }  connectDB();  app.post("/api/login", async (req, res) => {    const { username, password } = req.body;    try {      const user = await usersCollection.findOne({ username });      if (!user) return res.status(400).json({ message: "User not found!" });      const isMatch = await bcrypt.compare(password, user.password);      if (!isMatch) return res.status(400).json({ message: "Invalid credentials!" });      res.json({ message: "Login successful!" });    } catch (error) {      res.status(500).json({ message: "Server error!" });    }  });  // Start HTTPS Server  https.createServer(httpsOptions, app).listen(5000, () => {    console.log("HTTPS Server running on https://localhost:5000");  }); |

3) Demonstrate the use of security headers (e.g., Content Security Policy, X-Frame-Options).

🡺

|  |
| --- |
| The HTTP X-Frame-Options response header can be used to indicate whether a browser should be allowed to render a page in a <frame>, <iframe>, <embed> or <object>. Sites can use this to avoid clickjacking attacks, by ensuring that their content is not embedded into other sites.  **Content-Security-Policy (CSP)**   * Restricts allowed sources for scripts, styles, and other content. * Prevents **XSS attacks** by blocking malicious inline scripts. * Example: "default-src 'self'; script-src 'self'" allows only scripts from the same origin.   **X-Frame-Options**   * Protects against **clickjacking attacks** by preventing embedding in iframes. * "DENY" completely blocks embedding, while "SAMEORIGIN" allows embedding from the same site.   **X-Content-Type-Options**   * Prevents browsers from **MIME-type sniffing**, reducing the risk of security bypasses. * "nosniff" ensures the correct Content-Type is enforced.   **Referrer-Policy**   * Controls how much referrer information is shared when navigating to another site. * "no-referrer" prevents sending the referrer header entirely.   **Strict-Transport-Security (HSTS)**   * Forces browsers to only access the site over HTTPS. * "max-age=31536000; includeSubDomains; preload" ensures all subdomains use HTTPS for one year   Use “helmet” package in backend   * Blocks inline JavaScript and only allows scripts from the same origin. * Clickjacking attacks by controlling whether the page can be embedded in an <iframe> * MIME-type sniffing attacks where browsers incorrectly interpret non-JavaScript files as scripts. * Referrer leakage to external sites, protecting user privacy. * Downgrade attacks where attackers trick users into using HTTP instead of HTTPS.   Server.js  import express from "express";  import bcrypt from "bcryptjs";  import cors from "cors";  import helmet from "helmet"; // Import Helmet  import { MongoClient } from "mongodb";  import fs from "fs";  import https from "https";  const app = express();  app.use(express.json());  app.use(cors());  app.use(helmet()); // Enable security headers  const httpsOptions = {    key: fs.readFileSync("localhost.key"),    cert: fs.readFileSync("localhost.crt"),  };  const mongoURI = "mongodb+srv://shreyjce22:shrey@cluster0.cnicu.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0";  const client = new MongoClient(mongoURI);  let usersCollection;  async function connectDB() {    try {      await client.connect();      const db = client.db("test");      usersCollection = db.collection("users");      console.log("Connected to MongoDB Atlas");    } catch (error) {      console.error("MongoDB connection error:", error);    }  }  connectDB();  app.post("/api/login", async (req, res) => {    const { username, password } = req.body;    try {      const user = await usersCollection.findOne({ username });      if (!user) return res.status(400).json({ message: "User not found!" });      const isMatch = await bcrypt.compare(password, user.password);      if (!isMatch) return res.status(400).json({ message: "Invalid credentials!" });      res.json({ message: "Login successful!" });    } catch (error) {      res.status(500).json({ message: "Server error!" });    }  });  https.createServer(httpsOptions, app).listen(5000, () => {    console.log("HTTPS Server running on https://localhost:5000");  }); |

4) Discuss how SOAP services handle security differently from RESTful services.

🡺

|  |  |
| --- | --- |
| **SOAP Security (More Secure)**   * **Built-in WS-Security**: Uses **XML-Encryption** and **XML-Signature** for secure messaging. * **Authentication**: Supports **UsernameToken** and **SAML Tokens** for identity verification. * **Message Security**: Provides **end-to-end encryption**, independent of transport layer security. * **Data Integrity**: Uses **XML-Signature** to ensure message authenticity and prevent tampering. * **Reliability**: Implements **WS-ReliableMessaging** for guaranteed message delivery. * **Best for**: **Banking, Healthcare, Government, Enterprise Applications** requiring high security. | **REST Security (More Flexible)**   * **No built-in security**: Relies on external mechanisms like **OAuth, JWT, and HTTPS**. * **Authentication**: Uses **OAuth 2.0, JWT (JSON Web Token), API Keys** for access control. * **Message Security**: Security is applied at the transport layer using **TLS/SSL (HTTPS)**. * **Data Integrity**: Ensured through **TLS encryption** during transmission. * **Reliability**: No built-in message reliability mechanism. * **Best for**: **Web & Mobile Apps, Cloud Services, Public APIs**, where scalability is key. |

5) Explain identity management concepts in web services (OAuth, OpenID Connect, SAML). Implement OAuth 2.0 authentication in your web service (Part 1) to restrict access.

🡺

|  |
| --- |
| **OAuth 2.0 (Authorization)**   * **Purpose**: Grants third-party apps limited access to a user's resources without exposing credentials. * **How It Works**: Uses **access tokens** issued by an **authorization server** to access protected resources. * **Example**: When you log in to a website using Google or Facebook instead of creating a new account.   **🔹 OpenID Connect (OIDC) (Authentication)**   * **Purpose**: An identity layer on top of OAuth 2.0 that provides **authentication**. * **How It Works**: Uses **ID tokens** to confirm user identity, in addition to OAuth's access tokens. * **Example**: Google Sign-In.   **🔹 SAML (Security Assertion Markup Language)**   * **Purpose**: Used in enterprise settings for **single sign-on (SSO)**. * **How It Works**: Uses XML-based authentication assertions between identity providers and service providers. * **Example**: Logging into multiple corporate apps with one set of credentials.   server.js  import express from "express";  import bcrypt from "bcryptjs";  import cors from "cors";  import session from "express-session";  import passport from "passport";  import { Strategy as GoogleStrategy } from "passport-google-oauth20";  import { MongoClient } from "mongodb";  import dotenv from "dotenv";  dotenv.config();  const app = express();  app.use(express.json());  app.use(cors({ origin: "http://localhost:3000", credentials: true }));  app.use(    session({      secret: process.env.SESSION\_SECRET,      resave: false,      saveUninitialized: true,    })  );  app.use(passport.initialize());  app.use(passport.session());  const mongoURI = process.env.MONGO\_URI;  const client = new MongoClient(mongoURI);  let usersCollection;  async function connectDB() {    try {      await client.connect();      const db = client.db("test");      usersCollection = db.collection("users");      console.log("Connected to MongoDB Atlas");    } catch (error) {      console.error("MongoDB connection error:", error);    }  }  await connectDB();  passport.use(    new GoogleStrategy(      {        clientID: process.env.GOOGLE\_CLIENT\_ID,        clientSecret: process.env.GOOGLE\_CLIENT\_SECRET,        callbackURL: "http://localhost:5000/auth/google/callback",      },      async (accessToken, refreshToken, profile, done) => {        try {          let user = await usersCollection.findOne({ googleId: profile.id });          if (!user) {            user = await usersCollection.insertOne({              googleId: profile.id,              username: profile.displayName,              email: profile.emails[0].value,            });          }          return done(null, user);        } catch (error) {          return done(error, null);        }      }    )  );  passport.serializeUser((user, done) => done(null, user.\_id));  passport.deserializeUser(async (id, done) => {    const user = await usersCollection.findOne({ \_id: id });    done(null, user);  });  app.get("/auth/google", passport.authenticate("google", { scope: ["profile", "email"] }));  app.get(    "/auth/google/callback",    passport.authenticate("google", { failureRedirect: "/login" }),    (req, res) => {      res.redirect("http://localhost:3000");    }  );  app.post("/api/login", async (req, res) => {    const { username, password } = req.body;    console.log("Received login request for username:", username);    try {      const user = await usersCollection.findOne({ username });      console.log("Database user result:", user);      if (!user) return res.status(400).json({ message: "User not found!" });      const isMatch = await bcrypt.compare(password, user.password);      if (!isMatch) return res.status(400).json({ message: "Invalid credentials!" });      res.json({ message: "Login successful!" });    } catch (error) {      res.status(500).json({ message: "Server error!" });    }  });  app.listen(5000, () => console.log("Server running on port 5000"));  add a .env file which contains MONGO\_URI, GOOGLE\_CLIENT\_ID, GOOGLE\_CLIENT\_SECRET  and SESSION\_SECRET. (Client Id and Client Secret are generated from Google cloud Console). |

6) Explain different authorization patterns used in web services. Implement a role-based access control (RBAC) mechanism for your web service.

🡺

|  |
| --- |
| Authorization is the process of granting or restricting access to resources based on user roles, permissions, or policies. Here are some common authorization patterns used in web services:   1. Role-Based Access Control (RBAC)   Users are assigned specific roles (e.g., Admin, User, Moderator). Each role has predefined permissions to access certain resources. Example: An admin can delete users, while a regular user cannot.   1. Attribute-Based Access Control (ABAC)   Access is granted based on attributes (e.g., user age, department, time of access). Example: A finance report can only be accessed by employees from the finance department.   1. Discretionary Access Control (DAC)   The owner of a resource decides who can access it. Example: A user who uploads a file can decide who else can view or edit it.   1. Mandatory Access Control (MAC)   Access is controlled based on predefined security labels assigned to users and resources. Commonly used in military and government systems.   1. OAuth Authorization   Uses OAuth 2.0 to delegate access to resources without sharing credentials. Example: Logging in to a third-party website using Google or Facebook.  Steps to Implement RBAC   1. Assign roles to users in the database.    * Extend User Schema with Role 2. Implement middleware to check if a user has the required role.    * Create RBAC Middleware 3. Protect specific routes based on user roles.    * Protect Admin Routes |
| **server.js**  import express from "express";  import bcrypt from "bcryptjs";  import cors from "cors";  import session from "express-session";  import passport from "passport";  import { Strategy as GoogleStrategy } from "passport-google-oauth20";  import { MongoClient, ObjectId } from "mongodb";  import dotenv from "dotenv";  dotenv.config();  const app = express();  app.use(express.json());  app.use(cors({ origin: "http://localhost:3000", credentials: true }));  app.use(    session({      secret: process.env.SESSION\_SECRET,      resave: false,      saveUninitialized: true,    })  );  app.use(passport.initialize());  app.use(passport.session());  const mongoURI = process.env.MONGO\_URI;  const client = new MongoClient(mongoURI);  let usersCollection;  async function connectDB() {    try {      await client.connect();      const db = client.db("test");      usersCollection = db.collection("users");      console.log("Connected to MongoDB Atlas");    } catch (error) {      console.error("MongoDB connection error:", error);    }  }  await connectDB();  passport.use(    new GoogleStrategy(      {        clientID: process.env.GOOGLE\_CLIENT\_ID,        clientSecret: process.env.GOOGLE\_CLIENT\_SECRET,        callbackURL: "http://localhost:5000/auth/google/callback",      },      async (accessToken, refreshToken, profile, done) => {        try {          let user = await usersCollection.findOne({ googleId: profile.id });          if (!user) {            user = await usersCollection.insertOne({              googleId: profile.id,              username: profile.displayName,              email: profile.emails[0].value,              role: "user",            });            user = await usersCollection.findOne({ googleId: profile.id });          }          return done(null, user);        } catch (error) {          return done(error, null);        }      }    )  );  passport.serializeUser((user, done) => done(null, user.\_id));  passport.deserializeUser(async (id, done) => {    const user = await usersCollection.findOne({ \_id: new ObjectId(id) });    done(null, user ? { \_id: user.\_id, role: user.role } : null);  });  const requireRole = (role) => {    return (req, res, next) => {      if (!req.isAuthenticated() || req.user.role !== role) {        return res.status(403).json({ message: "Access denied!" });      }      next();    };  };  app.get("/auth/google", passport.authenticate("google", { scope: ["profile", "email"] }));  app.get(    "/auth/google/callback",    passport.authenticate("google", { failureRedirect: "/login" }),    (req, res) => {      res.redirect("http://localhost:3000");    }  );  app.post("/api/login", async (req, res) => {    const { username, password } = req.body;    console.log("Received login request for username:", username);    try {      const user = await usersCollection.findOne({ username });      console.log("Database user result:", user);      if (!user) return res.status(400).json({ message: "User not found!" });      const isMatch = await bcrypt.compare(password, user.password);      if (!isMatch) return res.status(400).json({ message: "Invalid credentials!" });      req.session.user = user;      res.json({ message: "Login successful!" });    } catch (error) {      res.status(500).json({ message: "Server error!" });    }  });  app.get("/admin", requireRole("admin"), (req, res) => {    res.json({ message: "Welcome, Admin!" });  });  app.get("/profile", (req, res) => {    if (!req.isAuthenticated()) return res.status(401).json({ message: "Unauthorized" });    res.json({ user: req.user });  });  app.get("/logout", (req, res) => {    req.logout(() => {      res.json({ message: "Logged out successfully!" });    });  });  app.listen(5000, () => console.log("Server running on port 5000")); |

7) Implement WS-Security for a sample SOAP service, ensuring message integrity and confidentiality.

🡺

|  |
| --- |
| **service.wsdl**  <definitions name="WSService"      targetNamespace="http://example.com/ws-security"      xmlns:tns="http://example.com/ws-security"      xmlns:xsd="http://www.w3.org/2001/XMLSchema"      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"      xmlns="http://schemas.xmlsoap.org/wsdl/">      <message name="GetDataRequest">          <part name="username" type="xsd:string"/>          <part name="password" type="xsd:string"/>          <part name="input" type="xsd:string"/>      </message>      <message name="GetDataResponse">          <part name="result" type="xsd:string"/>      </message>      <portType name="WSServicePortType">          <operation name="GetData">              <input message="tns:GetDataRequest"/>              <output message="tns:GetDataResponse"/>          </operation>      </portType>      <binding name="WSServiceBinding" type="tns:WSServicePortType">          <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>          <operation name="GetData">              <soap:operation soapAction="http://example.com/ws-security/GetData"/>              <input>                  <soap:body use="literal"/>              </input>              <output>                  <soap:body use="literal"/>              </output>          </operation>      </binding>      <service name="WSService">          <port name="WSServicePort" binding="tns:WSServiceBinding">              <soap:address location="http://localhost:8000/ws-security"/>          </port>      </service>  </definitions>  **server.js**  import express from "express";  import soap from "soap";  import fs from "fs";  import path from "path";  import { DOMParser } from "xmldom";  const app = express();  const PORT = 8000;  const wsdlPath = path.join(process.cwd(), "service.wsdl");  const wsdl = fs.readFileSync(wsdlPath, "utf8");  const users = {    "admin": "password123",    "user": "userpass"  };  const service = {    WSService: {      WSServicePortType: {        GetData: (args, callback) => {          console.log("Received Request:", args);          const securityHeader = args.Security;          if (!securityHeader) {            return callback({ message: "Missing WS-Security Header" });          }          const { Username, Password } = securityHeader;          if (!users[Username] || users[Username] !== Password) {            return callback({ message: "Authentication Failed" });          }          const input = args.input;          const result = input.split("").reverse().join("");          callback(null, { result });        },      },    },  };  soap.listen(app, "/ws-security", service, wsdl, () => {    console.log(`SOAP service running on http://localhost:${PORT}/ws-security`);  });  app.listen(PORT, () => {    console.log(`Express server started on port ${PORT}`);  });  **client.js**  import soap from "soap";  const url = "http://localhost:8000/ws-security?wsdl";  const args = {    Security: { Username: "admin", Password: "password123" },    input: "HelloWorld",  };  soap.createClient(url, (err, client) => {    if (err) {      console.error("SOAP Client Error:", err);      return;    }    client.GetData(args, (err, result) => {      if (err) {        console.error("SOAP Request Error:", err);        return;      }      console.log("SOAP Response:", result);    });  });  **SOAP-based web service** using **WS-Security authentication** to provide message integrity and confidentiality.   * **SOAP Service (server.js)**   + Created a **SOAP server** using the soap module.   + Defined a GetData function that **reverses the input string** as an example.   + **Validated WS-Security headers** (username & password) before processing the request. * **WSDL Definition (service.wsdl)**   + Defined the **SOAP service** structure, including GetData operation.   + Specified **WS-Security headers** for authentication. * **SOAP Client (client.js)**   + Created a **SOAP client** to send a request with **WS-Security authentication**.   + Sent a request with **username, password, and input data**.   + Printed the **response** after authentication and processing.   **How Does WS-Security Work Here?**   1. **Client sends a SOAP request** with a security header (username & password). 2. **Server verifies the credentials** against predefined users. 3. If authentication **fails**, it returns an error. 4. If authentication **succeeds**, the server **processes the request** and returns a response.   **Example Scenario**  **Client Request:**   * Sends username: "admin" * Sends password: "password123" * Sends input: "HelloWorld"   **Server Processing:**   * Checks WS-Security credentials. * Reverses "HelloWorld" → "dlroWolleH". * Returns "dlroWolleH" as the response. |

Project Repository:

GitHub - <https://github.com/rachitshah-1/CyberSec-IA>

Reference:

1. [https://auth0.com/blog/cors-tutorial-a-guide-to-cross-origin-resource-sharing](https://auth0.com/blog/cors-tutorial-a-guide-to-cross-origin-resource-sharing/)
2. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/X-Frame-Options>
3. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CSP>
4. <https://developers.google.com/identity/protocols/oauth2/javascript-implicit-flow>
5. <https://medium.com/@Mohd_Aamir_17/implementing-role-based-access-control-rbac-in-java-987bcc393739>
6. <https://www.ibm.com/docs/en/was-liberty/base?topic=wssml-securing-web-service-by-using-ws-security-policy>