CYBER IA-II
PART II

1	**Identify common security threats for HTTP applications.**
	HTTP applications are prone to various security threats, including: **Cross-Site Scripting (XSS):** Malicious scripts are injected into trusted websites. **SQL Injection:** Attackers inject SQL queries through user inputs to manipulate the database. **Cross-Site Request Forgery (CSRF):** Unauthorized commands are transmitted from a user that the application trusts. **Man-in-the-Middle (MitM) Attacks:** Attackers intercept communication between client and server. **Session Hijacking:** Stealing or manipulating user session tokens. **Insecure Direct Object References (IDOR)**: Gaining unauthorized access to resources by manipulating URLs or request parameters. **Brute Force Attacks**: Attempting to gain access by trying many passwords or tokens. **Denial of Service (DoS):** Overloading the server with requests to make the service unavailable.

MALWARE
PHISHING
RANSOMWARE
DENIAL-OF-SERVICE (DDOS) ATTACKS
MAN-IN-THE-MIDDLE (MITM) ATTACKS
ZERO-DAY ATTACKS
SOCIAL ENGINEERING
INSIDER THREATS
What are the 8 main cyber security threats?

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

Security Measures in Your HTTP Service:

1. HTTPS – Ensures encrypted data transmission, safeguarding against Man-in-the-Middle (MITM) attacks.

2. CORS (Cross-Origin Resource Sharing) – Controls domain access to your API, preventing unauthorized cross-origin requests.

3. Input Validation – Protects against SQL injection and XSS by sanitizing and validating user inputs.

```
const https = require('https');

const fs = require('fs');

const express = require('express');
const app = express();

// Load SSL certificate and key
const options = {
  key: fs.readFileSync('server.key'),
  cert: fs.readFileSync('server.cert'),
};

// Start HTTPS server
https.createServer(options, app).listen(443, () => {
  console.log('Secure server running on https://localhost');
});
```

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

Security Measures in Your HTTP Service:

1. HTTPS Encryption – Ensures encrypted data transmission, safeguarding against Man-in-the-Middle (MITM) attacks.
2. CORS Policy – Controls domain access to your API, preventing unauthorized cross-origin requests.
3. Input Validation – Protects against SQL injection and XSS by sanitizing and validating user inputs.

4. Content Security Policy (CSP) – Restricts the sources of executable scripts to enhance security.
5. X-Frame-Options Header – Prevents clickjacking attacks by restricting iframe embedding.

6. X-XSS-Protection Mechanism – Helps mitigate XSS attacks by blocking malicious scripts.
7. Strict-Transport-Security (HSTS) – Enforces HTTPS usage to prevent protocol downgrade attacks.

```
const express = require('express');
const helmet = require('helmet');

const app = express();
const PORT = process.env.PORT || 4000;

// Step 1: Use Helmet for common security headers
app.use(helmet());

// Step 2: Configure a custom Content Security Policy (CSP)
// This ensures only content from your domain (and allowed sources) is loaded.
app.use(
  helmet.contentSecurityPolicy({
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'", "cdnjs.cloudflare.com"],  // Allow scripts from your domain and CDNJS
      styleSrc: ["'self'", "fonts.googleapis.com"],     // Allow styles from your domain and Google Fonts
      fontSrc: ["'self'", "fonts.gstatic.com"],         // Allow fonts from your domain and Google Fonts
      imgSrc: ["'self'", "data:"],                      // Allow images from your domain and inline images
    },
  })
);

// Step 3: Set additional security headers via custom middleware
app.use((req, res, next) => {
  // Prevent clickjacking by disallowing framing
  res.setHeader("X-Frame-Options", "DENY");

  // Prevent browsers from MIME sniffing the content type
  res.setHeader("X-Content-Type-Options", "nosniff");

  // Enable basic XSS filtering in browsers
  res.setHeader("X-XSS-Protection", "1; mode=block");

  next();
});

// Step 4: Define a simple route
app.get('/', (req, res) => {
  res.send("Secure API is running with advanced security headers!");
});

// Step 5: Start the Server
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

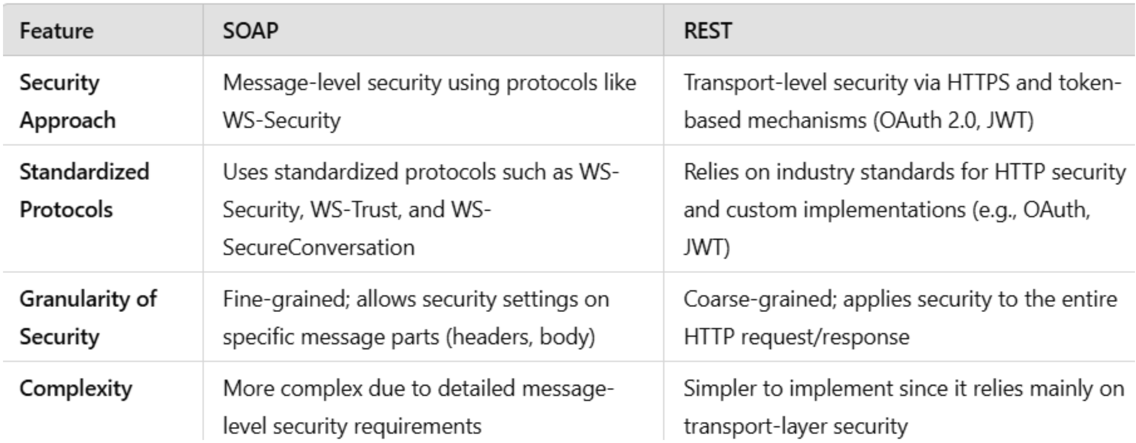
4. **Discuss how SOAP services handle security differently from RESTful services.**

SOAP Services :

- Use WS-Security, which provides message-level encryption and integrity.
- Support authentication using SAML (Security Assertion Markup Language).
- Use XML-based encryption and signatures.

RESTful Services :

- Rely on HTTPS for transport-level security.
- Use OAuth 2.0 and JWT for authentication.
- Implement security headers and CORS for API protection.



Feature	SOAP	REST
Security Approach	Message-level security using protocols like WS-Security	Transport-level security via HTTPS and token-based mechanisms (OAuth 2.0, JWT)
Standardized Protocols	Uses standardized protocols such as WS-Security, WS-Trust, and WS-SecureConversation	Relies on industry standards for HTTP security and custom implementations (e.g., OAuth, JWT)
Granularity of Security	Fine-grained; allows security settings on specific message parts (headers, body)	Coarse-grained; applies security to the entire HTTP request/response
Complexity	More complex due to detailed message-level security requirements	Simpler to implement since it relies mainly on transport-layer security

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.

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

<u>Authorization Patterns in Web Services</u>
API Keys:
A simple token provided with each request to identify the calling application. It's easy to implement but generally less secure.

<u>Basic Authentication:</u>
Uses a username and password encoded in the HTTP headers. It's straightforward but transmits credentials with each request (usually secured by HTTPS).

<u>OAuth 2.0:</u>
A robust framework that allows third-party applications to obtain limited access to an HTTP service. It involves authorization flows, scopes, and tokens, making it ideal for delegating authorization.

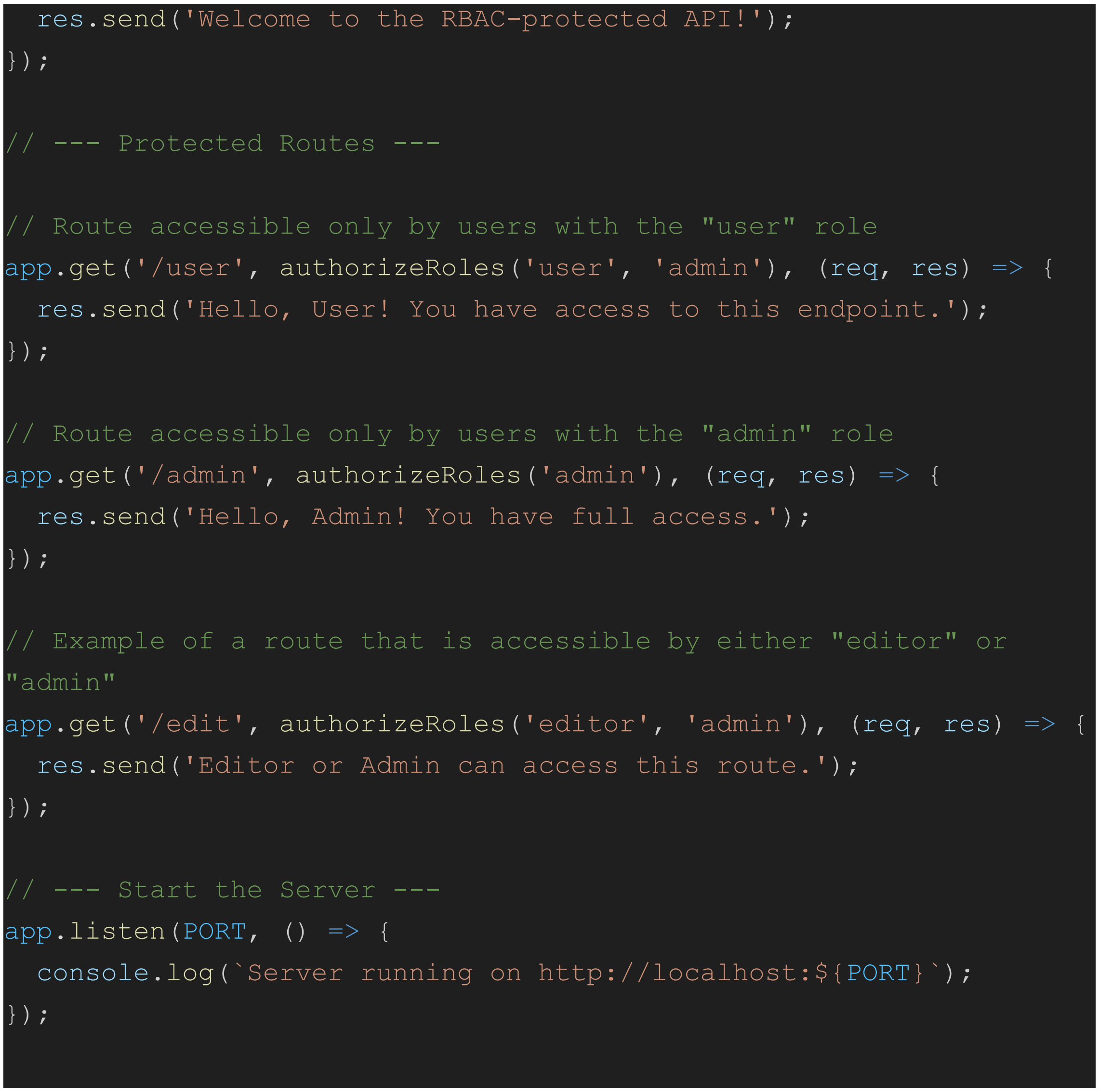
<u>JWT (JSON Web Tokens):</u>
A compact, self-contained token that includes claims about the user (such as roles) and is signed to prevent tampering. JWTs are commonly used in RESTful services.

<u>Role-Based Access Control (RBAC):</u>
Permissions are tied to roles rather than individual users. Users are assigned roles (like admin, user, editor), and roles define what actions are allowed. RBAC is often implemented using middleware that checks the user's role before allowing

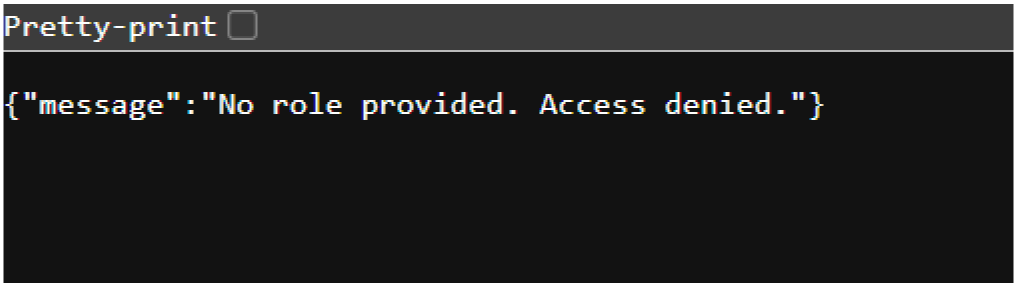
access to certain endpoints.

Attribute-Based Access Control (ABAC):
Uses policies that combine attributes of the user, resource, and environment. It offers granular control but is more complex to manage than RBAC.

```
const express = require('express');
const app = express();
const PORT = process.env.PORT || 4000;

// Middleware to parse JSON requests
app.use(express.json());

// --- RBAC Middleware ---
// This function checks if the user's role (passed in header "x-role")
// is allowed to access the endpoint.
function authorizeRoles(...allowedRoles) {
  return (req, res, next) => {
    // Simulated role extraction; in real cases, this might come from a decoded JWT
    const userRole = req.headers['x-role'];

    if (!userRole) {
      return res.status(401).json({ message: "No role provided. Access denied." });
    }

    if (!allowedRoles.includes(userRole)) {
      return res.status(403).json({ message: "You do not have permission to access this resource." });
    }
    next();
  };
}

// --- Public Route ---
app.get('/', (req, res) => {
```

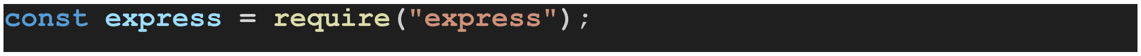


```
  res.send('Welcome to the RBAC-protected API!');
});

// --- Protected Routes ---

// Route accessible only by users with the "user" role
app.get('/user', authorizeRoles('user', 'admin'), (req, res) => {
  res.send('Hello, User! You have access to this endpoint.');
});

// Route accessible only by users with the "admin" role
app.get('/admin', authorizeRoles('admin'), (req, res) => {
  res.send('Hello, Admin! You have full access.');
});

// Example of a route that is accessible by either "editor" or
"admin"
app.get('/edit', authorizeRoles('editor', 'admin'), (req, res) => {
  res.send('Editor or Admin can access this route.');
});

// --- Start the Server ---
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```



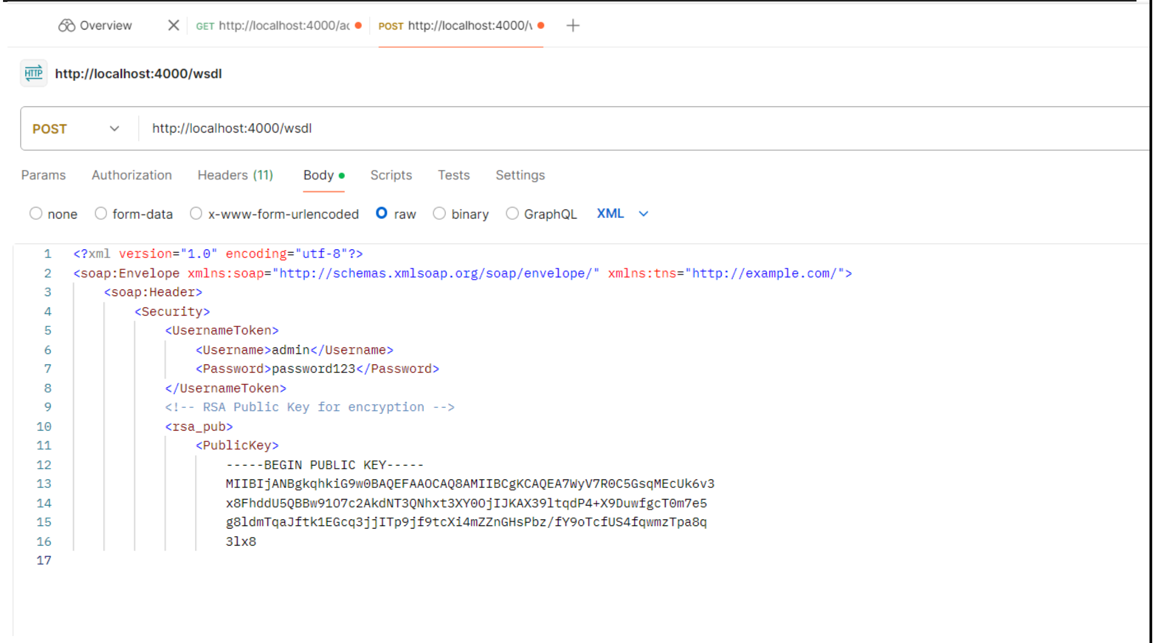
```
{"message":"No role provided. Access denied."}
```

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

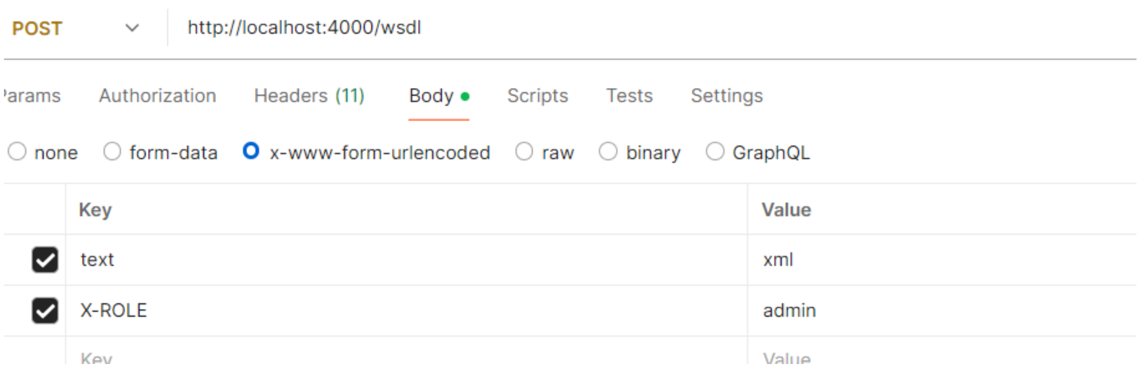


```
const express = require("express");
const soap = require("soap");
const { SignedXml } = require("xml-crypto");
const xmlenc = require("xml-encryption");

const app = express();
const PORT = 4000;

// Sample WSDL (Web Service Definition Language)
const wsdl = `
<definitions name="MyService" targetNamespace="http://example.com/"
    xmlns:tns="http://example.com/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

    <message name="GetMessageRequest">
        <part name="name" type="xsd:string"/>
    </message>

    <message name="GetMessageResponse">
        <part name="message" type="xsd:string"/>
    </message>

    <portType name="MyServicePort">
        <operation name="getMessage">
            <input message="tns:GetMessageRequest"/>
            <output message="tns:GetMessageResponse"/>
        </operation>
    </portType>

    <binding name="MyServiceBinding" type="tns:MyServicePort">
        <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="getMessage">
            <soap:operation
soapAction="http://example.com/getMessage"/>
            <input>
                <soap:body use="encoded"
namespace="http://example.com/"
```

```
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
                <soap:body use="encoded"
namespace="http://example.com/"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
    </binding>

    <service name="MyService">
        <port name="MyServicePort" binding="tns:MyServiceBinding">
            <soap:address location="http://localhost:4000/wsdl"/>
        </port>
    </service>
</definitions>`;

// User Authentication Middleware for WS-Security
function authenticateWSSE(headers) {
    if (!headers || !headers.Security ||
!headers.Security.UsernameToken) {
        throw new Error("Missing WS-Security headers");
    }

    const { Username, Password } = headers.Security.UsernameToken;
    if (Username !== "admin" || Password !== "password123") {
        throw new Error("Invalid WS-Security Credentials");
    }
}

// Sample Encryption Key (Use proper RSA keys in production)
const encryptionKey = "mySecretEncryptionKey";

// Encrypt and Sign Response
function secureResponse(response) {
    return new Promise((resolve, reject) => {
        xmlenc.encrypt(response, { key: encryptionKey }, (err,
encryptedXml) => {
            if (err) return reject(err);
            // Sign XML
            const sig = new SignedXml();
            sig.addReference("//*[local-name(.)='EncryptedData']");
            sig.signingKey = encryptionKey;
            sig.computeSignature(encryptedXml);

            resolve(sig.getSignedXml());
        });
    });
}

// SOAP Service Implementation
const service = {
    MyService: {
        MyServicePort: {
            getMessage: async function (args, callback, headers) {
                try {
                    authenticateWSSE(headers);

                    const responseMessage = `Hello, ${args.name}`;
                    const securedResponse = await
secureResponse(responseMessage);

                    return { message: securedResponse };
                } catch (error) {
                    return { message: `Security Error:
${error.message}` };
                }
            },
        },
    },
};

// Create SOAP Server
const server = require("http").createServer(app);
soap.listen(server, "/wsdl", service, wsdl);

server.listen(PORT, () => {
    console.log(`SOAP Web Service running on
http://localhost:${PORT}/wsdl`);
});
```



```xml
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:tns="http://example.com/">
    <soap:Header>
        <Security>
            <UsernameToken>
                <Username>admin</Username>
                <Password>password123</Password>
            </UsernameToken>
            <!-- RSA Public Key for encryption -->
            <rsa_pub>
                <PublicKey>
                    -----BEGIN PUBLIC KEY-----
                    MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA7WyV7R0C5GsqMEcUk6v3
                    x8FhddU5QBBw9107c2AkdNT3QNhxt3XY0OjIJKAX39ltqdP4+X9DuwfgcT0m7e5
                    g8ldmTqaJftk1EGcq3jjITp9jf9tcXi4mZZnGHsPbz/fY9oTcfUS4fqwmzTpa8q
                    3lx8

```



	Key	Value
☑	text	xml
☑	X-ROLE	admin
	Key	Value