HTML5 Security Cheat Sheet

Introduction

The following cheat sheet serves as a guide for implementing HTML 5 in a secure fashion.

Communication APIs

Web Messaging

Web Messaging (also known as Cross Domain Messaging) provides a means of messaging between documents from different origins in a way that is generally safer than the multiple hacks used in the past to accomplish this task. However, there are still some recommendations to keep in mind:

- When posting a message, explicitly state the expected origin as the second argument to `postMessage` rather than `*` in order to prevent sending the message to an unknown origin after a redirect or some other means of the target window's origin changing.
- The receiving page should **always**:
  - Check the `origin` attribute of the sender to verify the data is originating from the expected location.
  - Perform input validation on the `data` attribute of the event to ensure that it's in the desired format.
- Don't assume you have control over the `data` attribute. A single Cross Site Scripting flaw in the sending page allows an attacker to send messages of any given format.
- Both pages should only interpret the exchanged messages as **data**. Never evaluate passed messages as code (e.g. via `eval()`) or insert it to a page DOM (e.g. via `innerHTML`), as that would create a DOM-based XSS vulnerability. For more information see DOM based XSS Prevention Cheat Sheet.
- To assign the data value to an element, instead of using a insecure method like `element.innerHTML=data;`, use the safer option: `element.textContent=data;`
- Check the origin properly exactly to match the FQDN(s) you expect. Note that the following code: `if(message.origin.indexOf(".owasp.org")!=-1) { /* ... */ }` is very insecure and will not have the desired behavior as `owasp.org.attacker.com` will match.
- If you need to embed external content/untrusted gadgets and allow user-controlled scripts (which is highly discouraged), consider using a JavaScript rewriting framework such as Google Caja or check the information on sandboxed frames.

Cross Origin Resource Sharing

- Validate URLs passed to `XMLHttpRequest.open`. Current browsers allow these URLs to be cross domain; this behavior can lead to code injection by a remote attacker. Pay extra attention to absolute URLs.
- Ensure that URLs responding with `Access-Control-Allow-Origin: *` do not include any sensitive content or information that might aid attacker in further attacks. Use the `Access-Control-Allow-Origin` header only on chosen URLs that need to be accessed cross-domain. Don't use the header for the whole domain.
- Allow only selected, trusted domains in the `Access-Control-Allow-Origin` header. Prefer allowing specific domains over blocking or allowing any domain (do not use `*` wildcard nor blindly return the `Origin` header content without any checks).
- Keep in mind that CORS does not prevent the requested data from going to an unauthorized location. It's still important for the server to perform usual CSRF prevention.
- While the RFC recommends a pre-flight request with the `OPTIONS` verb, current implementations might not perform this request, so it's important that "ordinary" (`GET` and `POST`) requests perform any access control necessary.
- Discard requests received over plain HTTP with HTTPS origins to prevent mixed content bugs.
- Don't rely only on the Origin header for Access Control checks. Browser always sends this header in CORS requests, but may be spoofed outside the browser. Application-level protocols should be used to protect sensitive data.

WebSockets

- Drop backward compatibility in implemented client/servers and use only protocol versions above hybi-00. Popular Hixie-76 version (hiby-00) and older are outdated and insecure.
- The recommended version supported in latest versions of all current browsers is RFC 6455 (supported by Firefox 11+, Chrome 16+, Safari 6, Opera 12.50, and IE10).
- While it's relatively easy to tunnel TCP services through WebSockets (e.g. VNC, FTP), doing so enables access to these tunneled services for the in-browser attacker in case of a Cross Site Scripting attack. These services might also be called directly from a malicious page or program.
- The protocol doesn't handle authorization and/or authentication. Application-level protocols should handle that separately in case sensitive data is being transferred.

- Process the messages received by the websocket as data. Don't try to assign it directly to the DOM nor evaluate as code. If the response is JSON, never use the insecure `eval()` function; use the safe option JSON.parse() instead.
- Endpoints exposed through the `ws://` protocol are easily reversible to plain text. Only `wss://` (WebSockets over SSL/TLS) should be used for protection against Man-In-The-Middle attacks.
- Spoofing the client is possible outside a browser, so the WebSockets server should be able to handle incorrect/malicious input. Always validate input coming from the remote site, as it might have been altered.
- When implementing servers, check the `Origin:` header in the Websockets handshake. Though it might be spoofed outside a browser, browsers always add the Origin of the page that initiated the Websockets connection.
- As a WebSockets client in a browser is accessible through JavaScript calls, all Websockets communication can be spoofed or hijacked through Cross Site Scripting. Always validate data coming through a WebSockets connection.

Server-Sent Events

- Validate URLs passed to the `EventSource` constructor, even though only same-origin URLs are allowed.
- As mentioned before, process the messages (`event.data`) as data and never evaluate the content as HTML or script code.
- Always check the origin attribute of the message (`event.origin`) to ensure the message is coming from a trusted domain. Use an allow-list approach.

Storage APIs

Local Storage

- Also known as Offline Storage, Web Storage. Underlying storage mechanism may vary from one user agent to the next. In other words, any authentication your application requires can be bypassed by a user with local privileges to the machine on which the data is stored. Therefore, it's recommended to avoid storing any sensitive information in local storage where authentication would be assumed.
- Due to the browser's security guarantees it is appropriate to use local storage where access to the data is not assuming authentication or authorization.
- Use the object sessionStorage instead of localStorage if persistent storage is not needed. sessionStorage object is available only to that window/tab until the window is closed.
- A single Cross Site Scripting can be used to steal all the data in these objects, so again it's recommended not to store sensitive information in local storage.
- A single Cross Site Scripting can be used to load malicious data into these objects too, so don't consider objects in these to be trusted.
- Pay extra attention to "localStorage.getItem" and "setItem" calls implemented in HTML5 page. It helps in detecting when developers build solutions that put sensitive information in local storage, which can be a severe risk if authentication or authorization to that data is incorrectly assumed.
- Do not store session identifiers in local storage as the data is always accessible by JavaScript. Cookies can mitigate this risk using the `httpOnly` flag.
- There is no way to restrict the visibility of an object to a specific path like with the attribute path of HTTP Cookies, every object is shared within an origin and protected with the Same Origin Policy. Avoid host multiple applications on the same origin, all of them would share the same localStorage object, use different subdomains instead.

Client-side databases

- On November 2010, the W3C announced Web SQL Database (relational SQL database) as a deprecated specification. A new standard Indexed Database API or IndexedDB (formerly WebSimpleDB) is actively developed, which provides key-value database storage and methods for performing advanced queries.
- Underlying storage mechanisms may vary from one user agent to the next. In other words, any authentication your application requires can be bypassed by a user with local privileges to the machine on which the data is stored. Therefore, it's recommended not to store any sensitive information in local storage.
- If utilized, WebDatabase content on the client side can be vulnerable to SQL injection and needs to have proper validation and parameterization.
- Like Local Storage, a single Cross Site Scripting can be used to load malicious data into a web database as well. Don't consider data in these to be trusted.

Geolocation

- The Geolocation RFC recommends that the user agent ask the user's permission before calculating location. Whether or how this decision is remembered varies from browser to browser. Some user agents require the user to visit the page again in order to turn off the ability to get the user's location without asking, so for privacy reasons, it's recommended to require user input before calling `getCurrentPosition` or `watchPosition`.

Web Workers

- Web Workers are allowed to use `XMLHttpRequest` object to perform in-domain and Cross Origin Resource Sharing requests. See relevant section of this

Cheat Sheet to ensure CORS security.

- While Web Workers don't have access to DOM of the calling page, malicious Web Workers can use excessive CPU for computation, leading to Denial of Service condition or abuse Cross Origin Resource Sharing for further exploitation. Ensure code in all Web Workers scripts is not malevolent. Don't allow creating Web Worker scripts from user supplied input.
- Validate messages exchanged with a Web Worker. Do not try to exchange snippets of JavaScript for evaluation e.g. via `eval()` as that could introduce a DOM Based XSS vulnerability.

Tabnabbing

Attack is described in detail in this article.

To summarize, it's the capacity to act on parent page's content or location from a newly opened page via the back link exposed by the **opener** JavaScript object instance.

It applies to an HTML link or a JavaScript `window.open` function using the attribute/instruction `target` to specify a target loading location that does not replace the current location and then makes the current window/tab available.

To prevent this issue, the following actions are available:

Cut the back link between the parent and the child pages:

- For HTML links:
  - To cut this back link, add the attribute `rel="noopener"` on the tag used to create the link from the parent page to the child page. This attribute value cuts the link, but depending on the browser, lets referrer information be present in the request to the child page.
  - To also remove the referrer information use this attribute value: `rel="noopener noreferrer"`.
- For the JavaScript `window.open` function, add the values `noopener,noreferrer` in the windowFeatures parameter of the `window.open` function.

As the behavior using the elements above is different between the browsers, either use an HTML link or JavaScript to open a window (or tab), then use this configuration to maximize the cross supports:

- For HTML links, add the attribute `rel="noopener noreferrer"` to every link.
- For JavaScript, use this function to open a window (or tab):

```
function openPopup(url, name, windowFeatures){
  //Open the popup and set the opener and referrer policy instruction
  var newWindow = window.open(url, name, 'noopener,noreferrer,' + windowFeatures);
  //Reset the opener link
  newWindow.opener = null;
}
```

- Add the HTTP response header `Referrer-Policy: no-referrer` to every HTTP response sent by the application (Header Referrer-Policy information. This configuration will ensure that no referrer information is sent along with requests from the page.

Compatibility matrix:

- noopener
- noreferrer
- referrer-policy

Sandboxed frames

- Use the `sandbox` attribute of an `iframe` for untrusted content.
- The `sandbox` attribute of an `iframe` enables restrictions on content within an `iframe`. The following restrictions are active when the `sandbox` attribute is set:
  a. All markup is treated as being from a unique origin.
  b. All forms and scripts are disabled.
  c. All links are prevented from targeting other browsing contexts.
  d. All features that trigger automatically are blocked.
  e. All plugins are disabled.

It is possible to have a fine-grained control over `iframe` capabilities using the value of the `sandbox` attribute.

- In old versions of user agents where this feature is not supported, this attribute will be ignored. Use this feature as an additional layer of protection or check if the browser supports sandboxed frames and only show the untrusted content if supported.
- Apart from this attribute, to prevent Clickjacking attacks and unsolicited framing it is encouraged to use the header `X-Frame-Options` which supports

the `deny` and `same-origin` values. Other solutions like framebusting `if(window!==window.top) { window.top.location=location;}` are not recommended.

Credential and Personally Identifiable Information (PII) Input hints

- Protect the input values from being cached by the browser.

> Access a financial account from a public computer. Even though one is logged-off, the next person who uses the machine can log-in because the browser autocomplete functionality. To mitigate this, we tell the input fields not to assist in any way.

```
<input type="text" spellcheck="false" autocomplete="off" autocorrect="off" autocapitalize="off"></input>
```

Text areas and input fields for PII (name, email, address, phone number) and login credentials (username, password) should be prevented from being stored in the browser. Use these HTML5 attributes to prevent the browser from storing PII from your form:

- `spellcheck="false"`
- `autocomplete="off"`
- `autocorrect="off"`
- `autocapitalize="off"`

Offline Applications

- Whether the user agent requests permission from the user to store data for offline browsing and when this cache is deleted, varies from one browser to the next. Cache poisoning is an issue if a user connects through insecure networks, so for privacy reasons it is encouraged to require user input before sending any `manifest` file.
- Users should only cache trusted websites and clean the cache after browsing through open or insecure networks.

Progressive Enhancements and Graceful Degradation Risks

- The best practice now is to determine the capabilities that a browser supports and augment with some type of substitute for capabilities that are not directly supported. This may mean an onion-like element, e.g. falling through to a Flash Player if the `<video>` tag is unsupported, or it may mean additional scripting code from various sources that should be code reviewed.

HTTP Headers to enhance security

Consult the project OWASP Secure Headers in order to obtains the list of HTTP security headers that an application should use to enable defenses at browser level.

WebSocket implementation hints

In addition to the elements mentioned above, this is the list of areas for which caution must be taken during the implementation.

- Access filtering through the "Origin" HTTP request header
- Input / Output validation
- Authentication
- Authorization
- Access token explicit invalidation
- Confidentiality and Integrity

The section below will propose some implementation hints for every area and will go along with an application example showing all the points described.

The complete source code of the example application is available here.

Access filtering

During a websocket channel initiation, the browser sends the **Origin** HTTP request header that contains the source domain initiation for the request to handshake. Even if this header can be spoofed in a forged HTTP request (not browser based), it cannot be overridden or forced in a browser context. It then represents a good candidate to apply filtering according to an expected value.

An example of an attack using this vector, named *Cross-Site WebSocket Hijacking (CSWSH)*, is described here.

The code below defines a configuration that applies filtering based on an "allow list" of origins. This ensures that only allowed origins can establish a full

handshake:

```
import org.owasp.encoder.Encode;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.websocket.server.ServerEndpointConfig;
import java.util.Arrays;
import java.util.List;

/**
 * Setup handshake rules applied to all WebSocket endpoints of the application.
 * Use to setup the Access Filtering using "Origin" HTTP header as input information.
 *
 * @see "http://docs.oracle.com/javaee/7/api/index.html?javax/websocket/server/
 * ServerEndpointConfig.Configurator.html"
 * @see "https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Origin"
 */
public class EndpointConfigurator extends ServerEndpointConfig.Configurator {

    /**
     * Logger
     */
    private static final Logger LOG = LoggerFactory.getLogger(EndpointConfigurator.class);

    /**
     * Get the expected source origins from a JVM property in order to allow external configuration
     */
    private static final List<String> EXPECTED_ORIGINS =  Arrays.asList(System.getProperty("source.origins")
                                                          .split(";"));

    /**
     * {@inheritDoc}
     */
    @Override
    public boolean checkOrigin(String originHeaderValue) {
        boolean isAllowed = EXPECTED_ORIGINS.contains(originHeaderValue);
        String safeOriginValue = Encode.forHtmlContent(originHeaderValue);
        if (isAllowed) {
            LOG.info("[EndpointConfigurator] New handshake request received from {} and was accepted.",
                     safeOriginValue);
        } else {
            LOG.warn("[EndpointConfigurator] New handshake request received from {} and was rejected !",
                     safeOriginValue);
        }
        return isAllowed;
    }

}
```

Authentication and Input/Output validation

When using websocket as communication channel, it's important to use an authentication method allowing the user to receive an access *Token* that is not automatically sent by the browser and then must be explicitly sent by the client code during each exchange.

HMAC digests are the simplest method, and JSON Web Token is a good feature rich alternative, because it allows the transport of access ticket information in a stateless and not alterable way. Moreover, it defines a validity timeframe. You can find additional information about JWT token hardening on this cheat sheet.

JSON Validation Schema are used to define and validate the expected content in input and output messages.

The code below defines the complete authentication messages flow handling:

Authentication Web Socket endpoint - Provide a WS endpoint that enables authentication exchange

```
import org.owasp.pocwebsocket.configurator.EndpointConfigurator;
import org.owasp.pocwebsocket.decoder.AuthenticationRequestDecoder;
import org.owasp.pocwebsocket.encoder.AuthenticationResponseEncoder;
import org.owasp.pocwebsocket.handler.AuthenticationMessageHandler;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.websocket.CloseReason;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

/**
 * Class in charge of managing the client authentication.
 *
 * @see "http://docs.oracle.com/javaee/7/api/javax/websocket/server/ServerEndpointConfig.Configurator.html"
```

```
 * @see "http://svn.apache.org/viewvc/tomcat/trunk/webapps/examples/WEB-INF/classes/websocket/"
 */
@ServerEndpoint(value = "/auth", configurator = EndpointConfigurator.class,
subprotocols = {"authentication"}, encoders = {AuthenticationResponseEncoder.class},
decoders = {AuthenticationRequestDecoder.class})
public class AuthenticationEndpoint {

    /**
     * Logger
     */
    private static final Logger LOG = LoggerFactory.getLogger(AuthenticationEndpoint.class);

    /**
     * Handle the beginning of an exchange
     *
     * @param session Exchange session information
     */
    @OnOpen
    public void start(Session session) {
        //Define connection idle timeout and message limits in order to mitigate as much as possible
        //DOS attacks using massive connection opening or massive big messages sending
        int msgMaxSize = 1024 * 1024;//1 MB
        session.setMaxIdleTimeout(60000);//1 minute
        session.setMaxTextMessageBufferSize(msgMaxSize);
        session.setMaxBinaryMessageBufferSize(msgMaxSize);
        //Log exchange start
        LOG.info("[AuthenticationEndpoint] Session {} started", session.getId());
        //Affect a new message handler instance in order to process the exchange
        session.addMessageHandler(new AuthenticationMessageHandler(session.getBasicRemote()));
        LOG.info("[AuthenticationEndpoint] Session {} message handler affected for processing",
                session.getId());
    }

    /**
     * Handle error case
     *
     * @param session Exchange session information
     * @param thr     Error details
     */
    @OnError
    public void onError(Session session, Throwable thr) {
        LOG.error("[AuthenticationEndpoint] Error occur in session {}", session.getId(), thr);
    }

    /**
     * Handle close event
     *
     * @param session     Exchange session information
     * @param closeReason Exchange closing reason
     */
    @OnClose
    public void onClose(Session session, CloseReason closeReason) {
        LOG.info("[AuthenticationEndpoint] Session {} closed: {}", session.getId(),
                closeReason.getReasonPhrase());
    }

}
```

Authentication message handler - Handle all authentication requests

```
import org.owasp.pocwebsocket.enumeration.AccessLevel;
import org.owasp.pocwebsocket.util.AuthenticationUtils;
import org.owasp.pocwebsocket.vo.AuthenticationRequest;
import org.owasp.pocwebsocket.vo.AuthenticationResponse;
import org.owasp.encoder.Encode;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.websocket.EncodeException;
import javax.websocket.MessageHandler;
import javax.websocket.RemoteEndpoint;
import java.io.IOException;

/**
 * Handle authentication message flow
 */
public class AuthenticationMessageHandler implements MessageHandler.Whole<AuthenticationRequest> {

    private static final Logger LOG = LoggerFactory.getLogger(AuthenticationMessageHandler.class);

    /**
     * Reference to the communication channel with the client
     */
    private RemoteEndpoint.Basic clientConnection;

    /**
     * Constructor
```

```
     *
     * @param clientConnection Reference to the communication channel with the client
     */
    public AuthenticationMessageHandler(RemoteEndpoint.Basic clientConnection) {
        this.clientConnection = clientConnection;
    }


    /**
     * {@inheritDoc}
     */
    @Override
    public void onMessage(AuthenticationRequest message) {
        AuthenticationResponse response = null;
        try {
            //Authenticate
            String authenticationToken = "";
            String accessLevel = this.authenticate(message.getLogin(), message.getPassword());
            if (accessLevel != null) {
                //Create a simple JSON token representing the authentication profile
                authenticationToken = AuthenticationUtils.issueToken(message.getLogin(), accessLevel);
            }
            //Build the response object
            String safeLoginValue = Encode.forHtmlContent(message.getLogin());
            if (!authenticationToken.isEmpty()) {
                response = new AuthenticationResponse(true, authenticationToken, "Authentication succeed !");
                LOG.info("[AuthenticationMessageHandler] User {} authentication succeed.", safeLoginValue);
            } else {
                response = new AuthenticationResponse(false, authenticationToken, "Authentication failed !");
                LOG.warn("[AuthenticationMessageHandler] User {} authentication failed.", safeLoginValue);
            }
        } catch (Exception e) {
            LOG.error("[AuthenticationMessageHandler] Error occur in authentication process.", e);
            //Build the response object indicating that authentication fail
            response = new AuthenticationResponse(false, "", "Authentication failed !");
        } finally {
            //Send response
            try {
                this.clientConnection.sendObject(response);
            } catch (IOException | EncodeException e) {
                LOG.error("[AuthenticationMessageHandler] Error occur in response object sending.", e);
            }
        }
    }

    /**
     * Authenticate the user
     *
     * @param login    User login
     * @param password User password
     * @return The access level if the authentication succeed or NULL if the authentication failed
     */
    private String authenticate(String login, String password) {
      ....
    }
}
```

Utility class to manage JWT token - Handle the issuing and the validation of the access token. Simple JWT token has been used for the example (focus was made here on the global WS endpoint implementation) here without extra hardening (see this cheat sheet to apply extra hardening on the JWT token)

```
import com.auth0.jwt.JWT;
import com.auth0.jwt.JWTVerifier;
import com.auth0.jwt.algorithms.Algorithm;
import com.auth0.jwt.interfaces.DecodedJWT;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Calendar;
import java.util.Locale;

/**
 * Utility class to manage the authentication JWT token
 */
public class AuthenticationUtils {

    /**
     * Build a JWT token for a user
     *
     * @param login       User login
     * @param accessLevel Access level of the user
     * @return The Base64 encoded JWT token
     * @throws Exception If any error occur during the issuing
     */
    public static String issueToken(String login, String accessLevel) throws Exception {
        //Issue a JWT token with validity of 30 minutes
        Algorithm algorithm = Algorithm.HMAC256(loadSecret());
```

```
        Calendar c = Calendar.getInstance();
        c.add(Calendar.MINUTE, 30);
        return JWT.create().withIssuer("WEBSOCKET-SERVER").withSubject(login).withExpiresAt(c.getTime())
                  .withClaim("access_level", accessLevel.trim().toUpperCase(Locale.US)).sign(algorithm);
    }

    /**
     * Verify the validity of the provided JWT token
     *
     * @param token JWT token encoded to verify
     * @return The verified and decoded token with user authentication and
     * authorization (access level) information
     * @throws Exception If any error occur during the token validation
     */
    public static DecodedJWT validateToken(String token) throws Exception {
        Algorithm algorithm = Algorithm.HMAC256(loadSecret());
        JWTVerifier verifier = JWT.require(algorithm).withIssuer("WEBSOCKET-SERVER").build();
        return verifier.verify(token);
    }

    /**
     * Load the JWT secret used to sign token using a byte array for secret storage in order
     * to avoid persistent string in memory
     *
     * @return The secret as byte array
     * @throws IOException If any error occur during the secret loading
     */
    private static byte[] loadSecret() throws IOException {
        return Files.readAllBytes(Paths.get("src", "main", "resources", "jwt-secret.txt"));
    }
}
```

JSON schema of the input and output authentication message - Define the expected structure of the input and output messages from the authentication endpoint point of view

```
{
    "$schema": "http://json-schema.org/schema#",
    "title": "AuthenticationRequest",
    "type": "object",
    "properties": {
    "login": {
        "type": "string",
        "pattern": "^[a-zA-Z]{1,10}$"
    },
    "password": {
        "type": "string"
    }
    },
    "required": [
    "login",
    "password"
    ]
}

{
"$schema": "http://json-schema.org/schema#",
"title": "AuthenticationResponse",
"type": "object",
"properties": {
    "isSuccess;": {
    "type": "boolean"
    },
    "token": {
    "type": "string",
    "pattern": "^[a-zA-Z0-9+/=\\._-]{0,500}$"
    },
    "message": {
    "type": "string",
    "pattern": "^[a-zA-Z0-9!\\s]{0,100}$"
    }
},
"required": [
    "isSuccess",
    "token",
    "message"
]
}
```

Authentication message decoder and encoder - Perform the JSON serialization/deserialization and the input/output validation using dedicated JSON Schema. It makes it possible to systematically ensure that all messages received and sent by the endpoint strictly respect the expected structure and content.

```
import com.fasterxml.jackson.databind.JsonNode;
import com.github.fge.jackson.JsonLoader;
import com.github.fge.jsonschema.core.exceptions.ProcessingException;
```

```java
import com.github.fge.jsonschema.core.report.ProcessingReport;
import com.github.fge.jsonschema.main.JsonSchema;
import com.github.fge.jsonschema.main.JsonSchemaFactory;
import com.google.gson.Gson;
import org.owasp.pocwebsocket.vo.AuthenticationRequest;

import javax.websocket.DecodeException;
import javax.websocket.Decoder;
import javax.websocket.EndpointConfig;
import java.io.File;
import java.io.IOException;

/**
 * Decode JSON text representation to an AuthenticationRequest object
 * <p>
 * As there's one instance of the decoder class by endpoint session so we can use the
 * JsonSchema as decoder instance variable.
 */
public class AuthenticationRequestDecoder implements Decoder.Text<AuthenticationRequest> {

    /**
     * JSON validation schema associated to this type of message
     */
    private JsonSchema validationSchema = null;

    /**
     * Initialize decoder and associated JSON validation schema
     *
     * @throws IOException If any error occur during the object creation
     * @throws ProcessingException If any error occur during the schema loading
     */
    public AuthenticationRequestDecoder() throws IOException, ProcessingException {
        JsonNode node = JsonLoader.fromFile(
                        new File("src/main/resources/authentication-request-schema.json"));
        this.validationSchema = JsonSchemaFactory.byDefault().getJsonSchema(node);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public AuthenticationRequest decode(String s) throws DecodeException {
        try {
            //Validate the provided representation against the dedicated schema
            //Use validation mode with report in order to enable further inspection/tracing
            //of the error details
            //Moreover the validation method "validInstance()" generate a NullPointerException
            //if the representation do not respect the expected schema
            //so it's more proper to use the validation method with report
            ProcessingReport validationReport = this.validationSchema.validate(JsonLoader.fromString(s),
                                                                               true);
            //Ensure there no error
            if (!validationReport.isSuccess()) {
                //Simply reject the message here: Don't care about error details...
                throw new DecodeException(s, "Validation of the provided representation failed !");
            }
        } catch (IOException | ProcessingException e) {
            throw new DecodeException(s, "Cannot validate the provided representation to a"
                                      + " JSON valid representation !", e);
        }

        return new Gson().fromJson(s, AuthenticationRequest.class);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public boolean willDecode(String s) {
        boolean canDecode = false;

        //If the provided JSON representation is empty/null then we indicate that
        //representation cannot be decoded to our expected object
        if (s == null || s.trim().isEmpty()) {
            return canDecode;
        }

        //Try to cast the provided JSON representation to our object to validate at least
        //the structure (content validation is done during decoding)
        try {
            AuthenticationRequest test = new Gson().fromJson(s, AuthenticationRequest.class);
            canDecode = (test != null);
        } catch (Exception e) {
            //Ignore explicitly any casting error...
        }

        return canDecode;
    }
```

```
    /**
     * {@inheritDoc}
     */
    @Override
    public void init(EndpointConfig config) {
        //Not used
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void destroy() {
        //Not used
    }
}
```

```
import com.fasterxml.jackson.databind.JsonNode;
import com.github.fge.jackson.JsonLoader;
import com.github.fge.jsonschema.core.exceptions.ProcessingException;
import com.github.fge.jsonschema.core.report.ProcessingReport;
import com.github.fge.jsonschema.main.JsonSchema;
import com.github.fge.jsonschema.main.JsonSchemaFactory;
import com.google.gson.Gson;
import org.owasp.pocwebsocket.vo.AuthenticationResponse;

import javax.websocket.EncodeException;
import javax.websocket.Encoder;
import javax.websocket.EndpointConfig;
import java.io.File;
import java.io.IOException;

/**
 * Encode AuthenticationResponse object to JSON text representation.
 * <p>
 * As there one instance of the encoder class by endpoint session so we can use
 * the JsonSchema as encoder instance variable.
 */
public class AuthenticationResponseEncoder implements Encoder.Text<AuthenticationResponse> {

    /**
     * JSON validation schema associated to this type of message
     */
    private JsonSchema validationSchema = null;

    /**
     * Initialize encoder and associated JSON validation schema
     *
     * @throws IOException If any error occur during the object creation
     * @throws ProcessingException If any error occur during the schema loading
     */
    public AuthenticationResponseEncoder() throws IOException, ProcessingException {
        JsonNode node = JsonLoader.fromFile(
                        new File("src/main/resources/authentication-response-schema.json"));
        this.validationSchema = JsonSchemaFactory.byDefault().getJsonSchema(node);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public String encode(AuthenticationResponse object) throws EncodeException {
        //Generate the JSON representation
        String json = new Gson().toJson(object);
        try {
            //Validate the generated representation against the dedicated schema
            //Use validation mode with report in order to enable further inspection/tracing
            //of the error details
            //Moreover the validation method "validInstance()" generate a NullPointerException
            //if the representation do not respect the expected schema
            //so it's more proper to use the validation method with report
            ProcessingReport validationReport = this.validationSchema.validate(JsonLoader.fromString(json),
                                                                               true);
            //Ensure there no error
            if (!validationReport.isSuccess()) {
                //Simply reject the message here: Don't care about error details...
                throw new EncodeException(object, "Validation of the generated representation failed !");
            }
        } catch (IOException | ProcessingException e) {
            throw new EncodeException(object, "Cannot validate the generated representation to a"+
                                              " JSON valid representation !", e);
        }

        return json;
    }

    /**
     * {@inheritDoc}
```

```
     */
    @Override
    public void init(EndpointConfig config) {
        //Not used
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void destroy() {
        //Not used
    }

}
```

Note that the same approach is used in the messages handling part of the POC. All messages exchanged between the client and the server are systematically validated using the same way, using dedicated JSON schemas linked to messages dedicated Encoder/Decoder (serialization/deserialization).

Authorization and access token explicit invalidation

Authorization information is stored in the access token using the JWT *Claim* feature (in the POC the name of the claim is *access_level*). Authorization is validated when a request is received and before any other action using the user input information.

The access token is passed with every message sent to the message endpoint and a block list is used in order to allow the user to request an explicit token invalidation.

Explicit token invalidation is interesting from a user's point of view because, often when tokens are used, the validity timeframe of the token is relatively long (it's common to see a valid timeframe superior to 1 hour) so it's important to allow a user to have a way to indicate to the system "OK, I have finished my exchange with you, so you can close our exchange session and cleanup associated links".

It also helps the user to revoke itself of current access if a malicious concurrent access is detected using the same token (case of token stealing).

Token block list - Maintain a temporary list using memory and time limited Caching of hashes of token that are not allowed to be used anymore

```
import org.apache.commons.jcs.JCS;
import org.apache.commons.jcs.access.CacheAccess;
import org.apache.commons.jcs.access.exception.CacheException;

import javax.xml.bind.DatatypeConverter;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

/**
 * Utility class to manage the access token that have been declared as no
 * more usable (explicit user logout)
 */
public class AccessTokenBlocklistUtils {
    /**
     * Message content send by user that indicate that the access token that
     * come along the message must be block-listed for further usage
     */
    public static final String MESSAGE_ACCESS_TOKEN_INVALIDATION_FLAG = "INVALIDATE_TOKEN";

    /**
     * Use cache to store block-listed token hash in order to avoid memory exhaustion and be consistent
     * because token are valid 30 minutes so the item live in cache 60 minutes
     */
    private static final CacheAccess<String, String> TOKEN_CACHE;

    static {
        try {
            TOKEN_CACHE = JCS.getInstance("default");
        } catch (CacheException e) {
            throw new RuntimeException("Cannot init token cache !", e);
        }
    }

    /**
     * Add token into the block list
     *
     * @param token Token for which the hash must be added
     * @throws NoSuchAlgorithmException If SHA256 is not available
     */
    public static void addToken(String token) throws NoSuchAlgorithmException {
        if (token != null && !token.trim().isEmpty()) {
            String hashHex = computeHash(token);
            if (TOKEN_CACHE.get(hashHex) == null) {
                TOKEN_CACHE.putSafe(hashHex, hashHex);
            }
        }
    }
```

```
    /**
     * Check if a token is present in the block list
     *
     * @param token Token for which the presence of the hash must be verified
     * @return TRUE if token is block-listed
     * @throws NoSuchAlgorithmException If SHA256 is not available
     */
    public static boolean isBlocklisted(String token) throws NoSuchAlgorithmException {
        boolean exists = false;
        if (token != null && !token.trim().isEmpty()) {
            String hashHex = computeHash(token);
            exists = (TOKEN_CACHE.get(hashHex) != null);
        }
        return exists;
    }

    /**
     * Compute the SHA256 hash of a token
     *
     * @param token Token for which the hash must be computed
     * @return The hash encoded in HEX
     * @throws NoSuchAlgorithmException If SHA256 is not available
     */
    private static String computeHash(String token) throws NoSuchAlgorithmException {
        String hashHex = null;
        if (token != null && !token.trim().isEmpty()) {
            MessageDigest md = MessageDigest.getInstance("SHA-256");
            byte[] hash = md.digest(token.getBytes());
            hashHex = DatatypeConverter.printHexBinary(hash);
        }
        return hashHex;
    }

}
```

Message handling - Process a request from a user to add a message in the list. Show a authorization validation approach example

```
import com.auth0.jwt.interfaces.Claim;
import com.auth0.jwt.interfaces.DecodedJWT;
import org.owasp.pocwebsocket.enumeration.AccessLevel;
import org.owasp.pocwebsocket.util.AccessTokenBlocklistUtils;
import org.owasp.pocwebsocket.util.AuthenticationUtils;
import org.owasp.pocwebsocket.util.MessageUtils;
import org.owasp.pocwebsocket.vo.MessageRequest;
import org.owasp.pocwebsocket.vo.MessageResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.websocket.EncodeException;
import javax.websocket.RemoteEndpoint;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

/**
 * Handle message flow
 */
public class MessageHandler implements javax.websocket.MessageHandler.Whole<MessageRequest> {

    private static final Logger LOG = LoggerFactory.getLogger(MessageHandler.class);

    /**
     * Reference to the communication channel with the client
     */
    private RemoteEndpoint.Basic clientConnection;

    /**
     * Constructor
     *
     * @param clientConnection Reference to the communication channel with the client
     */
    public MessageHandler(RemoteEndpoint.Basic clientConnection) {
        this.clientConnection = clientConnection;
    }


    /**
     * {@inheritDoc}
     */
    @Override
    public void onMessage(MessageRequest message) {
        MessageResponse response = null;
        try {
            /*Step 1: Verify the token*/
            String token = message.getToken();
            //Verify if is it in the block list
```

```
            if (AccessTokenBlocklistUtils.isBlocklisted(token)) {
                throw new IllegalAccessException("Token is in the block list !");
            }

            //Verify the signature of the token
            DecodedJWT decodedToken = AuthenticationUtils.validateToken(token);

            /*Step 2: Verify the authorization (access level)*/
            Claim accessLevel = decodedToken.getClaim("access_level");
            if (accessLevel == null || AccessLevel.valueOf(accessLevel.asString()) == null) {
                throw new IllegalAccessException("Token have an invalid access level claim !");
            }

            /*Step 3: Do the expected processing*/
            //Init the list of the messages for the current user
            if (!MessageUtils.MESSAGES_DB.containsKey(decodedToken.getSubject())) {
                MessageUtils.MESSAGES_DB.put(decodedToken.getSubject(), new ArrayList<>());
            }

            //Add message to the list of message of the user if the message is a not a token invalidation
            //order otherwise add the token to the block list
            if (AccessTokenBlocklistUtils.MESSAGE_ACCESS_TOKEN_INVALIDATION_FLAG
                .equalsIgnoreCase(message.getContent().trim())) {
                AccessTokenBlocklistUtils.addToken(message.getToken());
            } else {
                MessageUtils.MESSAGES_DB.get(decodedToken.getSubject()).add(message.getContent());
            }

            //According to the access level of user either return only is message or return all message
            List<String> messages = new ArrayList<>();
            if (accessLevel.asString().equals(AccessLevel.USER.name())) {
                MessageUtils.MESSAGES_DB.get(decodedToken.getSubject())
                .forEach(s -> messages.add(String.format("(%s): %s", decodedToken.getSubject(), s)));
            } else if (accessLevel.asString().equals(AccessLevel.ADMIN.name())) {
                MessageUtils.MESSAGES_DB.forEach((k, v) ->
                v.forEach(s -> messages.add(String.format("(%s): %s", k, s))));
            }

            //Build the response object indicating that exchange succeed
            if (AccessTokenBlocklistUtils.MESSAGE_ACCESS_TOKEN_INVALIDATION_FLAG
                .equalsIgnoreCase(message.getContent().trim())) {
                response = new MessageResponse(true, messages, "Token added to the block list");
            }else{
                response = new MessageResponse(true, messages, "");
            }

        } catch (Exception e) {
            LOG.error("[MessageHandler] Error occur in exchange process.", e);
            //Build the response object indicating that exchange fail
            //We send the error detail on client because ware are in POC (it will not the case in a real app)
            response = new MessageResponse(false, new ArrayList<>(), "Error occur during exchange: "
                    + e.getMessage());
        } finally {
            //Send response
            try {
                this.clientConnection.sendObject(response);
            } catch (IOException | EncodeException e) {
                LOG.error("[MessageHandler] Error occur in response object sending.", e);
            }
        }
    }
}
```

Confidentiality and Integrity

If the raw version of the protocol is used (protocol `ws://`) then the transferred data is exposed to eavesdropping and potential on-the-fly alteration.

Example of capture using Wireshark and searching for password exchanges in the stored PCAP file, not printable characters has been explicitly removed from the command result:

```
$ grep -aE '(password)' capture.pcap
{"login":"bob","password":"bob123"}
```

There is a way to check, at WebSocket endpoint level, if the channel is secure by calling the method `isSecure()` on the *session* object instance.

Example of implementation in the method of the endpoint in charge of setup of the session and affects the message handler:

```
/**
 * Handle the beginning of an exchange
 *
 * @param session Exchange session information
 */
@OnOpen
public void start(Session session) {
```

```
    ...
    //Affect a new message handler instance in order to process the exchange only if the channel is secured
    if(session.isSecure()) {
        session.addMessageHandler(new AuthenticationMessageHandler(session.getBasicRemote()));
    }else{
        LOG.info("[AuthenticationEndpoint] Session {} do not use a secure channel so no message handler " +
                "was affected for processing and session was explicitly closed !", session.getId());
        try{
            session.close(new CloseReason(CloseReason.CloseCodes.CANNOT_ACCEPT,"Insecure channel used !"));
        }catch(IOException e){
            LOG.error("[AuthenticationEndpoint] Session {} cannot be explicitly closed !", session.getId(),
                    e);
        }

    }
    LOG.info("[AuthenticationEndpoint] Session {} message handler affected for processing", session.getId());
}
```

Expose WebSocket endpoints only on wss:// protocol (WebSockets over SSL/TLS) in order to ensure *Confidentiality* and *Integrity* of the traffic like using HTTP over SSL/TLS to secure HTTP exchanges.