HMAC (Hash-based Message Authentication Code) is a robust method for securing Spring Boot APIs by ensuring both data integrity and authenticity.[1] It's a great alternative or addition to traditional token-based authentication like JWT, especially when you need to verify that a request hasn't been tampered with and comes from a trusted source.

Here's a breakdown of how HMAC authentication works and how to implement it in a Spring Boot API:

How HMAC Authentication Works

1. **Shared Secret:** Both the client and the server possess a pre-shared secret key.[2] This key is never transmitted over the network.
2. **Message to be Signed:** The client constructs a string (the "message to be signed") that includes various parts of the HTTP request, such as:
    - HTTP method (GET, POST, PUT, DELETE)
    - Request URI/path
    - Request body (for POST/PUT requests)
    - Timestamp (to prevent replay attacks)[3]
    - Nonce (a unique, random value used once, also for replay attack prevention)[4]
    - Other relevant headers (e.g., Content-Type)
3. **HMAC Generation (Client-side):** The client uses the shared secret key and a cryptographic hash function (e.g., SHA-256, SHA-512) to compute an HMAC of the "message to be signed."[5]
4. **Header Transmission:** The client sends the API request, including the calculated HMAC in a specific HTTP header (e.g., Authorization, X-HMAC-Signature), along with the timestamp and nonce.
5. **HMAC Verification (Server-side):**
    - The server receives the request.
    - It extracts the HMAC, timestamp, and nonce from the request headers.
    - Using its own copy of the shared secret key and the same cryptographic hash function, the server reconstructs the "message to be signed" based on the incoming request's details (method, URI, body, timestamp, nonce).
    - It then computes its own HMAC of this reconstructed message.
    - Finally, the server compares its calculated HMAC with the HMAC received from the client.
6. **Authentication and Authorization:**
    - If the HMACs match, it confirms that the request originated from a legitimate source (possessing the shared secret) and that the message hasn't been altered during transit.[6]
    - The server also checks the timestamp and nonce to prevent replay attacks (e.g., ensuring the request isn't too old and the nonce hasn't been used before).
    - If all checks pass, the request is authenticated, and the server proceeds with

processing the request. Otherwise, it rejects the request (e.g., with a 401 Unauthorized or 403 Forbidden status).

Key Components for Spring Boot Implementation

To implement HMAC authentication in Spring Boot, you'll typically need:

1. **Custom Filter/Interceptor:** To intercept incoming requests and perform the HMAC verification. Spring Security's Filter or HandlerInterceptor are good candidates.
2. **HMAC Utility Class:** A helper class to generate and verify HMACs using Java's javax.crypto.Mac class.
3. **Secret Key Management:** A secure way to store and retrieve the shared secret keys (e.g., in application.properties, environment variables, or a secure key store).
4. **Client-side Implementation:** You'll also need to implement the HMAC generation logic on your client-side application that consumes the Spring Boot API.

Example Implementation (Conceptual)

Let's outline the core parts.

1. application.properties (or similar for key management)

Properties

```
hmac.auth.secret-key=your-super-secret-key-that-should-be-long-and-random
hmac.auth.signature-header=X-HMAC-Signature
hmac.auth.access-key-header=X-Access-Key
hmac.auth.timestamp-header=X-Timestamp
hmac.auth.nonce-header=X-Nonce
hmac.auth.validity-duration-seconds=300 # 5 minutes for timestamp validity
```

2. HMAC Utility Class (HmacUtil.java)[7]

Java

```
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.StandardCharsets;
import java.security.InvalidKeyException;
```

```
import java.security.NoSuchAlgorithmException;
import java.util.Base64;

public class HmacUtil {

    private static final String HMAC_ALGORITHM = "HmacSHA256";

    public static String calculateHmac(String data, String secretKey)
            throws NoSuchAlgorithmException, InvalidKeyException {
        Mac hmacSha256 = Mac.getInstance(HMAC_ALGORITHM);
        SecretKeySpec secretKeySpec = new
SecretKeySpec(secretKey.getBytes(StandardCharsets.UTF_8), HMAC_ALGORITHM);
        hmacSha256.init(secretKeySpec);
        byte[] hmacBytes = hmacSha256.doFinal(data.getBytes(StandardCharsets.UTF_8));
        return Base64.getEncoder().encodeToString(hmacBytes);
    }

    public static boolean verifyHmac(String data, String hmacToVerify, String secretKey)
            throws NoSuchAlgorithmException, InvalidKeyException {
        String calculatedHmac = calculateHmac(data, secretKey);
        return calculatedHmac.equals(hmacToVerify);
    }

    // Helper to build the string to be signed.
    // This is crucial and must be consistent between client and server.
    public static String buildDataToSign(String httpMethod, String requestUri, String requestBody, long
timestamp, String nonce) {
        StringBuilder data = new StringBuilder();
        data.append(httpMethod.toUpperCase()).append("\n");
        data.append(requestUri).append("\n");
        data.append(requestBody != null ? requestBody : "").append("\n"); // Include body for
POST/PUT
        data.append(timestamp).append("\n");
        data.append(nonce);
        return data.toString();
    }
}
```

3. Custom Spring Security Filter (HmacAuthFilter.java)

This filter will intercept requests before they hit your controllers.

Java

```java
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.util.Collections;
import java.util.concurrent.TimeUnit;

public class HmacAuthFilter extends OncePerRequestFilter {

    private final String secretKey;
    private final String signatureHeader;
    private final String accessKeyHeader;
    private final String timestampHeader;
    private final String nonceHeader;
    private final long validityDurationSeconds; // e.g., 300 seconds (5 minutes)

    // In a real application, you'd fetch user details based on the accessKey
    // and store associated secret keys. For simplicity, we use a single global secret.
    // Consider a Map<String, String> for accessKey -> secretKey.
    public HmacAuthFilter(String secretKey, String signatureHeader, String accessKeyHeader,
                          String timestampHeader, String nonceHeader, long validityDurationSeconds) {
        this.secretKey = secretKey;
        this.signatureHeader = signatureHeader;
        this.accessKeyHeader = accessKeyHeader;
        this.timestampHeader = timestampHeader;
        this.nonceHeader = nonceHeader;
        this.validityDurationSeconds = validityDurationSeconds;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain)
            throws ServletException, IOException {
```

```
        String signature = request.getHeader(signatureHeader);
        String accessKey = request.getHeader(accessKeyHeader);
        String timestampStr = request.getHeader(timestampHeader);
        String nonce = request.getHeader(nonceHeader);

        if (signature == null || accessKey == null || timestampStr == null || nonce == null) {
            response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Missing HMAC
authentication headers.");
            return;
        }

        try {
            long timestamp = Long.parseLong(timestampStr);

            // 1. Check timestamp to prevent replay attacks
            long currentTimestamp = System.currentTimeMillis();
            if (TimeUnit.MILLISECONDS.toSeconds(currentTimestamp - timestamp) >
validityDurationSeconds) {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "HMAC signature has
expired.");
                return;
            }

            // 2. Read request body if applicable (for POST/PUT)
            // Note: Reading HttpServletRequest.getInputStream() directly consumes it.
            // You might need a wrapper or a caching mechanism if other filters/handlers
            // also need to read the body.
            String requestBody = "";
            if ("POST".equalsIgnoreCase(request.getMethod()) ||
"PUT".equalsIgnoreCase(request.getMethod())) {
                requestBody = new String(request.getInputStream().readAllBytes(),
StandardCharsets.UTF_8);
            }

            // 3. Reconstruct the data string for HMAC calculation
            String dataToSign = HmacUtil.buildDataToSign(
                    request.getMethod(),
                    request.getRequestURI(),
                    requestBody,
                    timestamp,
                    nonce
            );
```

```
            // 4. Verify HMAC
            if (HmacUtil.verifyHmac(dataToSign, signature, secretKey)) {
                // HMAC is valid, set authentication in Spring Security context
                // In a real app, you'd fetch UserDetails for the 'accessKey'
                // and assign roles/authorities.
                UsernamePasswordAuthenticationToken authentication = new
UsernamePasswordAuthenticationToken(
                        accessKey, null, Collections.emptyList()); // No credentials, no authorities for now
                SecurityContextHolder.getContext().setAuthentication(authentication);
            } else {
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Invalid HMAC signature.");
                return;
            }

        } catch (NumberFormatException e) {
            response.sendError(HttpServletResponse.SC_BAD_REQUEST, "Invalid timestamp
format.");
            return;
        } catch (NoSuchAlgorithmException | InvalidKeyException e) {
            response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR, "HMAC
algorithm error: " + e.getMessage());
            return;
        }

        filterChain.doFilter(request, response);
    }
}
```

4. Spring Security Configuration (SecurityConfig.java)

Java

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;
```

```
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Value("${hmac.auth.secret-key}")
    private String secretKey;

    @Value("${hmac.auth.signature-header}")
    private String signatureHeader;

    @Value("${hmac.auth.access-key-header}")
    private String accessKeyHeader;

    @Value("${hmac.auth.timestamp-header}")
    private String timestampHeader;

    @Value("${hmac.auth.nonce-header}")
    private String nonceHeader;

    @Value("${hmac.auth.validity-duration-seconds}")
    private long validityDurationSeconds;

    @Bean
    public HmacAuthFilter hmacAuthFilter() {
        return new HmacAuthFilter(secretKey, signatureHeader, accessKeyHeader,
                                  timestampHeader, nonceHeader, validityDurationSeconds);
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf().disable() // Disable CSRF for API
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS) // No
sessions
            .and()
            .authorizeRequests()
                .antMatchers("/public/**").permitAll() // Example: allow public endpoints
                .anyRequest().authenticated() // All other requests require authentication
            .and()
            .addFilterBefore(hmacAuthFilter(), UsernamePasswordAuthenticationFilter.class); // Add
```

```
your HMAC filter

        return http.build();
    }
}
```

5. Controller (Example)

Java

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.security.access.prepost.PreAuthorize; // Optional: for method-level security

@RestController
public class MyApiController {

    @GetMapping("/api/data")
    public String getProtectedData() {
        return "This is protected data!";
    }

    @PostMapping("/api/submit")
    public String submitData(@RequestBody String payload) {
        return "Data submitted: " + payload;
    }
}
```

Client-Side Considerations

The client needs to mirror the server's logic for generating the HMAC.

Java

```
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.util.UUID;

public class HmacApiClient {

    private final String secretKey;
    private final String accessKey; // Represents the client's identifier
    private final String signatureHeader;
    private final String accessKeyHeader;
    private final String timestampHeader;
    private final String nonceHeader;

    public HmacApiClient(String accessKey, String secretKey, String signatureHeader, String
accessKeyHeader, String timestampHeader, String nonceHeader) {
        this.accessKey = accessKey;
        this.secretKey = secretKey;
        this.signatureHeader = signatureHeader;
        this.accessKeyHeader = accessKeyHeader;
        this.timestampHeader = timestampHeader;
        this.nonceHeader = nonceHeader;
    }

    public String sendGetRequest(String url) throws IOException, InterruptedException,
NoSuchAlgorithmException, InvalidKeyException {
        long timestamp = System.currentTimeMillis();
        String nonce = UUID.randomUUID().toString();
        String requestUri = URI.create(url).getPath();

        String dataToSign = HmacUtil.buildDataToSign("GET", requestUri, null, timestamp, nonce);
        String signature = HmacUtil.calculateHmac(dataToSign, secretKey);

        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(url))
                .header(signatureHeader, signature)
                .header(accessKeyHeader, accessKey)
                .header(timestampHeader, String.valueOf(timestamp))
```

```
                .header(nonceHeader, nonce)
                .GET()
                .build();

        HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
        return response.body();
    }

    public String sendPostRequest(String url, String requestBody) throws IOException,
InterruptedException, NoSuchAlgorithmException, InvalidKeyException {
        long timestamp = System.currentTimeMillis();
        String nonce = UUID.randomUUID().toString();
        String requestUri = URI.create(url).getPath();

        String dataToSign = HmacUtil.buildDataToSign("POST", requestUri, requestBody,
timestamp, nonce);
        String signature = HmacUtil.calculateHmac(dataToSign, secretKey);

        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(url))
                .header("Content-Type", "application/json") // Don't forget content type for POST
                 .header(signatureHeader, signature)
                 .header(accessKeyHeader, accessKey)
                 .header(timestampHeader, String.valueOf(timestamp))
                 .header(nonceHeader, nonce)
                 .POST(HttpRequest.BodyPublishers.ofString(requestBody))
                .build();

        HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
        return response.body();
    }

    public static void main(String[] args) throws IOException, InterruptedException,
NoSuchAlgorithmException, InvalidKeyException {
        // Use the same keys as configured in your Spring Boot application
        String clientAccessKey = "my-client-id"; // Corresponds to X-Access-Key
        String clientSecretKey = "your-super-secret-key-that-should-be-long-and-random";

        HmacApiClient apiClient = new HmacApiClient(
                clientAccessKey,
                clientSecretKey,
                "X-HMAC-Signature",
                "X-Access-Key",
```

```
            "X-Timestamp",
            "X-Nonce"
        );

        // Example GET request
        String getResponse = apiClient.sendGetRequest("http://localhost:8080/api/data");
        System.out.println("GET Response: " + getResponse);

        // Example POST request
        String postPayload = "{\"message\": \"Hello from client!\"}";
        String postResponse = apiClient.sendPostRequest("http://localhost:8080/api/submit",
postPayload);
        System.out.println("POST Response: " + postResponse);
    }
}
```

Important Considerations and Best Practices

- **Secret Key Management:** This is critical. Never hardcode secret keys in production. Use environment variables, Spring Cloud Config, Vault, or other secure key management solutions.
- **Timestamp and Nonce:**
    - **Timestamp:** Crucial for preventing replay attacks.[8] Set a reasonable validity window (e.g., 5 minutes).
    - **Nonce:** Essential for preventing replay attacks where an attacker re-sends an old, valid request.[9] Implement a server-side cache (e.g., Redis, ConcurrentHashMap for simple cases) to store used nonces and expire them after the timestamp validity period. If a nonce is received that's already in the cache (and not expired), reject the request.
- **Canonicalization of Data to Sign:** The string used to calculate the HMAC *must* be identical on both the client and server. Pay close attention to:
    - **Order of elements:** Define a strict order.
    - **Case sensitivity:** Standardize (e.g., all lowercase or uppercase).
    - **Encoding:** Use UTF-8 consistently.
    - **Request body:** For POST/PUT, the raw request body should be used, not a parsed JSON object. If you're using HttpServletRequest.getInputStream(), be aware it can only be read once. Consider wrapping the request or using a ContentCachingRequestWrapper.
- **Error Handling:** Provide clear and informative error messages to the client without leaking sensitive information.[10]
- **Logging:** Log authentication attempts and failures for monitoring and debugging.
- **Rate Limiting:** Implement rate limiting to prevent brute-force attacks on your API, even with HMAC.[11]

- **User/Client Management:** In a real-world scenario, you'd have a database or service to store and retrieve unique accessKey (client ID) and their corresponding secretKey pairs, potentially mapping them to internal user roles or permissions.
- **Thread Safety:** Ensure your nonce management and secret key retrieval are thread-safe.

By carefully implementing these components, you can build a secure Spring Boot API using HMAC authentication.

Sources

1. https://mobisoftinfotech.com/resources/blog/securing-api-leveraging-hmac-api-security-java
2. https://github.com/athenahealth/aone-fhir-subscriptions
3. https://github.com/Joel-Schaltenbrand/TicketSystem
4. https://github.com/Jcedielb/Caso3-CanalesSeguros
5. https://stackoverflow.com/questions/41007639/cannot-decrypt-blob-send-by-java-server-in-python-client
6. https://github.com/ralscha/wampspring
7. https://github.com/h-AuD/learn-note-projects
8. https://www.jianshu.com/p/f47c7b6b3bf7
9. https://github.com/redouane59/twitter-personal-analyzer-bot
10. https://github.com/nosqlbench/nosqlbench subject to license (Apache - 2.0)
11. https://www.happycoders.eu/java/java-11-features/
12. https://github.com/fasten-project/fasten subject to license (Apache - 2.0)
13. https://github.com/CodeMachine0121/Custom_Blockchain