Java Security Patterns for Amazon Code Review

Overview

Master Java-specific security vulnerability patterns and secure coding practices for Amazon's live code review interviews, with focus on enterprise-scale Spring Boot applications.

Common Java Security Anti-Patterns

1. SQL Injection Vulnerabilities

Vulnerable Pattern - String Concatenation

```
// CRITICAL: SQL Injection vulnerability
@RestController
public class UserController {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @GetMapping("/users/{userId}")
    public ResponseEntity<User> getUser(@PathVariable String userId) {
        // Vulnerable: Direct string concatenation
        String sql = "SELECT * FROM users WHERE id = " + userId;

        List<User> users = jdbcTemplate.query(sql, new BeanPropertyRowMapper<>
(User.class));

        if (!users.isEmpty()) {
            return ResponseEntity.ok(users.get(0));
        }
        return ResponseEntity.notFound().build();
    }
}
```

Business Impact for Amazon Scale:

- 50M+ customer records exposed
- $8.25B potential breach cost ($165 per record)
- GDPR fines up to 4% of global revenue
- Complete authentication system compromise

Secure Pattern - Parameterized Queries

```
// SECURE: Parameterized queries prevent SQL injection
@RestController
public class UserController {
```

```
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @GetMapping("/users/{userId}")
    public ResponseEntity<User> getUser(@PathVariable String userId) {
        // Secure: Parameterized query
        String sql = "SELECT * FROM users WHERE id = ?";

        try {
            User user = jdbcTemplate.queryForObject(sql,
                new BeanPropertyRowMapper<>(User.class), userId);
            return ResponseEntity.ok(user);
        } catch (EmptyResultDataAccessException e) {
            return ResponseEntity.notFound().build();
        }
    }
}
```

Advanced Secure Pattern - Spring Data JPA

```
// BEST PRACTICE: Spring Data JPA with automatic parameterization
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    // Secure: Spring Data automatically parameterizes
    @Query("SELECT u FROM User u WHERE u.email = :email AND u.active = true")
    Optional<User> findActiveUserByEmail(@Param("email") String email);

    // Secure: Method name queries are automatically safe
    Optional<User> findByEmailAndActiveTrue(String email);
}

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User findUserByEmail(String email) {
        return userRepository.findActiveUserByEmail(email)
            .orElseThrow(() -> new UserNotFoundException("User not found"));
    }
}
```

2. Authentication and Authorization Flaws

Vulnerable Pattern - Missing Authorization

```
// CRITICAL: Missing authorization checks
@RestController
@RequestMapping("/admin")
public class AdminController {

    @Autowired
    private UserService userService;

    @GetMapping("/users/{userId}")
    public ResponseEntity<User> getUser(@PathVariable Long userId) {
        // Vulnerable: No authorization check - any authenticated user can access
        User user = userService.findById(userId);
        return ResponseEntity.ok(user);
    }

    @DeleteMapping("/users/{userId}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long userId) {
        // Vulnerable: Critical admin action without permission verification
        userService.deleteUser(userId);
        return ResponseEntity.ok().build();
    }
}
```

Amazon Scale Impact:

- Any user can access admin functions
- Customer data accessible to unauthorized users
- Potential for mass customer data deletion
- Violation of least privilege principle

Secure Pattern - Method-Level Security

```
// SECURE: Method-level authorization with Spring Security
@RestController
@RequestMapping("/admin")
@PreAuthorize("hasRole('ADMIN')")  // Class-level admin requirement
public class AdminController {

    @Autowired
    private UserService userService;

    @GetMapping("/users/{userId}")
    @PreAuthorize("hasAuthority('USER_READ') or authentication.name ==
#userId.toString()")
    public ResponseEntity<User> getUser(@PathVariable Long userId, Authentication
auth) {
        // Secure: Authorization verified before execution
        User user = userService.findById(userId);

        // Additional business logic authorization
```

```
        if (!canAccessUser(auth, userId)) {
            throw new AccessDeniedException("Insufficient permissions");
        }

        return ResponseEntity.ok(user);
    }

    @DeleteMapping("/users/{userId}")
    @PreAuthorize("hasAuthority('USER_DELETE')")
    @PostAuthorize("returnObject.body == null or hasRole('SUPER_ADMIN')")
    public ResponseEntity<Void> deleteUser(@PathVariable Long userId,
Authentication auth) {
        // Secure: Multiple authorization checks
        auditService.logDeletionAttempt(auth.getName(), userId);
        userService.deleteUser(userId);
        return ResponseEntity.ok().build();
    }

    private boolean canAccessUser(Authentication auth, Long userId) {
        // Business logic: Users can access their own data, admins can access all
        return auth.getAuthorities().contains(new
SimpleGrantedAuthority("ROLE_ADMIN")) ||
               auth.getName().equals(userId.toString());
    }
}
```

3. Input Validation Vulnerabilities

Vulnerable Pattern - Missing Validation

```
// CRITICAL: No input validation
@RestController
public class DocumentController {

    @PostMapping("/upload")
    public ResponseEntity<String> uploadDocument(
            @RequestParam("file") MultipartFile file,
            @RequestParam("category") String category,
            @RequestParam("description") String description) {

        // Vulnerable: No validation of file type, size, or content
        String filename = file.getOriginalFilename();

        try {
            // Dangerous: Direct file system write without validation
            Files.copy(file.getInputStream(),
                       Paths.get("/uploads/" + filename));

            // Vulnerable: Storing unvalidated input in database
            documentService.createDocument(filename, category, description);
```

```
            return ResponseEntity.ok("File uploaded successfully");
        } catch (IOException e) {
            return ResponseEntity.status(500).body("Upload failed");
        }
    }
}
```

Secure Pattern - Comprehensive Validation

```
// SECURE: Comprehensive input validation
@RestController
@Validated
public class DocumentController {

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

    @PostMapping("/upload")
    public ResponseEntity<String> uploadDocument(
            @RequestParam("file") @NotNull MultipartFile file,
            @RequestParam("category") @Valid @Pattern(regexp = "^[a-zA-Z0-9_-]+$")
String category,
            @RequestParam("description") @Valid @Size(max = 1000) String
description,
            Authentication auth) {

        // Secure: Comprehensive file validation
        validateFile(file);
        validateUser(auth);

        // Secure: Generate safe filename
        String safeFilename = generateSecureFilename(file.getOriginalFilename());
        String sanitizedCategory = sanitizeInput(category);
        String sanitizedDescription = sanitizeInput(description);

        try {
            // Secure: Virus scanning before storage
            if (!virusScanner.isClean(file.getInputStream())) {
                throw new SecurityException("File failed security scan");
            }

            // Secure: Store in secure location with access controls
            Path uploadPath = secureFileStorage.store(file.getInputStream(),
safeFilename);

            // Secure: Store metadata with proper escaping
            Document document = documentService.createDocument(
                safeFilename, sanitizedCategory, sanitizedDescription,
auth.getName());
```

```
            // Secure: Audit logging
            auditService.logFileUpload(auth.getName(), safeFilename,
file.getSize());

            return ResponseEntity.ok("File uploaded successfully: " +
document.getId());

        } catch (IOException | SecurityException e) {
            auditService.logFailedUpload(auth.getName(),
file.getOriginalFilename(), e.getMessage());
            return ResponseEntity.status(500).body("Upload failed");
        }
    }

    private void validateFile(MultipartFile file) {
        if (file.isEmpty()) {
            throw new ValidationException("File cannot be empty");
        }

        if (file.getSize() > MAX_FILE_SIZE) {
            throw new ValidationException("File size exceeds maximum allowed");
        }

        String filename = file.getOriginalFilename();
        if (filename == null || filename.contains("..")) {
            throw new ValidationException("Invalid filename");
        }

        String extension = getFileExtension(filename).toLowerCase();
        if (!ALLOWED_EXTENSIONS.contains(extension)) {
            throw new ValidationException("File type not allowed");
        }
    }

    private String generateSecureFilename(String originalFilename) {
        String extension = getFileExtension(originalFilename);
        return UUID.randomUUID().toString() + "." + extension;
    }

    private String sanitizeInput(String input) {
        return input.replaceAll("[<>\"'%;()&+]", "");
    }
}
```

4. Deserialization Vulnerabilities

Vulnerable Pattern - Unsafe Deserialization

```
// CRITICAL: Unsafe Java deserialization
@RestController
public class SessionController {
```

```
    @PostMapping("/session/restore")
    public ResponseEntity<SessionData> restoreSession(@RequestBody byte[]
sessionData) {
        try {
            // DANGEROUS: Deserializing untrusted data
            ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(sessionData));
            SessionData session = (SessionData) ois.readObject();

            // Vulnerable to arbitrary code execution
            return ResponseEntity.ok(session);
        } catch (Exception e) {
            return ResponseEntity.badRequest().build();
        }
    }
}
```

Secure Pattern - Safe Deserialization

```
// SECURE: Safe deserialization with allowlisting
@RestController
public class SessionController {

    private final ObjectMapper objectMapper;
    private final Set<String> allowedClasses;

    public SessionController() {
        this.objectMapper = new ObjectMapper();
        // Configure Jackson to be restrictive
        objectMapper.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL,
JsonTypeInfo.As.PROPERTY);
        objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
true);

        // Allowlist of safe classes
        this.allowedClasses = Set.of(
            "com.amazon.security.SessionData",
            "java.lang.String",
            "java.util.Date",
            "java.lang.Long"
        );
    }

    @PostMapping("/session/restore")
    public ResponseEntity<SessionData> restoreSession(@RequestBody String
sessionJson, Authentication auth) {
        try {
            // Secure: Use JSON instead of Java serialization
            SessionData session = objectMapper.readValue(sessionJson,
SessionData.class);
```

```
            // Secure: Validate session belongs to authenticated user
            if (!session.getUserId().equals(auth.getName())) {
                throw new SecurityException("Session does not belong to
authenticated user");
            }

            // Secure: Validate session is not expired
            if (session.getExpirationTime().before(new Date())) {
                throw new SecurityException("Session has expired");
            }

            // Secure: Additional validation
            validateSessionData(session);

            return ResponseEntity.ok(session);
        } catch (Exception e) {
            auditService.logSuspiciousActivity(auth.getName(), "Invalid session
restoration attempt");
            return ResponseEntity.badRequest().build();
        }
    }

    private void validateSessionData(SessionData session) {
        if (session.getUserId() == null || session.getUserId().trim().isEmpty()) {
            throw new ValidationException("Invalid user ID in session");
        }

        if (session.getPermissions() == null) {
            throw new ValidationException("Session permissions cannot be null");
        }

        // Validate permissions are reasonable
        if (session.getPermissions().size() > 50) {
            throw new ValidationException("Too many permissions in session");
        }
    }
}
```

5. Cross-Site Scripting (XSS) Prevention

Vulnerable Pattern - Unescaped Output

```
// VULNERABLE: Direct output without escaping
@Controller
public class ProfileController {

    @GetMapping("/profile/{userId}")
    public String showProfile(@PathVariable String userId, Model model) {
        User user = userService.findById(userId);
```

```
        // Vulnerable: User-controlled data directly in model
        model.addAttribute("userName", user.getName());
        model.addAttribute("bio", user.getBio());
        model.addAttribute("website", user.getWebsite());

        return "profile"; // Thymeleaf template
    }
}
```

```
<!-- profile.html - Vulnerable template -->
<div class="profile">
    <h1 th:text="${userName}">Username</h1>  <!-- Safe: th:text escapes -->
    <div th:utext="${bio}">Bio content</div>  <!-- DANGEROUS: th:utext doesn't
escape -->
    <a th:href="${website}">Website</a>       <!-- DANGEROUS: href injection
possible -->
</div>
```

Secure Pattern - Proper Escaping and Validation

```
// SECURE: Proper input validation and output escaping
@Controller
public class ProfileController {

    @Autowired
    private HtmlSanitizer htmlSanitizer;

    @GetMapping("/profile/{userId}")
    public String showProfile(@PathVariable String userId, Model model,
Authentication auth) {
        User user = userService.findById(userId);

        // Secure: Validate user can access this profile
        if (!canViewProfile(auth, userId)) {
            throw new AccessDeniedException("Cannot view this profile");
        }

        // Secure: Sanitize all user-controlled content
        String safeName = htmlSanitizer.sanitize(user.getName());
        String safeBio = htmlSanitizer.sanitizeWithAllowedTags(user.getBio(),
            Set.of("p", "br", "strong", "em"));
        String safeWebsite = validateAndSanitizeUrl(user.getWebsite());

        model.addAttribute("userName", safeName);
        model.addAttribute("bio", safeBio);
        model.addAttribute("website", safeWebsite);
        model.addAttribute("isOwnProfile", auth.getName().equals(userId));

        return "profile";
```

```
    }

    private String validateAndSanitizeUrl(String url) {
        if (url == null || url.trim().isEmpty()) {
            return "";
        }

        try {
            URI uri = new URI(url);
            // Only allow HTTP/HTTPS protocols
            if (!"http".equals(uri.getScheme()) &&
!"https".equals(uri.getScheme())) {
                return "";
            }

            // Basic URL validation
            return uri.toString();
        } catch (URISyntaxException e) {
            return "";
        }
    }
}
```

Amazon-Specific Java Security Patterns

Spring Boot Security Configuration

```
// Amazon-style comprehensive security configuration
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class AmazonSecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .maximumSessions(1)
                .maxSessionsPreventsLogin(false)
                .and()
            .authorizeHttpRequests()
                .requestMatchers("/health", "/metrics").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/api/v1/**").authenticated()
                .anyRequest().authenticated()
                .and()
            .oauth2ResourceServer()
                .jwt()
                .jwtDecoder(jwtDecoder())
                .and()
```

```
            .csrf()

.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
                .and()
            .headers()
                .frameOptions().deny()
                .contentTypeOptions().and()
                .httpStrictTransportSecurity(hstsConfig ->
                    hstsConfig.maxAgeInSeconds(31536000).includeSubdomains(true))
                .and()
            .cors().configurationSource(corsConfigurationSource());

        return http.build();
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();

configuration.setAllowedOriginPatterns(Arrays.asList("https://*.amazon.com"));
        configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT",
"DELETE"));
        configuration.setAllowCredentials(true);
        configuration.setAllowedHeaders(Arrays.asList("*"));

        UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }
}
```

Audit Logging for Security Events

```
// Amazon-scale security audit logging
@Component
public class SecurityAuditLogger {

    private static final Logger auditLogger =
LoggerFactory.getLogger("SECURITY_AUDIT");

    @EventListener
    public void handleAuthenticationSuccess(AuthenticationSuccessEvent event) {
        String username = event.getAuthentication().getName();
        String ipAddress = getClientIpAddress();

        auditLogger.info("AUTHENTICATION_SUCCESS user={} ip={} timestamp={}",
            username, ipAddress, Instant.now());
    }

    @EventListener
```

```
    public void handleAuthenticationFailure(AbstractAuthenticationFailureEvent
event) {
        String username = event.getAuthentication().getName();
        String ipAddress = getClientIpAddress();
        String reason = event.getException().getMessage();

        auditLogger.warn("AUTHENTICATION_FAILURE user={} ip={} reason={}
timestamp={}",
            username, ipAddress, reason, Instant.now());
    }

    @EventListener
    public void handleAuthorizationFailure(AuthorizationDeniedEvent event) {
        String username = event.getAuthentication().getName();
        String resource = event.getSource().toString();

        auditLogger.warn("AUTHORIZATION_DENIED user={} resource={} timestamp={}",
            username, resource, Instant.now());
    }

    private String getClientIpAddress() {
        // Implementation to get real client IP behind load balancers
        HttpServletRequest request = ((ServletRequestAttributes)
            RequestContextHolder.currentRequestAttributes()).getRequest();

        String xForwardedFor = request.getHeader("X-Forwarded-For");
        if (xForwardedFor != null && !xForwardedFor.isEmpty()) {
            return xForwardedFor.split(",")[0].trim();
        }

        return request.getRemoteAddr();
    }
}
```

Interview Application

Sample Live Code Review Question:

"Here's a Spring Boot controller for user management. Find and fix the security issues."

Amazon-Quality Review Process:

1. **Quick Scan** (30 seconds): Look for obvious patterns
    - String concatenation in queries → SQL injection
    - Missing @PreAuthorize annotations → Authorization bypass
    - Direct user input in responses → XSS risk
2. **Systematic Analysis** (2 minutes): Apply AMAZON framework
    - **A**ssess business context (user management = high risk)
    - **M**ap attack vectors (input validation, authentication, authorization)

- **A**nalyze code patterns for common Java vulnerabilities

3. **Business Impact Communication** (1 minute):
    - "This SQL injection could expose all 50M customer records"
    - "Missing authorization allows any user to delete accounts"
    - "XSS vulnerability enables account takeover attacks"
4. **Secure Solution** (2 minutes): Provide working secure code
    - Parameterized queries with Spring Data JPA
    - Method-level security with proper roles
    - Input validation and output escaping

This Java-specific security review demonstrates both technical depth and business impact awareness that Amazon security engineers need for protecting customer data at scale.