Amazon Live Code Review Mastery

Overview

Master Amazon's secure code review interview through systematic vulnerability identification, scalable remediation strategies, and business impact assessment.

Amazon's Code Review Interview Format

Interview Structure (30 minutes of technical discussion)

- **Live screen sharing** - Analyze code in real-time
- **Multiple vulnerabilities** - Find 3-5 security issues quickly
- **Business impact discussion** - Quantify risk and customer impact
- **Remediation strategies** - Propose fixes that scale to millions of users
- **Follow-up questions** - Deep dive into specific vulnerabilities

Languages Tested

- **Java** - Spring Boot, authentication services, data processing
- **Python** - Flask/Django web apps, automation scripts, AWS integrations
- **JavaScript** - Node.js APIs, React components, authentication flows

Amazon's Evaluation Criteria

1. **Speed of identification** - Spot critical issues within 2-3 minutes
2. **Depth of analysis** - Understand full business and technical impact
3. **Scalable solutions** - Fixes that work at Amazon's 100M+ user scale
4. **Business communication** - Explain risks in terms of customer trust and revenue
5. **Systematic prevention** - Process improvements to prevent recurrence

The AMAZON Code Review Framework

Systematic 5-Minute Analysis Method

A - **Assess Context** (30 seconds)

- Business function and data sensitivity
- User scale and deployment environment
- Trust boundaries and privilege levels

M - **Map Attack Vectors** (60 seconds)

- Input validation points
- Authentication and authorization flows
- Data access patterns

A - **Analyze Vulnerabilities** (2 minutes)

- Common vulnerability patterns (OWASP Top 10)
- Framework-specific security issues
- Logic flaws and business rule bypasses

Z - **Zero in on Critical Issues** (60 seconds)

- Highest business impact vulnerabilities
- Customer trust and data exposure risks
- Infrastructure and availability threats

O - **Operationalize Solutions** (90 seconds)

- Immediate fixes for critical issues
- Scalable prevention strategies
- CI/CD and automation opportunities

N - **Navigate Business Impact** (30 seconds)

- Customer impact quantification
- Regulatory and compliance implications
- Cost of vulnerability vs. cost of fixes

Step 1: Context Assessment (30 seconds)

Questions to Ask Yourself:

- What's the business function of this code?
- Where does user input enter the system?
- What sensitive data is being processed?
- What are the trust boundaries?
- What's the deployment scale?

Example Internal Dialogue:

> "This looks like a customer authentication service. I see database queries, user input processing, and session management. Given Amazon's scale, this could process millions of authentication attempts daily. Any vulnerability here affects customer trust directly."

Step 2: Rapid Vulnerability Scanning (2-3 minutes)

Amazon's Most Common Issues (Prioritize These):

Authentication & Authorization Flaws

```
// CRITICAL: Look for these patterns immediately
// 1. Missing authorization checks
@RequestMapping("/admin/users/{userId}")
public User getUser(@PathVariable String userId) {
    return userService.findById(userId);  // No auth check!
}
```

```
// 2. Insecure session management
String sessionId = "session_" + System.currentTimeMillis(); // Predictable!

// 3. Password handling issues
if (user.getPassword().equals(requestPassword)) { // Plain text comparison!
    // login success
}
```

Injection Vulnerabilities

```
# SQL Injection - Spot immediately
query = f"SELECT * FROM users WHERE id = {user_id}"  # String interpolation!
cursor.execute(query)

# Command Injection - Critical for infrastructure
filename = request.args.get('file')
os.system(f"convert {filename} output.jpg")  # Direct command execution!

# NoSQL Injection - Common in modern apps
db.users.find({"$where": user_input})  # Dangerous $where usage!
```

Data Exposure Issues

```
// Information Disclosure
app.get('/api/debug', (req, res) => {
    res.json({
        environment: process.env,  // Exposes secrets!
        database_url: DB_CONNECTION,
        api_keys: CONFIG.keys
    });
});

// Sensitive data in logs
console.log(`Login attempt: ${username}:${password}`); // Logging credentials!
```

Step 3: Amazon-Scale Impact Analysis (2-3 minutes)

For each vulnerability, assess:

Immediate Risk:

- "This SQL injection could expose all customer records"
- "The authentication bypass affects global user base"
- "Command injection could compromise infrastructure serving millions"

Business Impact:

- **Customer Impact**: "50M customers could have accounts compromised"
- **Financial Impact**: "$8.25B potential breach cost at $165/record"
- **Regulatory Impact**: "GDPR fines up to 4% of annual revenue"
- **Operational Impact**: "Service outage affecting 100M+ requests/day"

Compliance Implications:

- "PCI DSS violation for payment processing"
- "SOX compliance failure for financial controls"
- "GDPR breach notification required within 72 hours"

Step 4: Scalable Remediation (3-4 minutes)

Amazon Expects Solutions That Scale:

Individual Fix

```
// Before: Vulnerable code
String query = "SELECT * FROM users WHERE id = " + userId;

// After: Immediate fix
String query = "SELECT * FROM users WHERE id = ?";
PreparedStatement stmt = connection.prepareStatement(query);
stmt.setString(1, userId);
```

Systematic Prevention

```
// Amazon-scale solution: Framework-level prevention
@Component
public class SecurityAuditInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response,
                             Object handler) throws Exception {

        // Automatic SQL injection detection
        String[] parameters = request.getParameterValues();
        for (String param : parameters) {
            if (containsSQLInjectionPatterns(param)) {
                logSecurityIncident(request, "SQL_INJECTION_ATTEMPT");
                response.setStatus(403);
                return false;
            }
        }
        return true;
    }
}
```

Process-Level Prevention

```
# CI/CD Integration for Scale
security_pipeline:
  static_analysis:
    - tool: "semgrep"
      rules: "amazon-security-rules"
      fail_on: "high,critical"

  dynamic_testing:
    - tool: "custom-scanner"
      endpoints: "all-authenticated"

  security_review:
    - required_for: "authentication,payment,admin"
    - reviewers: "security-champions"
```

Live Interview Scenarios

Scenario 1: Customer Authentication Service

Code Presented:

```
@RestController
public class AuthController {

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody LoginRequest request) {

        String sql = "SELECT * FROM users WHERE email = '" +
                     request.getEmail() + "' AND password = '" +
                     request.getPassword() + "'";

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

        if (!users.isEmpty()) {
            User user = users.get(0);
            String token = JWT.create()
                .withSubject(user.getEmail())
                .withExpiresAt(new Date(System.currentTimeMillis() + 86400000))
                .sign(Algorithm.HMAC256("secretkey"));

            return ResponseEntity.ok(new AuthResponse(token, user.getRole()));
        }

        return ResponseEntity.status(401).body("Login failed for " +
request.getEmail());
    }
}
```

Amazon-Quality Response (5-minute analysis):

Immediate Issues Identified (30 seconds):

> "I see several critical security vulnerabilities here. First, there's a SQL injection vulnerability in the login query using string concatenation. Second, passwords are being compared in plain text. Third, there's information disclosure in the error message. Fourth, I see a hardcoded JWT secret."

Risk Assessment (1 minute):

> "At Amazon's scale, this authentication service could handle millions of login attempts daily. The SQL injection alone could expose our entire customer database - potentially 300M+ customer records worth $49.5B in breach costs. The plain text passwords violate basic security standards and could trigger massive compliance violations."

Scalable Solutions (2 minutes):

> "For immediate remediation: implement parameterized queries, add bcrypt password hashing, use generic error messages, and externalize JWT secrets to AWS Secrets Manager. For systemic prevention: I'd implement automated SAST scanning in CI/CD that specifically catches string concatenation in SQL contexts, add security unit tests for authentication flows, and require security champion review for any authentication-related code changes."

Business Communication (1 minute):

> "The business impact is severe - this could trigger a company-threatening security incident. However, the fixes are straightforward and can be implemented within a sprint. I recommend treating this as a P0 security issue requiring immediate hotfix deployment, followed by comprehensive audit of all authentication-related code across our services."

Process Improvement (30 seconds):

> "To prevent this systematically, I'd recommend implementing security-focused code review checklists, automated security testing in CI/CD, and security champion training for all developers working on authentication services."

Scenario 2: File Processing Service

Code Presented:

```
from flask import Flask, request, send_file
import os
import subprocess

app = Flask(__name__)

@app.route('/process', methods=['POST'])
def process_file():
    filename = request.form.get('filename')
    operation = request.form.get('operation', 'resize')
```

```
    # Save uploaded file
    file = request.files['file']
    filepath = os.path.join('/uploads', filename)
    file.save(filepath)

    # Process file based on operation
    if operation == 'resize':
        cmd = f"convert {filepath} -resize 800x600 /processed/{filename}"
    elif operation == 'compress':
        cmd = f"gzip {filepath} -c > /processed/{filename}.gz"
    else:
        cmd = f"file {filepath}"

    result = subprocess.run(cmd, shell=True, capture_output=True, text=True)

    return {
        'status': 'processed',
        'output': result.stdout,
        'command': cmd  # For debugging
    }

@app.route('/download/<path:filename>')
def download_file(filename):
    return send_file(f'/processed/{filename}')
```

Amazon-Quality Analysis:

Critical Vulnerabilities (45 seconds):

> "This code has multiple critical vulnerabilities: command injection through user-controlled filename and operation parameters, path traversal in both upload and download endpoints, and information disclosure by returning the executed command. There's also unrestricted file upload with no validation."

Amazon-Scale Impact (45 seconds):

> "At Amazon's processing volumes, this could enable attackers to execute arbitrary commands on servers processing millions of customer files. The path traversal could expose AWS credentials or other secrets. If this service processes customer data, we're looking at massive data exposure and infrastructure compromise affecting global operations."

Immediate Fixes (90 seconds):

> "Critical fixes: implement input validation with whitelists for filenames and operations, use secure_filename() for path handling, replace shell=True with parameterized subprocess calls, remove command echoing in responses, and add file type validation. For the download endpoint, validate that requested files are within the allowed directory and sanitize the path parameter."

Scalable Prevention (60 seconds):

> "Systematically, I'd implement a secure file processing framework with built-in validation, sandboxed execution environments using containers, automated security scanning for command injection patterns, and mandatory security review for any file processing code. We should also implement monitoring for unusual command execution patterns and automated blocking of suspicious activities."

Amazon Code Review Best Practices

Speed Optimization

- **Pattern Recognition**: Train your eye to spot common vulnerability patterns instantly
- **Prioritization**: Focus on authentication, data access, and input handling first
- **Framework Knowledge**: Understand common frameworks and their security implications

Business Communication

- **Quantify Risk**: Always include potential business impact numbers
- **Customer Focus**: Frame security issues in terms of customer impact
- **Operational Impact**: Consider how vulnerabilities affect service reliability

Scalable Thinking

- **Process Integration**: How does this fit into CI/CD pipelines?
- **Automation Opportunities**: What can be automated to prevent recurrence?
- **Monitoring**: How would we detect this vulnerability in production?

Amazon-Specific Considerations

- **Service Scale**: Solutions must work for millions of requests
- **Global Deployment**: Consider multi-region security implications
- **AWS Integration**: Leverage AWS security services where appropriate
- **Customer Trust**: Every security decision impacts customer confidence

This Amazon-focused approach to code review demonstrates the rapid identification, business impact assessment, and scalable solution thinking that Amazon values in their security engineers.