OWASP Top 10 2021 Security Vulnerabilities in Scala Applications

This comprehensive guide examines each OWASP Top 10 2021 vulnerability category specifically for Scala applications using Spring Boot and Apache Pekko HTTP frameworks. The analysis provides detailed vulnerable code patterns, secure implementations, and Scala-specific security considerations across both Scala 2 and Scala 3.

A01: Broken Access Control

Ranking: #1 in OWASP Top 10 2021 (moved up from #5 in 2017) OWASP Foundation +2 **Prevalence**: 94% of applications tested had some form of broken access control OWASP **Impact**: Most occurrences in contributed dataset with over 318,000 instances OWASP Foundation +2

Scala Application Context

Broken Access Control manifests in Scala applications through failures in authentication, authorization, and permission enforcement. Apache The most common issues include Insecure Direct Object References (IDOR), missing function-level access controls, privilege escalation vulnerabilities, and CORS misconfigurations. OWASP Foundation

Vulnerable Code Patterns

Scala 2 with Spring Boot - Missing Authorization

```scala
```

```
@RestController
class VulnerableUserController @Autowired()(userService: UserService) {

 // VULNERABILITY: No authorization check - any authenticated user can access any user
 @GetMapping(Array("/users/{userId}"))
 def getUser(@PathVariable userId: String): ResponseEntity[User] = {
  val user = userService.findById(userId)
  ResponseEntity.ok(user)
 }

 // VULNERABILITY: Missing role-based authorization
 @PostMapping(Array("/admin/users"))
 def createUser(@RequestBody user: User): ResponseEntity[User] = {
  val created = userService.create(user)
  ResponseEntity.ok(created)
 }
}
```

Scala 3 with Spring Boot - Insecure Direct Object Reference

```
scala
```

```
@RestController
class VulnerableDocumentController @Autowired()(documentService: DocumentService):

 @GetMapping(Array("/documents"))
 def getDocuments(@RequestParam ownerId: Option[String]): ResponseEntity[List[Document]] =
  // VULNERABILITY: Insecure Direct Object Reference
  val documents = ownerId match
   case Some(id) => documentService.findByOwner(id) // Any user can query any owner's docs
   case None => documentService.findAll()
  ResponseEntity.ok(documents.asJava)
```

Scala 2 with Pekko HTTP - No Authentication Required

```scala
```

```
import org.apache.pekko.http.scaladsl.server.Directives._

object VulnerableUserRoutes {
  def routes: Route = pathPrefix("api" / "users") {
    get {
      // VULNERABILITY: No authentication required
      path(Segment) { userId =>
        complete(s"User data for: $userId") // Exposes any user data
      }
    } ~
    post {
      // VULNERABILITY: Missing authorization for admin operations
      path("admin" / "create") {
        entity(as[String]) { userData =>
          complete("User created")
        }
      }
    }
  }
}
```

Secure Implementation

Scala 2 with Spring Boot - Proper Access Control

```
scala
```

```
@RestController
class SecureUserController @Autowired()(userService: UserService) {

 @GetMapping(Array("/users/{userId}"))
 @PreAuthorize("@userService.canAccess(authentication.name, #userId)")
 def getUser(@PathVariable userId: String, auth: Authentication): ResponseEntity[User] = {
  val user = userService.findById(userId)
  ResponseEntity.ok(user)
 }

 @PostMapping(Array("/admin/users"))
 @PreAuthorize("hasRole('ADMIN')")
 def createUser(@RequestBody user: User): ResponseEntity[User] = {
  val created = userService.create(user)
  ResponseEntity.ok(created)
 }
}

@Configuration
@EnableMethodSecurity(prePostEnabled = true)
class SecurityConfig extends WebSecurityConfigurerAdapter {
 override def configure(http: HttpSecurity): Unit = {
  http
   .authorizeHttpRequests(authz => authz
    .requestMatchers("/admin/**").hasRole("ADMIN")
    .requestMatchers("/users/**").authenticated()
    .anyRequest().permitAll())
   .oauth2ResourceServer(_.jwt())
 }
}
```

Scala 3 with Pekko HTTP - Complete Authorization

scala

```scala
import org.apache.pekko.http.scaladsl.server.Directives.*

object SecureUserRoutes {
 def routes: Route = pathPrefix("api" / "users") {
  authenticateOAuth2("realm", validateToken) { user =>
   get {
    path(Segment) { userId =>
     authorize(user.id == userId || user.hasRole("ADMIN")) {
      complete(s"User data for: $userId")
     }
    }
   } ~
   post {
    path("admin" / "create") {
     authorize(user.hasRole("ADMIN")) {
      entity(as[String]) { userData =>
       complete("User created")
      }
     }
    }
   }
  }
 }

 def validateToken(credentials: Credentials): Option[User] = credentials match {
  case Credentials.Provided(token) => JWTService.validateToken(token)
  case _ => None
 }
}
```

A02: Cryptographic Failures

Ranking: #2 in OWASP Top 10 2021 (up from #3 in 2017) OWASP OWASP Foundation **Focus**: Failures related to cryptography leading to sensitive data exposure OWASP +2 **Key Issues**: CWE-259 (Hard-coded Password), CWE-327 (Broken Crypto Algorithm), CWE-331 (Insufficient Entropy) OWASP OWASP Foundation

Scala Application Context

Cryptographic failures in Scala applications often stem from improper use of Java cryptographic APIs, weak key generation and management, insecure storage of sensitive data, and use of deprecated cryptographic algorithms. Kodemsecurity

Vulnerable Code Patterns

Scala 2 with Spring Boot - Weak Cryptography

```scala
```

```
@Service
class VulnerableCryptoService {

  // VULNERABILITY: Using MD5 for password hashing
  def hashPassword(password: String): String = {
    import java.security.MessageDigest
    val md = MessageDigest.getInstance("MD5")
    md.digest(password.getBytes()).map("%02x".format(_)).mkString
  }

  // VULNERABILITY: Hardcoded encryption key
  private val secretKey = "mySecretKey123"

  def encryptData(data: String): String = {
    import javax.crypto.Cipher
    import javax.crypto.spec.SecretKeySpec
    val key = new SecretKeySpec(secretKey.getBytes(), "AES")
    val cipher = Cipher.getInstance("AES")
    cipher.init(Cipher.ENCRYPT_MODE, key)
    new String(cipher.doFinal(data.getBytes()))
  }
}
```

Scala 2 with Pekko HTTP - Insecure Encryption Mode

```
scala
```

```
import javax.crypto.Cipher
import javax.crypto.spec.SecretKeySpec

object VulnerableCryptoService {

 // VULNERABILITY: ECB mode and fixed key
 private val fixedKey = "1234567890123456".getBytes()

 def encryptPayload(data: String): String = {
  val key = new SecretKeySpec(fixedKey, "AES")
  val cipher = Cipher.getInstance("AES/ECB/PKCS5Padding") // ECB is insecure
  cipher.init(Cipher.ENCRYPT_MODE, key)
  java.util.Base64.getEncoder.encodeToString(cipher.doFinal(data.getBytes()))
 }

 // VULNERABILITY: Using deprecated SHA1
 def hashData(data: String): String = {
  import java.security.MessageDigest
  val sha1 = MessageDigest.getInstance("SHA1")
  sha1.digest(data.getBytes()).map("%02x".format(_)).mkString
 }
}
```

Secure Implementation

Scala 2/3 with Spring Boot - Strong Cryptography

```
scala
```

```
@Service
class SecureCryptoService @Autowired()(passwordEncoder: BCryptPasswordEncoder) {

 // SECURE: Using BCrypt with proper work factor
 def hashPassword(password: String): String = {
  passwordEncoder.encode(password)
 }

 // SECURE: Using Spring Security Crypto for key management
 @Value("${app.encryption.key}")
 private val encryptionKey: String = _

 def encryptData(data: String): String = {
  import org.springframework.security.crypto.encrypt.Encryptors
  val textEncryptor = Encryptors.text(encryptionKey, KeyGenerators.string().generateKey())
  textEncryptor.encrypt(data)
 }
}

@Configuration
class CryptoConfig {
 @Bean
 def passwordEncoder(): BCryptPasswordEncoder = new BCryptPasswordEncoder(12)

 @Bean
 def keyGenerator(): StringKeyGenerator = KeyGenerators.string()
}
```

Pekko HTTP - AES-GCM with Proper IV

```
scala
```

```
import javax.crypto.{Cipher, KeyGenerator}
import javax.crypto.spec.{GCMParameterSpec, SecretKeySpec}
import java.security.SecureRandom

object SecureCryptoService {

 private val secureRandom = new SecureRandom()

 // SECURE: Using AES-GCM with proper IV generation
 def encryptPayload(data: String, secretKey: Array[Byte]): String = {
  val cipher = Cipher.getInstance("AES/GCM/NoPadding")
  val iv = new Array[Byte](12)
  secureRandom.nextBytes(iv)

  val key = new SecretKeySpec(secretKey, "AES")
  val spec = new GCMParameterSpec(128, iv)
  cipher.init(Cipher.ENCRYPT_MODE, key, spec)

  val encrypted = cipher.doFinal(data.getBytes("UTF-8"))
  java.util.Base64.getEncoder.encodeToString(iv ++ encrypted)
 }

 // SECURE: Using SHA-256
 def hashData(data: String): String = {
  import java.security.MessageDigest
  val sha256 = MessageDigest.getInstance("SHA-256")
  sha256.digest(data.getBytes("UTF-8")).map("%02x".format(_)).mkString
 }
}
```

A03: Injection

Ranking: #3 in OWASP Top 10 2021 (down from #1 in 2017) OWASP OWASP Foundation
Prevalence: 94% of applications tested for injection vulnerabilities OWASP OWASP Foundation
Scope: Now includes Cross-Site Scripting (XSS) in this category OWASP OWASP Foundation

Scala Application Context

Injection vulnerabilities in Scala applications commonly occur through improper use of string interpolation in SQL queries, unsafe deserialization practices, inadequate input validation, and command injection through system calls. Kodemsecurity

Vulnerable Code Patterns

Scala 2 with Spring Boot - SQL Injection

```scala
```

```
@Repository
class VulnerableUserRepository @Autowired()(jdbcTemplate: JdbcTemplate) {

 // VULNERABILITY: SQL Injection through string concatenation
 def findUserByEmail(email: String): Option[User] = {
  val query = s"SELECT * FROM users WHERE email = '$email'"
  val users = jdbcTemplate.query(query, new BeanPropertyRowMapper[User](classOf[User]))
  users.asScala.headOption
 }
}

@RestController
class VulnerableSearchController {

 // VULNERABILITY: Reflected XSS through unsanitized input
 @GetMapping(Array("/search"))
 def search(@RequestParam query: String): String = {
  s"""<html><body>
     <h1>Search Results for: $query</h1>
     <p>No results found</p>
     </body></html>"""
 }
}
```

Scala 3 with Spring Boot - Slick Injection

```
scala
```

```
@Repository
class VulnerableOrderRepository @Autowired()(jdbcTemplate: JdbcTemplate):

 // VULNERABILITY: Using Slick with #$ interpolation
 def getOrdersByStatus(status: String): List[Order] =
  import slick.jdbc.MySQLProfile.api._
  val db = Database.forConfig("mydb")
  val query = sql"SELECT * FROM orders WHERE status = #$status".as[Order]
  // #$ performs literal interpolation - vulnerable to SQL injection
  Await.result(db.run(query), Duration.Inf).toList
```

Scala 2 with Pekko HTTP - Command Injection

```
scala
```

```
import org.apache.pekko.http.scaladsl.server.Directives._

object VulnerableApiRoutes {

  def routes: Route = pathPrefix("api") {
    get {
      path("users") {
        parameter("filter") { filter =>
          // VULNERABILITY: Command injection
          val command = s"grep '$filter' /var/log/users.log"
          val result = sys.process.Process(command).!!
          complete(result)
        }
      }
    }
  }
}
```

Secure Implementation

Scala 2/3 with Spring Boot - Parameterized Queries

```
scala
```

```
@Repository
class SecureUserRepository @Autowired()(jdbcTemplate: JdbcTemplate) {

 // SECURE: Using parameterized queries
 def findUserByEmail(email: String): Option[User] = {
  val query = "SELECT * FROM users WHERE email = ?"
  val users = jdbcTemplate.query(query, new BeanPropertyRowMapper[User](classOf[User]), email)
  users.asScala.headOption
 }
}

@RestController
class SecureSearchController {

 // SECURE: Input sanitization and output encoding
 @GetMapping(Array("/search"))
 def search(@RequestParam query: String): String = {
  val sanitizedQuery = StringEscapeUtils.escapeHtml4(query)
  s"""<html><body>
     <h1>Search Results for: $sanitizedQuery</h1>
     <p>No results found</p>
     </body></html>"""
 }
}
```

Scala 3 - Safe Slick Queries

```
scala
```

```
@Repository
class SecureOrderRepository @Autowired()(jdbcTemplate: JdbcTemplate):

 // SECURE: Using Slick with proper parameterization
 def getOrdersByStatus(status: String): List[Order] =
  import slick.jdbc.MySQLProfile.api._

  // Input validation
  require(isValidStatus(status), "Invalid order status")

  val db = Database.forConfig("mydb")
  val query = sql"SELECT * FROM orders WHERE status = $status".as[Order]
  // $ interpolation uses bind variables - safe from SQL injection
  Await.result(db.run(query), Duration.Inf).toList

 private def isValidStatus(status: String): Boolean =
  Set("PENDING", "COMPLETED", "CANCELLED").contains(status.toUpperCase)
```

A04: Insecure Design

New Category: Represents architectural and design flaws that expose applications to security threats (OWASP) **Focus**: Missing threat modeling, insufficient business logic validation, lack of secure design patterns (OWASP) (OWASP Foundation)

Scala Application Context

Insecure Design manifests as missing threat modeling, insufficient business logic validation, lack of secure design patterns, missing security controls in critical flows, and inadequate session management design.

Vulnerable Code Patterns

Scala 2 with Spring Boot - Missing Business Logic Controls

```scala
@RestController
class UserController @Autowired()(userService: UserService) {

 // VULNERABLE: No rate limiting, business logic flaws
 @PostMapping("/reset-password")
 def resetPassword(@RequestParam email: String): ResponseEntity[String] = {
  // Missing rate limiting - allows brute force
  // Missing validation of business rules
  userService.sendResetEmail(email)
  ResponseEntity.ok("Reset email sent")
 }

 // VULNERABLE: Trust boundary violation
 @PostMapping("/transfer")
 def transferMoney(@RequestParam fromAccount: String,
                   @RequestParam toAccount: String,
                   @RequestParam amount: BigDecimal): ResponseEntity[String] = {
  // No validation of account ownership
  // Missing transaction limits
  bankService.transfer(fromAccount, toAccount, amount)
  ResponseEntity.ok("Transfer completed")
 }
}
```

Secure Implementation with Business Logic

Complete Security Design Pattern

scala

```
@RestController
class SecureUserController @Autowired()(
  userService: UserService,
  rateLimitService: RateLimitService,
  auditService: AuditService
) {

  @PostMapping("/reset-password")
  def resetPassword(@RequestParam email: String,
                    httpRequest: HttpServletRequest): ResponseEntity[String] = {
    val clientIp = httpRequest.getRemoteAddr

    // Implement rate limiting
    if (!rateLimitService.isAllowed(s"password-reset:$clientIp", 3, Duration.ofHours(1))) {
      auditService.log(AuditEvent.RATE_LIMIT_EXCEEDED, clientIp, email)
      return ResponseEntity.status(429).body("Too many requests")
    }

    // Validate email exists before revealing information
    if (userService.emailExists(email)) {
      // Use secure token-based reset instead of security questions
      val resetToken = secureTokenService.generateResetToken(email)
      emailService.sendSecureResetLink(email, resetToken)
      auditService.log(AuditEvent.PASSWORD_RESET_REQUESTED, clientIp, email)
    }

    // Always return same response to prevent email enumeration
    ResponseEntity.ok("If the email exists, a reset link has been sent")
  }

  @PostMapping("/transfer")
  def transferMoney(@RequestParam fromAccount: String,
```

```
            @RequestParam toAccount: String,
            @RequestParam amount: BigDecimal,
            authentication: Authentication): ResponseEntity[String] = {

  val userId = authentication.getName

  // Validate account ownership
  if (!accountService.isOwner(userId, fromAccount)) {
   auditService.log(AuditEvent.UNAUTHORIZED_TRANSFER_ATTEMPT, userId, fromAccount)
   return ResponseEntity.status(403).body("Access denied")
  }

  // Apply business rules and limits
  val dailyLimit = accountService.getDailyTransferLimit(fromAccount)
  val todayTransfers = transferService.getTodayTransfers(fromAccount)

  if (todayTransfers + amount > dailyLimit) {
   return ResponseEntity.badRequest().body("Daily transfer limit exceeded")
  }

  // Implement transaction verification for large amounts
  if (amount > BigDecimal(10000)) {
   val verificationId = verificationService.requireTwoFactorAuth(userId)
   return ResponseEntity.accepted().body(s"Verification required: $verificationId")
  }

  bankService.transfer(fromAccount, toAccount, amount)
  auditService.log(AuditEvent.TRANSFER_COMPLETED, userId, s"$fromAccount->$toAccount:$amount")
  ResponseEntity.ok("Transfer completed")
 }
}
```

A05: Security Misconfiguration

Ranking: #5 in OWASP Top 10 2021 (OWASP Foundation) **Focus**: Improperly implemented, maintained, or default security settings

Vulnerable Configuration Patterns

Scala 2 Spring Boot - Insecure Security Configuration

```scala
```

```
// VULNERABLE: Insecure Spring Security configuration
@Configuration
@EnableWebSecurity
class InsecureSecurityConfig extends WebSecurityConfigurerAdapter {

 override def configure(http: HttpSecurity): Unit = {
  http
   // VULNERABLE: CSRF disabled globally
   .csrf().disable()
   // VULNERABLE: Permits all requests
   .authorizeRequests().anyRequest().permitAll()
   // VULNERABLE: No security headers
   .and()
   // VULNERABLE: Basic auth without HTTPS requirement
   .httpBasic()
 }

 // VULNERABLE: Default password
 @Bean
 override def userDetailsService(): UserDetailsService = {
  val user = User.withDefaultPasswordEncoder()
   .username("admin")
   .password("password") // VULNERABLE: Default password
   .roles("ADMIN")
   .build()
  new InMemoryUserDetailsManager(user)
 }
}
```

Secure Configuration Implementation

Comprehensive Security Configuration

```scala
```

```
@Configuration
@EnableWebSecurity
class SecureSecurityConfig extends WebSecurityConfigurerAdapter {

 override def configure(http: HttpSecurity): Unit = {
  http
   // Enable CSRF protection
   .csrf()
    .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
   .and()
   // Secure authorization rules
   .authorizeRequests()
    .antMatchers("/public/**").permitAll()
    .antMatchers("/admin/**").hasRole("ADMIN")
    .anyRequest().authenticated()
   .and()
   // Add security headers
   .headers(headers => {
    headers
     .frameOptions().deny()
     .contentTypeOptions()
     .httpStrictTransportSecurity(hstsConfig => {
      hstsConfig
       .maxAgeInSeconds(31536000)
       .includeSubdomains(true)
       .preload(true)
     })
     .addHeaderWriter(new StaticHeaderWriter("Content-Security-Policy",
      "default-src 'self'; script-src 'self' 'unsafe-inline'; object-src 'none';"))
   })
   .and()
   // Secure session management
```

```
      .sessionManagement()
       .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
       .maximumSessions(1)
       .maxSessionsPreventsLogin(true)
      .and()
      // Force HTTPS
      .requiresChannel()
       .anyRequest().requiresSecure()
  }

  // Secure password encoder
  @Bean
  def passwordEncoder(): PasswordEncoder = new BCryptPasswordEncoder(12)
}
```

A06: Vulnerable and Outdated Components

Ranking: #6 in OWASP Top 10 2021 **Focus**: Using components with known security vulnerabilities or no longer maintained

Build Tool Security Configuration

SBT Security Setup

```
scala
```

```
// project/plugins.sbt
addSbtPlugin("net.vonbuchholtz" % "sbt-dependency-check" % "5.1.0")
addSbtPlugin("com.github.sbt" % "sbt-dependency-graph" % "0.13.4")
addSbtPlugin("com.github.sbt" % "sbt-updates" % "0.6.4")

// build.sbt
ThisBuild / scalaVersion := "2.13.14" // Use latest stable

// Enable dependency vulnerability checking
dependencyCheckAutoUpdate := Some(true)
dependencyCheckFormat := "All"
dependencyCheckOutputDirectory := Some(file("target/dependency-check"))

// Secure dependency versions
libraryDependencies ++= Seq(
  "org.springframework.boot" % "spring-boot-starter-web" % "3.1.5",
  "com.fasterxml.jackson.core" % "jackson-databind" % "2.15.3",
  "org.apache.commons" % "commons-lang3" % "3.13.0",
  "mysql" % "mysql-connector-java" % "8.1.0",
  "ch.qos.logback" % "logback-classic" % "1.4.11", // Avoid Log4j
  "org.yaml" % "snakeyaml" % "2.2"
)

// Security tasks
addCommandAlias("securityCheck", ";dependencyCheck;dependencyUpdates")
addCommandAlias("vulnerabilityReport", "dependencyCheck")
```

Bazel Security Configuration

```
python
```

```
# WORKSPACE - Secure dependency management
workspace(name = "secure_scala_app")

# Use Bazel rules with integrity hashes
http_archive(
    name = "io_bazel_rules_scala",
    sha256 = "4df73c4d6a14b0d8b1c7bb7d44b7d8e4b5b6d7f", # Pin with known hash
    strip_prefix = "rules_scala-5.0.0",
    url = "https://github.com/bazelbuild/rules_scala/archive/v5.0.0.tar.gz",
)

# Maven dependencies with vulnerability scanning
maven_install(
    artifacts = [
        "org.springframework.boot:spring-boot-starter-web:3.1.5",
        "org.apache.pekko:pekko-actor-typed_2.13:1.0.2",
        "org.slf4j:slf4j-api:2.0.9",
    ],
    repositories = [
        "https://repo1.maven.org/maven2",
    ],
    fail_on_missing_checksum = True, # Require integrity verification
    version_conflict_policy = "pinned", # Prevent version conflicts
)
```

A07: Identification and Authentication Failures

Ranking: #7 in OWASP Top 10 2021 (previously "Broken Authentication") OWASP owasp **Focus**: Failures related to identification, authentication, and session management OWASP

Vulnerable Authentication Patterns

Scala 2 with Spring Boot - Weak Authentication

```scala
@RestController
class AuthController @Autowired()(val userService: UserService) {

 @PostMapping("/login")
 def login(@RequestParam username: String, @RequestParam password: String,
         request: HttpServletRequest): ResponseEntity[String] = {
  // VULNERABLE: No rate limiting
  if (userService.validateUser(username, password)) {
   val session = request.getSession(true)
   session.setAttribute("user", username)
   // VULNERABLE: Session ID exposed in response
   ResponseEntity.ok(s"Login successful. Session: ${session.getId}")
  } else {
   ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid credentials")
  }
 }
}

// VULNERABLE: Weak password validation
class UserService {
 def validateUser(username: String, password: String): Boolean = {
  // VULNERABLE: No protection against timing attacks
  val user = findUser(username)
  user.isDefined && user.get.password == password // Plain text comparison
 }
}
```

Secure Authentication Implementation

Comprehensive Secure Authentication

```scala
```

```
@RestController
class SecureAuthController @Autowired()(
  val userService: UserService,
  val rateLimitService: RateLimitService
) {

  @PostMapping("/login")
  def login(@RequestParam username: String, @RequestParam password: String,
            request: HttpServletRequest): ResponseEntity[String] = {

    val clientIp = request.getRemoteAddr

    // SECURE: Rate limiting
    if (!rateLimitService.isAllowed(clientIp)) {
      return ResponseEntity.status(HttpStatus.TOO_MANY_REQUESTS)
        .body("Rate limit exceeded")
    }

    val result = userService.validateUser(username, password)
    result match {
      case ValidationResult.Success(user) =>
        val session = request.getSession(true)
        session.setAttribute("user", user.username)
        session.setMaxInactiveInterval(1800) // 30 minutes
        ResponseEntity.ok("Login successful") // No session ID exposed

      case ValidationResult.Failure =>
        // SECURE: Same response time to prevent user enumeration
        Thread.sleep(100)
        ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid credentials")
    }
  }
```

```
}

// SECURE: User validation service
class UserService {
  def validateUser(username: String, password: String): ValidationResult = {
    findUser(username) match {
      case Some(user) if isValidPassword(password, user.passwordHash) =>
        ValidationResult.Success(user)
      case _ =>
        // SECURE: Constant time operation to prevent timing attacks
        BCrypt.checkpw("dummy", "$2a$10$dummy.hash.to.prevent.timing")
        ValidationResult.Failure
    }
  }

  private def isValidPassword(password: String, hash: String): Boolean = {
    BCrypt.checkpw(password, hash)
  }
}

sealed trait ValidationResult
object ValidationResult {
  case class Success(user: User) extends ValidationResult
  case object Failure extends ValidationResult
}
```

A08: Software and Data Integrity Failures

New Category: Focuses on assumptions about software updates, critical data, and CI/CD pipelines without verifying integrity (OWASP) (owasp) **Includes**: Insecure deserialization vulnerabilities (Kodemsecurity +2) (formerly A08:2017) (OWASP) (OWASP Foundation)

Vulnerable Deserialization Patterns

Scala 2 with Spring Boot - Unsafe Deserialization

```scala
```

```
@RestController
class DataController {

 @PostMapping("/process-data")
 def processData(@RequestBody data: Array[Byte]): ResponseEntity[String] = {
  try {
   // VULNERABLE: Deserializing untrusted data
   val ois = new ObjectInputStream(new ByteArrayInputStream(data))
   val obj = ois.readObject()

   obj match {
    case userData: UserData =>
     processUserData(userData)
     ResponseEntity.ok("Data processed successfully")
    case _ =>
     ResponseEntity.badRequest().body("Invalid data format")
   }
  } catch {
   case e: Exception =>
    ResponseEntity.internalServerError().body("Processing failed")
  }
 }
}

// VULNERABLE: Serializable class without validation
@SerialVersionUID(1L)
case class UserData(name: String, email: String, preferences: Map[String, Any]) extends Serializable {
 // VULNERABLE: Custom readObject without validation
 private def readObject(ois: ObjectInputStream): Unit = {
  ois.defaultReadObject()
  // No integrity checks
```

```
  }
}
```

Secure Data Integrity Implementation

Safe Deserialization with Signature Verification

```scala
```

```
@RestController
class SecureDataController(
  val signatureService: SignatureService,
  val allowedClasses: Set[String]
) {

  @PostMapping("/process-data")
  def processData(@RequestBody signedData: SignedData): ResponseEntity[String] = {
    try {
      // SECURE: Verify signature first
      if (!signatureService.verifySignature(signedData)) {
        return ResponseEntity.status(HttpStatus.FORBIDDEN).body("Invalid signature")
      }

      // SECURE: Use safe deserialization
      val result = safeDeserialize(signedData.data)
      result match {
        case Success(userData: UserData) =>
          processUserData(userData)
          ResponseEntity.ok("Data processed successfully")
        case Failure(error) =>
          logger.warn(s"Deserialization failed: ${error.getMessage}")
          ResponseEntity.badRequest().body("Invalid data")
      }
    } catch {
      case e: SecurityException =>
        logger.error("Security violation in data processing", e)
        ResponseEntity.status(HttpStatus.FORBIDDEN).body("Security violation")
    }
  }

  // SECURE: Safe deserialization with class allowlist
```

```
  private def safeDeserialize(data: Array[Byte]): Try[AnyRef] = {
    Try {
      val safeStream = new SafeObjectInputStream(new ByteArrayInputStream(data), allowedClasses)
      safeStream.readObject()
    }
  }
}

// SECURE: Safe deserialization class
class SafeObjectInputStream(inputStream: InputStream, allowedClasses: Set[String])
    extends ObjectInputStream(inputStream) {

  override def resolveClass(desc: ObjectStreamClass): Class[_] = {
    val className = desc.getName
    if (!allowedClasses.contains(className)) {
      throw new InvalidClassException("Unauthorized deserialization attempt", className)
    }
    super.resolveClass(desc)
  }
}

// SECURE: Data classes with validation
case class SignedData(data: Array[Byte], signature: Array[Byte], timestamp: Long)

case class UserData(name: String, email: String) {
  // SECURE: Validation in constructor
  require(name.nonEmpty && name.length <= 100, "Invalid name")
  require(email.matches("^[A-Za-z0-9+_.-]+@(.+)$"), "Invalid email format")
}
```

A09: Security Logging and Monitoring Failures

Ranking: #9 in OWASP Top 10 2021 **Focus**: Insufficient logging, detection, monitoring, and active response

Vulnerable Logging Patterns

Scala 2 with Spring Boot - Insecure Logging

```scala
```

```
@RestController
class UserController(userService: UserService) {
  private val logger = LoggerFactory.getLogger(getClass)

  @PostMapping("/login")
  def login(@RequestParam username: String, @RequestParam password: String): ResponseEntity[String] = {
    // VULNERABLE: Logging sensitive data
    logger.info(s"Login attempt for user: $username with password: $password")

    val user = userService.authenticate(username, password)
    if (user.isDefined) {
      // VULNERABLE: No logging of successful authentication
      ResponseEntity.ok("Login successful")
    } else {
      // VULNERABLE: Generic error, no details for monitoring
      logger.error("Login failed")
      ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Login failed")
    }
  }

  @DeleteMapping("/user/{id}")
  def deleteUser(@PathVariable id: String): ResponseEntity[String] = {
    try {
      userService.deleteUser(id)
      // VULNERABLE: No audit logging for sensitive operation
      ResponseEntity.ok("User deleted")
    } catch {
      case e: Exception =>
        // VULNERABLE: Log injection possible
        logger.error(s"Failed to delete user: ${e.getMessage}")
        ResponseEntity.internalServerError().body("Deletion failed")
    }
```

```
  }
}
```

Secure Logging Implementation

Comprehensive Security Logging

```scala
```

```
@RestController
class SecureUserController(
  userService: UserService,
  auditService: AuditService
) {
  private val logger = LoggerFactory.getLogger(getClass)
  private val securityLogger = LoggerFactory.getLogger("SECURITY")

  @PostMapping("/login")
  def login(@RequestParam username: String,
          @RequestParam password: String,
          request: HttpServletRequest): ResponseEntity[String] = {

    val clientIp = request.getRemoteAddr
    val userAgent = request.getHeader("User-Agent")
    val sessionId = request.getSession.getId

    // SECURE: Log authentication attempt without sensitive data
    securityLogger.info("Authentication attempt: username={}, clientIp={}, sessionId={}",
      sanitizeForLog(username), clientIp, sessionId)

    val user = userService.authenticate(username, password)
    user match {
      case Some(authenticatedUser) =>
        // SECURE: Audit successful authentication
        auditService.logSecurityEvent(
          eventType = "LOGIN_SUCCESS",
          userId = authenticatedUser.id,
          clientIp = clientIp,
          details = Map("sessionId" -> sessionId)
        )
        ResponseEntity.ok("Login successful")
```

```
    case None =>
     // SECURE: Detailed security logging for failed attempts
     securityLogger.warn("Authentication failed: username={}, clientIp={}, reason=INVALID_CREDENTIALS",
      sanitizeForLog(username), clientIp)
     ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Login failed")
   }
  }

  // SECURE: Log sanitization to prevent log injection
  private def sanitizeForLog(input: String): String = {
   input.replaceAll("[\\r\\n\\t]", "_").take(100)
  }
}

// SECURE: Audit service with structured logging
@Service
class AuditService {
  private val auditLogger = LoggerFactory.getLogger("AUDIT")

  def logSecurityEvent(eventType: String, userId: String,
                       clientIp: String = "", targetResource: String = "",
                       details: Map[String, String] = Map.empty): Unit = {

   val auditEvent = AuditEvent(
    eventType = eventType,
    userId = userId,
    clientIp = clientIp,
    targetResource = targetResource,
    timestamp = Instant.now(),
    details = details
   )
```

```
    auditLogger.info(auditEvent.toJson)
  }
}

case class AuditEvent(
  eventType: String,
  userId: String,
  clientIp: String,
  targetResource: String,
  timestamp: Instant,
  details: Map[String, String]
) {
  def toJson: String = {
    s"""{"eventType":"$eventType","userId":"$userId","clientIp":"$clientIp","targetResource":"$targetResource","
  }
}
```

A10: Server-Side Request Forgery (SSRF)

Ranking: #10 in OWASP Top 10 2021 (new entry) **Focus**: Applications fetching remote resources without validating user-supplied URLs

Vulnerable SSRF Patterns

Scala 2 with Spring Boot - No URL Validation

```scala
```

```
@RestController
class ImageProxyController {
  private val httpClient = HttpClient.newHttpClient()

  @GetMapping("/fetch-image")
  def fetchImage(@RequestParam url: String): ResponseEntity[Array[Byte]] = {
    try {
      // VULNERABLE: No URL validation
      val request = HttpRequest.newBuilder()
        .uri(URI.create(url))
        .build()

      val response = httpClient.send(request, HttpResponse.BodyHandlers.ofByteArray())
      ResponseEntity.ok(response.body())
    } catch {
      case e: Exception =>
        ResponseEntity.badRequest().build()
    }
  }
}
```

Scala 2/3 with Pekko HTTP - Arbitrary URL Fetching

```
scala
```

```
import org.apache.pekko.http.scaladsl.Http
import org.apache.pekko.http.scaladsl.model._

class ProxyRoutes {
 def routes: Route = {
  path("fetch") {
   get {
    parameter("url") { url =>
     // VULNERABLE: No URL validation
     val futureResponse = Http().singleRequest(HttpRequest(uri = url))

     onComplete(futureResponse) {
      case Success(response) =>
       complete("Data fetched successfully")
      case Failure(_) =>
       complete(StatusCodes.BadRequest, "Request failed")
     }
    }
   }
  }
 }
}
```

Secure SSRF Prevention

Comprehensive URL Validation

scala

```
@RestController
class SecureImageProxyController(urlValidator: UrlValidator) {

 private val httpClient = HttpClient.newBuilder()
  .connectTimeout(Duration.ofSeconds(10))
  .build()

 private val allowedHosts = Set(
  "images.example.com",
  "cdn.example.com",
  "api.trusted-partner.com"
 )

 @GetMapping("/fetch-image")
 def fetchImage(@RequestParam url: String): ResponseEntity[Array[Byte]] = {
  // SECURE: Validate URL before making request
  urlValidator.validateUrl(url) match {
   case Valid(validatedUrl) =>
    try {
     val request = HttpRequest.newBuilder()
      .uri(validatedUrl)
      .timeout(Duration.ofSeconds(30))
      .build()

     val response = httpClient.send(request, HttpResponse.BodyHandlers.ofByteArray())

     // SECURE: Validate response
     if (response.statusCode() == 200 && isValidImageResponse(response)) {
      ResponseEntity.ok(response.body())
     } else {
      ResponseEntity.badRequest().build()
     }
```

```
      } catch {
        case e: Exception =>
          logger.warn(s"Failed to fetch image: ${e.getMessage}")
          ResponseEntity.internalServerError().build()
      }

    case Invalid(reason) =>
      logger.warn(s"Invalid URL rejected: $url, reason: $reason")
      ResponseEntity.badRequest().body(s"Invalid URL: $reason".getBytes)
  }
 }

 private def isValidImageResponse(response: HttpResponse[Array[Byte]]): Boolean = {
  val contentType = response.headers().firstValue("content-type").orElse("")
  contentType.startsWith("image/") && response.body().length < 10 * 1024 * 1024
 }
}

// SECURE: URL validation service
@Service
class UrlValidator {
 private val allowedSchemes = Set("https")
 private val blockedHosts = Set(
  "localhost", "127.0.0.1", "0.0.0.0",
  "169.254.169.254", // AWS metadata
  "metadata.google.internal" // GCP metadata
 )

 def validateUrl(url: String): ValidationResult[URI] = {
  try {
   val uri = new URI(url)
```

```
    // Check scheme
    if (!allowedSchemes.contains(uri.getScheme)) {
      return Invalid(s"Scheme ${uri.getScheme} not allowed")
    }

    // Check for blocked hosts
    val host = uri.getHost
    if (host == null || blockedHosts.contains(host.toLowerCase)) {
      return Invalid(s"Host $host is blocked")
    }

    // Check for private IP ranges
    if (isPrivateIp(host)) {
      return Invalid(s"Private IP addresses not allowed: $host")
    }

    Valid(uri)
  } catch {
    case e: URISyntaxException =>
      Invalid(s"Invalid URI format: ${e.getMessage}")
  }
 }

 private def isPrivateIp(host: String): Boolean = {
  try {
    val addr = InetAddress.getByName(host)
    addr.isSiteLocalAddress || addr.isLoopbackAddress || addr.isLinkLocalAddress
  } catch {
    case _: Exception => false
  }
 }
}
```

```
sealed trait ValidationResult[+T]
case class Valid[T](value: T) extends ValidationResult[T]
case class Invalid(reason: String) extends ValidationResult[Nothing]
```

Scala-Specific Security Libraries and Ecosystem

Essential Security Libraries

Authentication and Authorization

- **Silhouette** 7.0.0: Comprehensive authentication library for Play Framework
- **play-pac4j** 10.0.2: Security library supporting OAuth, CAS, SAML, OpenID Connect
- **http4s-jwt-auth** 2.0.10: JWT authentication for Http4s applications
- **Guardian Pan-Domain Authentication** 10.0.0: Federated authentication across services

Cryptography

- **Scrypto** 3.0.0: Comprehensive cryptographic primitives with hash algorithms, encoding, elliptic curves
- **authentikat-jwt** 0.4.5: Simple JWT library for claims-based authentication

Security Testing

- **ScalaTest** 3.2.19: Primary testing framework with security-focused capabilities
- **Contrast Security Agent**: Runtime vulnerability detection for Play and Akka applications

Build Tool Security Configuration

SBT Security Plugins

```scala
// project/plugins.sbt
addSbtPlugin("net.vonbuchholtz" % "sbt-dependency-check" % "5.1.0")
addSbtPlugin("com.github.sbt" % "sbt-dependency-graph" % "0.13.4")
addSbtPlugin("com.github.sbt" % "sbt-updates" % "0.6.4")

// build.sbt security configuration
dependencyCheckAutoUpdate := Some(true)
dependencyCheckFormat := "All"
addCommandAlias("securityCheck", ";dependencyCheck;dependencyUpdates")
```

Static Analysis Tools

- **DerScanner**: Specialized SAST tool for Scala with 170+ security rules
- **Scalastyle**: Popular style checker with security-focused rules
- **Scapegoat**: Compiler plugin for static analysis
- **WartRemover**: Flexible linter focusing on bad practices
- **SonarQube**: Multi-language tool with hundreds of Scala security rules

Scala Language Security Features

Type Safety Benefits

- **Compile-time Error Prevention**: Option types eliminate null pointer exceptions
- **Pattern Matching**: Ensures exhaustive case coverage
- **Strong Type System**: Prevents type confusion attacks
- **Immutability**: Reduces state-related vulnerabilities

Scala 3 Security Improvements

- **Enhanced Type Safety**: Union and intersection types for precise security modeling
- **Improved Implicit System**: `given` and `using` clauses with clearer intent
- **Opaque Types**: Eliminate runtime overhead of security wrappers
- **Better Macro Security**: More secure than Scala 2's experimental macros

Security-Focused Code Patterns

Domain-Driven Security Types (Scala 3)

```scala
opaque type UserId = String
opaque type SecureToken = String

def authenticateUser(id: UserId, token: SecureToken): AuthResult = {
 // Type system ensures correct parameter usage
}
```

Safe Option Handling

```scala
def safeOperation(input: Option[String]): String = input match {
  case Some(value) => processValue(value)
  case None => "Default safe value"
}
```

Immutable Security State

```scala
```

```scala
case class UserSession(
  userId: UserId,
  token: SessionToken,
  expiresAt: Instant,
  permissions: Set[Permission]
) {
  // Safe methods that return new instances
  def withPermission(perm: Permission): UserSession =
    copy(permissions = permissions + perm)

  def isExpired: Boolean = Instant.now().isAfter(expiresAt)
}
```

Security Testing Approaches

Property-Based Security Testing

```scala
class SecurityPropertyTest extends AnyPropSpec with PropertyChecks {

  property("SQL injection prevention") {
    forAll { (userInput: String) =>
      val query = QueryBuilder.buildSafeQuery(userInput)
      query should not contain "DROP TABLE"
      query should not contain "DELETE FROM"
    }
  }
}
```

Integration Security Testing

```scala
class ActorSecurityTest extends TestKit(ActorSystem("SecurityTest"))
  with AnyFlatSpecLike {

  "Secure actor" should "reject unauthorized messages" in {
    val secureActor = system.actorOf(Props[SecureActor])
    secureActor ! UnauthorizedMessage("sensitive")
    expectMsg(AccessDenied)
  }
}
```

Vulnerability Management and CVEs

Critical Scala-Specific CVEs

- **CVE-2022-36944** (High Severity): Scala 2.13.x deserialization vulnerability affecting scala-library-2.13.8.jar and earlier
- **CVE-2017-15288** (Medium Severity): Scala compiler daemon weak permissions vulnerability
- **CVE-2020-26238**: Reflection-based remote code execution vulnerability

Recommended Secure Dependency Versions

```scala
```

```
libraryDependencies ++= Seq(
  // Spring Boot
  "org.springframework.boot" % "spring-boot-starter-web" % "3.1.5",
  "org.springframework.boot" % "spring-boot-starter-security" % "3.1.5",

  // Pekko HTTP
  "org.apache.pekko" %% "pekko-http" % "1.0.2",
  "org.apache.pekko" %% "pekko-stream" % "1.0.2",

  // Database
  "com.typesafe.slick" %% "slick" % "3.4.1",
  "com.typesafe.slick" %% "slick-hikaricp" % "3.4.1",

  // Security
  "org.springframework.security" % "spring-security-crypto" % "6.0.0",
  "org.scorexfoundation" %% "scrypto" % "3.0.0",

  // Scala versions
  "org.scala-lang" % "scala-library" % "2.13.14", // Scala 2
  "org.scala-lang" % "scala3-library_3" % "3.3.1"  // Scala 3
)
```

Key Security Recommendations

Development Best Practices

1. **Leverage Type Safety**: Use Scala's type system to prevent security issues at compile time
2. **Implement Defense in Depth**: Apply security controls at multiple layers
3. **Use Immutable Data Structures**: Reduce state-related vulnerabilities

4. **Apply Functional Programming Principles**: Prefer pure functions for security-critical operations
5. **Regular Dependency Updates**: Keep Scala and library versions current with security patches

Testing and Monitoring

1. **Security-Focused Unit Tests**: Include security validation in test suites
2. **Property-Based Testing**: Use ScalaCheck for comprehensive security property validation
3. **Static Analysis Integration**: Implement tools like DerScanner, Scalastyle, and SonarQube
4. **Comprehensive Audit Logging**: Log all security events with proper sanitization
5. **Automated Vulnerability Scanning**: Use sbt-dependency-check and Snyk integration

Framework-Specific Security

- **Spring Boot**: Integrate with Spring Security, use proper method-level security annotations
- **Pekko HTTP**: Leverage authentication/authorization directives, implement custom middleware
- **Play Framework**: Use Silhouette for comprehensive authentication, implement CSRF protection

The Scala ecosystem provides robust security features through its type system, functional programming paradigms, and mature security libraries. However, applications remain vulnerable to OWASP Top 10 threats without proper implementation of security controls. By following these comprehensive patterns and leveraging Scala's inherent security benefits, developers can build significantly more secure applications while maintaining the language's productivity advantages.