Contents

1 SEC-009: Sensitive Data Exposure in Code/Logs

1.1 Severity

High

1.2 OWASP Reference

A02:2021 - Cryptographic Failures

1.3 CWE References

- CWE-798: Use of Hard-Coded Credentials
- CWE-532: Insertion of Sensitive Information into Log File

1.4 Description

Sensitive Data Exposure vulnerabilities occur when an application inadvertently exposes confidential information through source code, log files, error messages, temporary files, or configuration files. In MCP servers, this includes exposure of API keys, database credentials, private encryption keys, personal information, or business-critical data. Attackers can extract this information through code repositories, log analysis, backup files, or error pages, leading to unauthorized access, data breaches, or system compromise.

1.5 Compliance Mapping

1.5.1 SOC2

- **CC6.1**: Logical and Physical Access Controls - The organization restricts access to sensitive data in code and logs.
- **CC6.2**: Access Control - Sensitive credentials are managed through secure mechanisms, not hardcoded.

- **CC7.2**: System Monitoring - The organization monitors logs and prevents exposure of sensitive data.

1.5.2 HIPAA

- **§164.312(a)(1)**: Access Control - Restrict access to ePHI in logs and source code.
- **§164.312(d)**: Encryption and Decryption - Encrypt sensitive data in transit and at rest in logs.
- **§164.312(b)**: Audit Controls - Implement mechanisms to record access to sensitive information.

1.5.3 PCI DSS

- **2.1**: Sensitive data must not be stored in source code or logs
- **6.5.10**: Log files and error messages must not contain payment card data or sensitive credentials
- **8.2.3**: Passwords must not be displayed or stored in logs
- **11.5.2**: Implement alerts for unauthorized attempts to access sensitive data

1.6 Vulnerable Code Example

```
// □ INSECURE: Hardcoded credentials and sensitive data in logs
const express = require('express');
const mysql = require('mysql2');
const app = express();

app.use(express.json());

// DANGER: Hardcoded database credentials in code
const db = mysql.createConnection({
  host: 'localhost',
  user: 'admin',
  password: 'MyP@ssw0rd123',
  database: 'mcp_production'
});

// DANGER: API key hardcoded in source code
const STRIPE_API_KEY = 'sk_live_51H8X5E2eZvKLc2p5nF9mK0L2p3q4r5s6t7u8v9w0x1y2z3a4b5

// Vulnerable endpoint: user login with sensitive logging
app.post('/api/login', (req, res) => {
  const { username, password, mfa_code } = req.body;

  // DANGER: Logging credentials in plain text
  console.log(`Login attempt: ${username} with password ${password}`);

  // Verify credentials
```

```
  db.query(
    'SELECT * FROM users WHERE username = ? AND password = ?',
    [username, password],
    (err, results) => {
      if (err) {
        // DANGER: Exposing database error details
        console.error('Database error:', err);
        return res.status(500).json({
          error: err.message,
          details: err.sqlState
        });
      }

      if (results.length > 0) {
        // DANGER: Logging MFA codes
        console.log(`MFA code for user ${username}: ${mfa_code}`);
        res.json({ success: true, user: results[0] });
      } else {
        res.status(401).json({ error: 'Invalid credentials' });
      }
    }
  );
});

// Vulnerable endpoint: payment processing
app.post('/api/payment', (req, res) => {
  const { amount, cardNumber, cvv, expiry } = req.body;

  // DANGER: Card details logged to console
  console.log(`Processing payment: ${cardNumber} ${expiry}`);

  // DANGER: Hardcoded Stripe key used in request
  // Never expose in frontend or logs
  const response = makeStripeRequest(amount, cardNumber, STRIPE_API_KEY);

  res.json(response);
});

// Vulnerable endpoint: user profile with PII
app.get('/api/user/:id', (req, res) => {
  const userId = req.params.id;

  db.query(
    'SELECT * FROM users WHERE id = ?',
    [userId],
    (err, results) => {
      if (err) {
        return res.status(500).json({ error: err.message });
```

```
      }

      // DANGER: Exposing all user data including SSN, phone, email
      res.json(results[0]);
    }
  );
});

app.listen(3000);
```

1.7 Secure Implementation

```
// ✅ SECURE: Credentials in environment variables, sensitive data protected
const express = require('express');
const mysql = require('mysql2/promise');
const winston = require('winston');
const app = express();

app.use(express.json());

// Load credentials from environment variables (never hardcode)
const db = mysql.createPool({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD, // From .env or secrets manager
  database: process.env.DB_NAME,
  waitForConnections: true,
  connectionLimit: 10
});

const STRIPE_API_KEY = process.env.STRIPE_API_KEY; // From secrets manager
const LOG_LEVEL = process.env.LOG_LEVEL || 'info';

// Configure logger to exclude sensitive data
const logger = winston.createLogger({
  level: LOG_LEVEL,
  format: winston.format.json(),
  transports: [
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' })
  ]
});

// Middleware to sanitize request body before logging
const sanitizeRequest = (req, res, next) => {
  req.sanitized = JSON.parse(JSON.stringify(req.body));
```

```
  // Remove sensitive fields from sanitized copy
  const sensitiveFields = ['password', 'mfa_code', 'cardNumber', 'cvv', 'expiry',
  sensitiveFields.forEach(field => {
    if (req.sanitized[field]) {
      req.sanitized[field] = '[REDACTED]';
    }
  });

  next();
};

app.use(sanitizeRequest);

// Secure endpoint: user login with secure logging
app.post('/api/login', async (req, res) => {
  const { username, password, mfa_code } = req.body;

  if (!username || !password) {
    return res.status(400).json({ error: 'Missing credentials' });
  }

  try {
    // Log only non-sensitive information
    logger.info(`Login attempt`, {
      username: username,
      timestamp: new Date().toISOString(),
      ip: req.ip
    });

    // Query database
    const [rows] = await db.execute(
      'SELECT id, username, password_hash, mfa_enabled FROM users WHERE username =
      [username]
    );

    if (rows.length === 0) {
      logger.warn(`Failed login: user not found`, {
        username: username,
        ip: req.ip
      });
      return res.status(401).json({ error: 'Invalid credentials' });
    }

    const user = rows[0];

    // Verify password (don't log it)
    const bcrypt = require('bcrypt');
    const isValidPassword = await bcrypt.compare(password, user.password_hash);
```

```
    if (!isValidPassword) {
      logger.warn(`Failed login: invalid password`, {
        username: username,
        ip: req.ip
      });
      return res.status(401).json({ error: 'Invalid credentials' });
    }

    // Verify MFA if enabled (don't log code)
    if (user.mfa_enabled && !verifyMFA(user.id, mfa_code)) {
      logger.warn(`Failed login: invalid MFA`, {
        username: username,
        ip: req.ip
      });
      return res.status(401).json({ error: 'Invalid MFA code' });
    }

    logger.info(`Successful login`, {
      username: username,
      userId: user.id
    });

    // Don't return password hash
    delete user.password_hash;
    res.json({ success: true, user });

  } catch (error) {
    // Log error without exposing details
    logger.error(`Login error`, {
      error: error.name,
      timestamp: new Date().toISOString()
    });
    res.status(500).json({ error: 'Authentication failed' });
  }
});

// Secure endpoint: payment processing without logging sensitive data
app.post('/api/payment', async (req, res) => {
  const { amount, token } = req.body; // Use tokenized card, not raw card data

  if (!amount || !token) {
    return res.status(400).json({ error: 'Missing payment information' });
  }

  try {
    // Never handle raw card data in application
    // Use payment processor tokens or hosted forms
```

```
    logger.info(`Payment processing initiated`, {
      amount: amount,
      timestamp: new Date().toISOString(),
      ip: req.ip
      // Note: token is not logged
    });

    // Make payment request (token sent to Stripe, not raw card data)
    const stripeResponse = await makeStripeRequest(amount, token, STRIPE_API_KEY);

    if (stripeResponse.success) {
      logger.info(`Payment successful`, {
        transactionId: stripeResponse.transaction_id,
        amount: amount
      });

      res.json({ success: true, transactionId: stripeResponse.transaction_id });
    } else {
      logger.warn(`Payment failed`, {
        reason: stripeResponse.reason,
        amount: amount
      });
      res.status(400).json({ error: 'Payment failed' });
    }

  } catch (error) {
    logger.error(`Payment processing error`, {
      error: error.name,
      timestamp: new Date().toISOString()
    });
    res.status(500).json({ error: 'Payment processing failed' });
  }
});

// Secure endpoint: user profile with PII protection
app.get('/api/user/:id', async (req, res) => {
  const userId = req.params.id;

  // Validate user is requesting their own data
  if (req.user.id !== parseInt(userId)) {
    logger.warn(`Unauthorized profile access attempt`, {
      requestingUser: req.user.id,
      targetUser: userId,
      ip: req.ip
    });
    return res.status(403).json({ error: 'Forbidden' });
  }
```

```
  try {
    const [rows] = await db.execute(
      'SELECT id, username, email, created_at FROM users WHERE id = ?',
      [userId]
    );

    if (rows.length === 0) {
      return res.status(404).json({ error: 'User not found' });
    }

    // Never expose SSN, phone, or other sensitive fields in API response
    logger.info(`Profile retrieved`, {
      userId: userId
    });

    res.json(rows[0]);

  } catch (error) {
    logger.error(`Profile retrieval error`, {
      error: error.name
    });
    res.status(500).json({ error: 'Failed to retrieve profile' });
  }
});

app.listen(3000);
```

1.8 Remediation Steps

1. **Store Sensitive Credentials in Environment Variables or Secrets Manager**: Never hardcode API keys, database credentials, or encryption keys in source code. Use environment variables (.env files in development, secrets managers in production like AWS Secrets Manager, HashiCorp Vault, or Azure Key Vault). Rotate credentials regularly. Implement automatic secret detection in code repositories using tools like GitGuardian or TruffleHog.

2. **Implement Comprehensive Log Sanitization**: Configure logging frameworks to exclude sensitive fields (passwords, tokens, SSN, credit cards, MFA codes). Implement middleware to sanitize request/response bodies before logging. Use structured logging with field-level filtering. Never log full error messages that might reveal system internals - log only error types and safe error IDs.

3. **Apply Data Classification and Minimization**: Classify data as sensitive (PII, credentials, payment data) and non-sensitive. Only collect and log the minimum necessary data. Don't expose sensitive fields in API responses - return only what the client needs. Implement field-level access controls in responses based on user roles.

4. **Implement Detection and Response Mechanisms**: Use SIEM (Security Information and Event Management) systems to detect suspicious access patterns. Implement alerts for repeated failed login attempts, unusual data access, or attempts to access sensitive endpoints. Conduct regular code reviews focusing on hardcoded secrets. Implement automated scanning of repositories for exposed credentials. Monitor log retention policies to ensure old logs containing sensitive data are properly deleted.

1.9 References

- OWASP Sensitive Data Exposure
- CWE-798: Use of Hard-Coded Credentials
- CWE-532: Insertion of Sensitive Information into Log File
- OWASP Logging Cheat Sheet
- Node.js Credential Security

Last Updated: January 2026
Status: Published
Language: English