Contents

1 SEC-012: Weak Cryptography

1.1 Severity

High □

1.2 OWASP Reference

A02:2021 - Cryptographic Failures

1.3 CWE References

- CWE-327: Use of a Broken or Risky Cryptographic Algorithm
- CWE-326: Inadequate Encryption Strength

1.4 Description

Weak Cryptography vulnerabilities occur when an application uses deprecated, broken, or insufficiently strong cryptographic algorithms, outdated protocols, or improperly configured encryption mechanisms. In MCP servers, this allows attackers to decrypt sensitive data, forge authentication tokens, compromise encrypted communications, or break encryption keys. Common weaknesses include MD5/SHA1 for hashing, DES/3DES for encryption, self-generated encryption schemes, hardcoded cryptographic keys, or improper key management.

1.5 Compliance Mapping

1.5.1 SOC2

- **CC6.2**: Access Control - Strong cryptography is used to protect sensitive data and authentication mechanisms.
- **CC6.3**: Logical and Physical Access Controls - Encryption keys are securely managed and protected.

- **CC6.4**: Authentication and Authorization - Cryptographic mechanisms enforce secure authentication.

1.5.2 HIPAA

- **§164.312(c)(1)**: Integrity Controls - Implement encryption for ePHI to ensure data integrity.
- **§164.312(d)**: Encryption and Decryption - Use NIST-approved or equivalent encryption algorithms.
- **§164.312(e)(1)**: Transmission Security - Implement encryption for ePHI in transit using secure protocols.

1.5.3 PCI DSS

- **3.2.1**: Strong cryptographic algorithms must be used for encryption (AES-256, RSA-2048+)
- **3.4**: Render PAN unreadable anywhere it is stored using strong cryptography
- **4.1**: Use TLS 1.2 or higher for transmitting cardholder data
- **6.5.10**: Weak cryptographic algorithms are prevented through secure coding

1.6 Vulnerable Code Example

```
// □ INSECURE: Weak cryptography and poor key management
const express = require('express');
const crypto = require('crypto');
const md5 = require('md5');
const app = express();

app.use(express.json());

// DANGER: Hardcoded encryption key
const ENCRYPTION_KEY = 'my-secret-key';

// DANGER: Using MD5 for password hashing (insecure)
app.post('/api/register', (req, res) => {
  const { username, password } = req.body;

  // DANGER: MD5 is broken for cryptographic purposes
  const passwordHash = md5(password);

  // Store passwordHash in database
  res.json({ success: true, userId: 123 });
});

// DANGER: Using DES for encryption (outdated, weak)
app.post('/api/encrypt-data', (req, res) => {
  const { data } = req.body;
  // DANGER: DES has only 56-bit key (can be brute-forced in hours)
  const cipher = crypto.createCipher('des', ENCRYPTION_KEY);
  let encrypted = cipher.update(data, 'utf8', 'hex');
  encrypted += cipher.final('hex');

  res.json({ encrypted });
});

// DANGER: Self-implemented encryption (amateur cryptography)
app.post('/api/secure-data', (req, res) => {
  const { data } = req.body;

  // DANGER: XOR cipher is trivially broken
  const xorEncrypt = (str, key) => {
    return str.split('').map(char =>
      String.fromCharCode(char.charCodeAt(0) ^ key.charCodeAt(0))
    ).join('');
  };

  const encrypted = xorEncrypt(data, ENCRYPTION_KEY);
  res.json({ encrypted });
});

// DANGER: Random number generation for security purposes
app.post('/api/generate-token', (req, res) => {
  // DANGER: Math.random() is not cryptographically secure
  const token = Math.random().toString(36).substring(2, 15);

  res.json({ token });
});

// DANGER: SHA1 for digital signatures (broken for collision resistance)
app.post('/api/sign-data', (req, res) => {
  const { data } = req.body;

  // DANGER: SHA1 has known collision attacks
  const signature = crypto.createHash('sha1').update(data).digest('hex');

  res.json({ signature });
});

app.listen(3000);
```

1.7 Secure Implementation

```
// □ SECURE: Strong cryptography with proper key management
const express = require('express');
const crypto = require('crypto');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const app = express();

app.use(express.json());

// Load encryption key from secure key management service
// Never hardcode keys!
const ENCRYPTION_KEY = Buffer.from(process.env.ENCRYPTION_KEY, 'base64');
if (ENCRYPTION_KEY.length !== 32) {
  throw new Error('ENCRYPTION_KEY must be 32 bytes (256 bits)');
}

const JWT_SECRET = process.env.JWT_SECRET;
const BCRYPT_ROUNDS = 12; // CPU-intensive to slow down brute-force attacks

// Helper function to encrypt data securely (AES-256-GCM)
function encryptData(plaintext) {
  // Generate random IV for each encryption (crucial for security)
  const iv = crypto.randomBytes(16);

  // Create cipher with AES-256-GCM
  const cipher = crypto.createCipheriv('aes-256-gcm', ENCRYPTION_KEY, iv);

  // Encrypt the data
  let encrypted = cipher.update(plaintext, 'utf8', 'hex');
  encrypted += cipher.final('hex');

  // Get authentication tag (prevents tampering)
  const authTag = cipher.getAuthTag();

  // Return IV + encrypted data + auth tag
  return {
    iv: iv.toString('hex'),
    data: encrypted,
    authTag: authTag.toString('hex')
  };
}

// Helper function to decrypt data securely
function decryptData(encrypted) {
  try {
    const decipher = crypto.createDecipheriv(
      'aes-256-gcm',
      ENCRYPTION_KEY,
      Buffer.from(encrypted.iv, 'hex')
    );

    // Set authentication tag for verification
    decipher.setAuthTag(Buffer.from(encrypted.authTag, 'hex'));

    // Decrypt
    let decrypted = decipher.update(encrypted.data, 'hex', 'utf8');
    decrypted += decipher.final('utf8');

    return decrypted;
  } catch (error) {
    throw new Error('Decryption failed - data may have been tampered with');
  }
}

// Secure endpoint: user registration with strong hashing
app.post('/api/register', async (req, res) => {
  const { username, password } = req.body;

  if (!password || password.length < 12) {
    return res.status(400).json({ error: 'Password must be at least 12 characters'
  }

  try {
    // Use bcrypt with 12 rounds (strong but performant)
    // bcrypt automatically handles salt generation
    const passwordHash = await bcrypt.hash(password, BCRYPT_ROUNDS);

    // Store passwordHash in database (never store plaintext passwords)
    // await db.query('INSERT INTO users VALUES (?, ?)', [username, passwordHash])

    res.json({ success: true, message: 'User registered' });

  } catch (error) {
    console.error('Registration error:', error);
    res.status(500).json({ error: 'Registration failed' });
  }
});

// Secure endpoint: data encryption with AES-256-GCM
app.post('/api/encrypt-data', (req, res) => {
  const { data } = req.body;

  if (!data || typeof data !== 'string' || data.length === 0) {
    return res.status(400).json({ error: 'Invalid data' });
  }

  try {
    // Encrypt with AES-256-GCM (authenticated encryption)
    const encrypted = encryptData(data);

    // Return encrypted object (store all parts together)
    res.json({
      success: true,
      encrypted: encrypted,
      algorithm: 'AES-256-GCM'
    });

  } catch (error) {
    console.error('Encryption error:', error);
    res.status(500).json({ error: 'Encryption failed' });
  }
});

// Secure endpoint: secure data decryption
app.post('/api/decrypt-data', (req, res) => {
  const { encrypted } = req.body;

  if (!encrypted || !encrypted.iv || !encrypted.data || !encrypted.authTag) {
    return res.status(400).json({ error: 'Invalid encrypted object' });
  }

  try {
    const decrypted = decryptData(encrypted);
    res.json({
      success: true,
      data: decrypted
    });

  } catch (error) {
    console.error('Decryption error:', error);
    res.status(400).json({ error: 'Decryption failed' });
  }
});

// Secure endpoint: JWT token generation
app.post('/api/generate-token', (req, res) => {
  const { userId } = req.body;

  if (!userId) {
    return res.status(400).json({ error: 'Missing userId' });
  }
  try {
    // Use JWT with HS256 (HMAC-SHA256) or RS256 (RSA signature)
    const token = jwt.sign(
      { userId: userId },
      JWT_SECRET,
      {
        algorithm: 'HS256',
        expiresIn: '1h' // Short expiration time
      }
    );

    res.json({
      success: true,
      token: token,
      expiresIn: 3600 // seconds
    });

  } catch (error) {
    console.error('Token generation error:', error);
    res.status(500).json({ error: 'Token generation failed' });
  }
});

// Secure endpoint: cryptographic signatures with SHA256
app.post('/api/sign-data', (req, res) => {
  const { data } = req.body;

  if (!data || typeof data !== 'string') {
    return res.status(400).json({ error: 'Invalid data' });
  }

  try {
    // Use HMAC-SHA256 (requires a secret key)
    const signature = crypto
      .createHmac('sha256', process.env.SIGNING_SECRET)
      .update(data)
      .digest('hex');

    res.json({
      success: true,
      signature: signature,
      algorithm: 'HMAC-SHA256'
    });

  } catch (error) {
    console.error('Signing error:', error);
    res.status(500).json({ error: 'Signing failed' });
  }
```

```
});

// Secure endpoint: generate cryptographically secure random tokens
app.post('/api/generate-secure-token', (req, res) => {
  try {
    // Generate 32 random bytes (256 bits) - cryptographically secure
    const token = crypto.randomBytes(32).toString('hex');

    res.json({
      success: true,
      token: token,
      length: 64 // 32 bytes = 64 hex characters
    });

  } catch (error) {
    console.error('Token generation error:', error);
    res.status(500).json({ error: 'Token generation failed' });
  }
});

// Secure endpoint: TLS/HTTPS enforcement
app.use((req, res, next) => {
  // Enforce HTTPS in production
  if (process.env.NODE_ENV === 'production' && req.header('x-forwarded-proto') !==
    return res.status(403).json({ error: 'HTTPS required' });
  }
  next();
});

app.listen(3000);
```

1.8 Remediation Steps

1. **Use Modern, NIST-Approved Cryptographic Algorithms**: Replace weak algorithms (MD5, SHA1, DES, 3DES, RC4) with strong alternatives (SHA256+ for hashing, AES-256 for encryption, RSA-2048+ for asymmetric encryption). Use authenticated encryption modes (AES-GCM) that provide both confidentiality and integrity. For password hashing, use bcrypt, scrypt, or Argon2 which are intentionally slow to resist brute-force attacks.

2. **Implement Secure Key Management**: Never hardcode cryptographic keys in source code. Use environment variables or dedicated key management services (AWS KMS, Azure Key Vault, HashiCorp Vault). Rotate keys regularly according to security policies. Store keys encrypted at rest. Use unique keys for different purposes (encryption, signing, authentication).

3. **Use Authenticated Encryption and Proper IVs**: Use encryption modes that provide authentication (AES-GCM) to prevent tampering. Generate cryp-

tographically random IVs/nonces for each encryption operation (never reuse). Use crypto.randomBytes() for generating cryptographic randomness, never Math.random(). Verify authentication tags before decrypting.

4. **Enforce Strong TLS/HTTPS and Protocol Versions**: Configure servers to use TLS 1.2 or higher (prefer 1.3). Disable legacy protocols (SSL 3.0, TLS 1.0, TLS 1.1). Use strong cipher suites with forward secrecy. Implement HSTS (HTTP Strict Transport Security) headers. Regularly update cryptographic libraries to patch vulnerabilities. Conduct cryptographic reviews and security audits regularly.

1.9 References

- OWASP Cryptographic Storage Cheat Sheet
- CWE-327: Use of a Broken or Risky Cryptographic Algorithm
- NIST SP 800-175B: Guideline for Using Cryptographic Standards
- OWASP Cryptographic Failures
- Node.js Crypto Documentation
- PortSwigger: Crypto Attacks

Last Updated: January 2026
Status: Published
Language: English