Contents

1 SEC-012: Criptografía Débil

1.1 Severidad

**Alta **

1.2 Referencia OWASP

A02:2021 - Fallos Criptográficos

1.3 Referencias CWE

- CWE-327: Uso de Algoritmo Criptográfico Roto o Arriesgado
- CWE-326: Fortaleza de Cifrado Inadecuada

1.4 Descripción

Las vulnerabilidades de Criptografía Débil ocurren cuando una aplicación usa algoritmos criptográficos deprecados, rotos o insuficientemente fuertes, protocolos desactualizados o mecanismos de cifrado configurados impropiamente. En servidores MCP, esto permite a los atacantes descifrar datos sensibles, falsificar tokens de autenticación, comprometer comunicaciones cifradas o romper claves de cifrado. Las debilidades comunes incluyen MD5/SHA1 para hashing, DES/3DES para cifrado, esquemas de cifrado autoimplementados, claves criptográficas codificadas o gestión de claves impropia.

1.5 Mapeo de Cumplimiento

1.5.1 SOC2

- **CC6.2**: Control de Acceso - Se utiliza criptografía fuerte para proteger datos sensibles y mecanismos de autenticación.
- **CC6.3**: Controles de Acceso Lógico y Físico - Las claves de cifrado se gestionan y protegen de forma segura.

- **CC6.4**: Autenticación y Autorización - Los mecanismos criptográficos aplican autenticación segura.

1.5.2 HIPAA

- **§164.312(c)(1)**: Controles de Integridad - Implementar cifrado para ePHI para asegurar integridad de datos.
- **§164.312(d)**: Cifrado y Descifrado - Usar algoritmos de cifrado aprobados por NIST o equivalentes.
- **§164.312(e)(1)**: Seguridad de Transmisión - Implementar cifrado para ePHI en tránsito usando protocolos seguros.

1.5.3 PCI DSS

- **3.2.1**: Se deben usar algoritmos criptográficos fuertes para cifrado (AES-256, RSA-2048+)
- **3.4**: Hacer ilegibles los PAN en cualquier lugar donde se almacenan usando criptografía fuerte
- **4.1**: Usar TLS 1.2 o superior para transmitir datos de titular de tarjeta
- **6.5.10**: Las vulnerabilidades de criptografía débil se previenen mediante codificación segura

1.6 Ejemplo de Código Vulnerable

```
// □ INSEGURO: Criptografía débil y gestión de claves deficiente
const express = require('express');
const crypto = require('crypto');
const md5 = require('md5');
const app = express();

app.use(express.json());

// PELIGRO: Clave de cifrado codificada
const ENCRYPTION_KEY = 'my-secret-key';

// PELIGRO: Usar MD5 para hashing de contraseña (inseguro)
app.post('/api/register', (req, res) => {
  const { username, password } = req.body;

  // PELIGRO: MD5 está roto para propósitos criptográficos
  const passwordHash = md5(password);

  // Almacenar passwordHash en base de datos
  res.json({ success: true, userId: 123 });
});

// PELIGRO: Usar DES para cifrado (desactualizad o, débil)
```

```
app.post('/api/encrypt-data', (req, res) => {
  const { data } = req.body;

  // PELIGRO: DES solo tiene clave de 56-bit (puede ser forzada en horas)
  const cipher = crypto.createCipher('des', ENCRYPTION_KEY);
  let encrypted = cipher.update(data, 'utf8', 'hex');
  encrypted += cipher.final('hex');

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

// PELIGRO: Cifrado autoimplementado (criptografía amateur)
app.post('/api/secure-data', (req, res) => {
  const { data } = req.body;

  // PELIGRO: El cifrado XOR es trivialmente roto
  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 });
});

// PELIGRO: Generación de números aleatorios con propósitos de seguridad
app.post('/api/generate-token', (req, res) => {
  // PELIGRO: Math.random() no es criptográficamente seguro
  const token = Math.random().toString(36).substring(2, 15);

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

// PELIGRO: SHA1 para firmas digitales (roto para resistencia de colisión)
app.post('/api/sign-data', (req, res) => {
  const { data } = req.body;

  // PELIGRO: SHA1 tiene ataques de colisión conocidos
  const signature = crypto.createHash('sha1').update(data).digest('hex');

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

app.listen(3000);
```

1.7 Implementación Segura

```
// ✅ SEGURO: Criptografía fuerte con gestión adecuada de claves
const express = require('express');
const crypto = require('crypto');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const app = express();

app.use(express.json());

// Cargar clave de cifrado desde servicio seguro de gestión de claves
// ¡Nunca codificar claves!
const ENCRYPTION_KEY = Buffer.from(process.env.ENCRYPTION_KEY, 'base64');
if (ENCRYPTION_KEY.length !== 32) {
  throw new Error('ENCRYPTION_KEY debe ser 32 bytes (256 bits)');
}

const JWT_SECRET = process.env.JWT_SECRET;
const BCRYPT_ROUNDS = 12; // Intensivo en CPU para ralentizar ataques de fuerza br

// Función auxiliar para cifrar datos de forma segura (AES-256-GCM)
function encryptData(plaintext) {
  // Generar IV aleatorio para cada cifrado (crucial para seguridad)
  const iv = crypto.randomBytes(16);

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

  // Cifrar los datos
  let encrypted = cipher.update(plaintext, 'utf8', 'hex');
  encrypted += cipher.final('hex');

  // Obtener etiqueta de autenticación (previene manipulación)
  const authTag = cipher.getAuthTag();

  // Devolver IV + datos cifrados + etiqueta de autenticación
  return {
    iv: iv.toString('hex'),
    data: encrypted,
    authTag: authTag.toString('hex')
  };
}

// Función auxiliar para descifrar datos de forma segura
function decryptData(encrypted) {
  try {
    const decipher = crypto.createDecipheriv(
```

```
      'aes-256-gcm',
      ENCRYPTION_KEY,
      Buffer.from(encrypted.iv, 'hex')
    );

    // Establecer etiqueta de autenticación para verificación
    decipher.setAuthTag(Buffer.from(encrypted.authTag, 'hex'));

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

    return decrypted;
  } catch (error) {
    throw new Error('Descifrado fallido - los datos pueden haber sido manipulados'
  }
}

// Endpoint seguro: registro de usuario con hashing fuerte
app.post('/api/register', async (req, res) => {
  const { username, password } = req.body;

  if (!password || password.length < 12) {
    return res.status(400).json({ error: 'La contraseña debe tener al menos 12 cara
  }

  try {
    // Usar bcrypt con 12 rondas (fuerte pero eficiente)
    // bcrypt maneja automáticamente la generación de salt
    const passwordHash = await bcrypt.hash(password, BCRYPT_ROUNDS);

    // Almacenar passwordHash en base de datos (nunca almacenar contraseñas en tex
    // await db.query('INSERT INTO users VALUES (?, ?)', [username, passwordHash])

    res.json({ success: true, message: 'Usuario registrado' });

  } catch (error) {
    console.error('Error de registro:', error);
    res.status(500).json({ error: 'Registro fallido' });
  }
});

// Endpoint seguro: cifrado de datos con 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: 'Datos inválidos' });
```

```
  }

  try {
    // Cifrar con AES-256-GCM (cifrado autenticado)
    const encrypted = encryptData(data);

    // Devolver objeto cifrado (almacenar todas las partes juntas)
    res.json({
      success: true,
      encrypted: encrypted,
      algorithm: 'AES-256-GCM'
    });

  } catch (error) {
    console.error('Error de cifrado:', error);
    res.status(500).json({ error: 'Cifrado fallido' });
  }
});

// Endpoint seguro: descifrado seguro de datos
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: 'Objeto cifrado inválido' });
  }

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

  } catch (error) {
    console.error('Error de descifrado:', error);
    res.status(400).json({ error: 'Descifrado fallido' });
  }
});

// Endpoint seguro: generación de token JWT
app.post('/api/generate-token', (req, res) => {
  const { userId } = req.body;

  if (!userId) {
    return res.status(400).json({ error: 'Falta userId' });
  }
```

```
  try {
    // Usar JWT con HS256 (HMAC-SHA256) o RS256 (firma RSA)
    const token = jwt.sign(
      { userId: userId },
      JWT_SECRET,
      {
        algorithm: 'HS256',
        expiresIn: '1h' // Tiempo de expiración corto
      }
    );

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

  } catch (error) {
    console.error('Error de generación de token:', error);
    res.status(500).json({ error: 'Generación de token fallida' });
  }
});

// Endpoint seguro: firmas criptográficas con SHA256
app.post('/api/sign-data', (req, res) => {
  const { data } = req.body;

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

  try {
    // Usar HMAC-SHA256 (requiere una clave secreta)
    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('Error de firma:', error);
    res.status(500).json({ error: 'Firma fallida' });
  }
```

```
});

// Endpoint seguro: generar tokens aleatorios criptográficamente seguros
app.post('/api/generate-secure-token', (req, res) => {
  try {
    // Generar 32 bytes aleatorios (256 bits) - criptográficamente seguro
    const token = crypto.randomBytes(32).toString('hex');

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

  } catch (error) {
    console.error('Error de generación de token:', error);
    res.status(500).json({ error: 'Generación de token fallida' });
  }
});

// Endpoint seguro: aplicación de TLS/HTTPS
app.use((req, res, next) => {
  // Aplicar HTTPS en producción
  if (process.env.NODE_ENV === 'production' && req.header('x-forwarded-proto') !==
    return res.status(403).json({ error: 'HTTPS requerido' });
  }
  next();
});

app.listen(3000);
```

1.8 Pasos de Remediación

1. **Usar Algoritmos Criptográficos Modernos Aprobados por NIST**: Reemplazar algoritmos débiles (MD5, SHA1, DES, 3DES, RC4) con alternativas fuertes (SHA256+ para hashing, AES-256 para cifrado, RSA-2048+ para cifrado asimétrico). Usar modos de cifrado autenticado (AES-GCM) que proporcionan confidencialidad e integridad. Para hashing de contraseña, usar bcrypt, scrypt o Argon2 que son intencionalmente lentos para resistir ataques de fuerza bruta.

2. **Implementar Gestión Segura de Claves**: Nunca codificar claves criptográficas en código fuente. Usar variables de entorno o servicios dedicados de gestión de claves (AWS KMS, Azure Key Vault, HashiCorp Vault). Rotar claves regularmente según políticas de seguridad. Almacenar claves cifradas en reposo. Usar claves únicas para diferentes propósitos (cifrado, firma, autenticación).

3. **Usar Cifrado Autenticado e IVs Apropiados**: Usar modos de cifrado que proporcionen autenticación (AES-GCM) para prevenir manipulación. Generar

IVs/nonces aleatorios criptográficamente para cada operación de cifrado (nunca reutilizar). Usar crypto.randomBytes() para generar aleatoriedad criptográfica, nunca Math.random(). Verificar etiquetas de autenticación antes de descifrar.

4. **Aplicar TLS/HTTPS Fuerte y Versiones de Protocolo**: Configurar servidores para usar TLS 1.2 o superior (preferir 1.3). Deshabilitar protocolos heredados (SSL 3.0, TLS 1.0, TLS 1.1). Usar suites de cifrado fuertes con forward secrecy. Implementar headers HSTS (HTTP Strict Transport Security). Actualizar regularmente librerías criptográficas para parchar vulnerabilidades. Realizar auditorías de seguridad y revisiones criptográficas regularmente.

1.9 Referencias

- OWASP Cryptographic Storage Cheat Sheet
- CWE-327: Uso de Algoritmo Criptográfico Roto o Arriesgado
- NIST SP 800-175B: Pauta para Usar Estándares Criptográficos
- OWASP Cryptographic Failures
- Node.js Crypto Documentation
- PortSwigger: Crypto Attacks

Última Actualización: Enero 2026
Estado: Publicado
Idioma: Español