Contents

1 SEC-006: Deserialización Insegura

1.1 Severidad

Crítica

1.2 Referencia OWASP

A08:2021 - Fallos de Integridad del Software y los Datos

1.3 Referencias CWE

- CWE-502: Deserialización de Datos No Confiables

1.4 Descripción

Las vulnerabilidades de deserialización insegura ocurren cuando una aplicación deserializa datos no confiables sin la validación adecuada, permitiendo a los atacantes manipular objetos serializados para lograr ejecución remota de código, escalada de privilegios o manipulación de datos. En servidores MCP que manejan formatos de datos serializados (JSON, XML, YAML, pickle, etc.), esta vulnerabilidad puede llevar al compromiso completo del sistema ya que los atacantes pueden crear payloads maliciosos que ejecutan código arbitrario durante el proceso de deserialización.

1.5 Mapeo de Cumplimiento

1.5.1 SOC2

- **CC6.1**: Controles de Acceso Lógico y Físico - La organización valida la integridad de los datos antes de procesarlos.
- **CC6.6**: Operaciones del Sistema - La organización implementa controles para prevenir la ejecución de código malicioso.
- **CC7.1**: Monitoreo del Sistema - La organización detecta y responde a eventos de seguridad relacionados con el procesamiento de datos.

- **CC8.1**: Gestión de Cambios - La organización evalúa las implicaciones de seguridad de los mecanismos de procesamiento de datos.

1.5.2 HIPAA

- **§164.312(c)(1)**: Controles de Integridad - Implementar políticas para proteger la ePHI de alteración o destrucción indebida.
- **§164.312(c)(2)**: Mecanismo para Autenticar ePHI - Implementar mecanismos electrónicos para corroborar que la ePHI no ha sido alterada o destruida.
- **§164.312(e)(1)**: Seguridad de Transmisión - Implementar medidas de seguridad técnica para proteger contra el acceso no autorizado a ePHI transmitida por redes.

1.5.3 PCI DSS

- **6.2.4**: El software personalizado y a medida se desarrolla de forma segura
- **6.5.8**: Las vulnerabilidades de deserialización insegura se previenen mediante codificación segura
- **8.3.2**: La integridad de los datos se mantiene mediante mecanismos de validación
- **11.3.1**: Las vulnerabilidades relacionadas con deserialización se identifican mediante pruebas de seguridad

1.6 Ejemplo de Código Vulnerable

```
// □ INSEGURO: Deserialización insegura de datos no confiables
const express = require('express');
const serialize = require('node-serialize');
const yaml = require('js-yaml');
const app = express();

app.use(express.text({ type: '*/*' }));

// Endpoint vulnerable: deserialización de datos de sesión de usuario
app.post('/api/restore-session', (req, res) => {
  const sessionData = req.body;

  // PELIGRO: Deserializar datos no confiables sin validación
  // El atacante puede inyectar objetos serializados maliciosos con ejecución de c
  const session = serialize.unserialize(sessionData);

  res.json({ message: 'Sesión restaurada', user: session.username });
});

// Endpoint vulnerable: parsing de YAML sin modo seguro
app.post('/api/config', (req, res) => {
  const configYaml = req.body;
```

```
  // PELIGRO: js-yaml.load() puede ejecutar código arbitrario
  // Entrada del atacante: "!!js/function 'return require(\"child_process\").exec(
  const config = yaml.load(configYaml);

  res.json({ message: 'Config actualizada', config });
});

// Endpoint vulnerable: eval() con entrada similar a JSON
app.post('/api/calculate', (req, res) => {
  const expression = req.body;

  // PELIGRO: Usar eval() en entrada de usuario
  // Entrada del atacante: "require('fs').readFileSync('/etc/passwd', 'utf8')"
  const result = eval(`(${expression})`);

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

// Endpoint vulnerable: constructor Function
app.post('/api/transform', (req, res) => {
  const { code, data } = req.body;

  // PELIGRO: Crear funciones desde entrada de usuario
  // El atacante puede ejecutar código arbitrario
  const transformFn = new Function('data', code);
  const transformed = transformFn(data);

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

app.listen(3000);
```

1.7 Implementación Segura

```
// ✅ SEGURO: Parsing seguro de datos con validación y operaciones restringidas
const express = require('express');
const yaml = require('js-yaml');
const crypto = require('crypto');
const app = express();

app.use(express.json({ limit: '1mb' })); // Usar parser JSON con límite de tamaño
app.use(express.text({ type: 'text/yaml', limit: '100kb' }));

// Clave secreta para validación de firma HMAC (almacenar de forma segura)
const HMAC_SECRET = process.env.HMAC_SECRET;

// Función auxiliar para crear token de sesión firmado
```

```
function createSignedSession(sessionData) {
  const jsonData = JSON.stringify(sessionData);
  const signature = crypto
    .createHmac('sha256', HMAC_SECRET)
    .update(jsonData)
    .digest('hex');

  return {
    data: jsonData,
    signature: signature
  };
}

// Función auxiliar para verificar token de sesión firmado
function verifySignedSession(data, signature) {
  const expectedSignature = crypto
    .createHmac('sha256', HMAC_SECRET)
    .update(data)
    .digest('hex');

  // Usar comparación segura contra timing attacks
  if (!crypto.timingSafeEqual(Buffer.from(signature), Buffer.from(expectedSignatur
    throw new Error('Firma inválida');
  }

  return JSON.parse(data);
}

// Endpoint seguro: restauración de sesión con verificación de firma
app.post('/api/restore-session', (req, res) => {
  const { data, signature } = req.body;

  if (!data || !signature) {
    return res.status(400).json({ error: 'Faltan datos de sesión o firma' });
  }

  try {
    // Verificar firma antes de parsear
    const session = verifySignedSession(data, signature);

    // Validar estructura de sesión
    if (!session.username || !session.userId || typeof session.userId !== 'number'
      return res.status(400).json({ error: 'Formato de sesión inválido' });
    }

    // Validación adicional: verificar expiración de sesión
    if (session.expiresAt < Date.now()) {
      return res.status(401).json({ error: 'Sesión expirada' });
```

```
    }

    res.json({
      message: 'Sesión restaurada',
      user: session.username,
      userId: session.userId
    });

  } catch (error) {
    console.error('Error de restauración de sesión:', error);
    res.status(401).json({ error: 'Sesión inválida' });
  }
});

// Endpoint seguro: parsing de YAML con modo seguro
app.post('/api/config', (req, res) => {
  const configYaml = req.body;

  // Validar tamaño de YAML
  if (configYaml.length > 10000) {
    return res.status(400).json({ error: 'Config demasiado grande' });
  }

  try {
    // Usar safeLoad para prevenir ejecución de código
    // Esto solo parsea tipos YAML básicos, no etiquetas personalizadas
    const config = yaml.load(configYaml, {
      schema: yaml.FAILSAFE_SCHEMA, // Esquema más restrictivo
      json: false // Deshabilitar características de compatibilidad JSON
    });

    // Validar estructura de config contra esquema esperado
    const validKeys = ['timeout', 'maxConnections', 'logLevel'];
    const configKeys = Object.keys(config || {});

    const hasInvalidKeys = configKeys.some(key => !validKeys.includes(key));
    if (hasInvalidKeys) {
      return res.status(400).json({ error: 'Claves de config inválidas' });
    }

    // Validación de tipos
    if (config.timeout && typeof config.timeout !== 'number') {
      return res.status(400).json({ error: 'Tipo de timeout inválido' });
    }

    if (config.maxConnections && (typeof config.maxConnections !== 'number' || con
      return res.status(400).json({ error: 'Valor de maxConnections inválido' });
    }
```

```
    res.json({ message: 'Config validada', config });

  } catch (error) {
    console.error('Error de parsing YAML:', error);
    res.status(400).json({ error: 'Formato YAML inválido' });
  }
});

// Endpoint seguro: cálculos matemáticos sin eval()
app.post('/api/calculate', (req, res) => {
  const { operation, values } = req.body;

  // Lista blanca de operaciones permitidas
  const allowedOperations = ['add', 'subtract', 'multiply', 'divide', 'average'];

  if (!allowedOperations.includes(operation)) {
    return res.status(400).json({ error: 'Operación inválida' });
  }

  // Validar array de valores
  if (!Array.isArray(values) || values.length === 0 || values.length > 100) {
    return res.status(400).json({ error: 'Array de valores inválido' });
  }

  // Validar que todos los valores son números
  if (!values.every(v => typeof v === 'number' && !isNaN(v))) {
    return res.status(400).json({ error: 'Todos los valores deben ser números' });
  }

  let result;

  // Implementación segura usando switch en lugar de eval
  switch (operation) {
    case 'add':
      result = values.reduce((sum, val) => sum + val, 0);
      break;
    case 'subtract':
      result = values.reduce((diff, val) => diff - val);
      break;
    case 'multiply':
      result = values.reduce((prod, val) => prod * val, 1);
      break;
    case 'divide':
      result = values.reduce((quot, val) => {
        if (val === 0) throw new Error('División por cero');
        return quot / val;
      });
```

```
      break;
    case 'average':
      result = values.reduce((sum, val) => sum + val, 0) / values.length;
      break;
  }

  res.json({ operation, result });
});

// Endpoint seguro: transformación de datos con operaciones en lista blanca
app.post('/api/transform', (req, res) => {
  const { operation, data } = req.body;

  // Lista blanca de operaciones de transformación seguras
  const allowedTransformations = {
    'uppercase': (str) => String(str).toUpperCase(),
    'lowercase': (str) => String(str).toLowerCase(),
    'trim': (str) => String(str).trim(),
    'reverse': (str) => String(str).split('').reverse().join(''),
    'wordcount': (str) => String(str).split(/\s+/).length
  };

  if (!allowedTransformations[operation]) {
    return res.status(400).json({ error: 'Operación de transformación inválida' })
  }

  if (typeof data !== 'string' || data.length > 10000) {
    return res.status(400).json({ error: 'Formato o tamaño de datos inválido' });
  }

  try {
    const transformed = allowedTransformations[operation](data);
    res.json({ operation, transformed });
  } catch (error) {
    console.error('Error de transformación:', error);
    res.status(500).json({ error: 'Transformación fallida' });
  }
});

app.listen(3000);
```

1.8 Pasos de Remediación

1. **Nunca Deserializar Datos No Confiables Sin Validación**: Evitar deserializar datos de fuentes no confiables siempre que sea posible. Si la deserialización es necesaria, usar métodos de parsing seguros (JSON.parse() para JSON, yaml.safeLoad() para YAML con FAILSAFE_SCHEMA). Implementar firmas

criptográficas (HMAC) para verificar la integridad de los datos antes de la deserialización. Nunca usar métodos inseguros como eval(), constructor Function o node-serialize en entrada de usuario.

2. **Implementar Validación Estricta de Entrada y Verificación de Tipos**: Definir esquemas explícitos para estructuras de datos esperadas usando librerías como Joi, Yup o validadores de JSON Schema. Validar tipos de datos, rangos y formatos antes de procesar. Rechazar cualquier dato que no se conforme al esquema esperado. Usar listas blancas para valores permitidos en lugar de listas negras.
3. **Usar Alternativas Seguras a Funciones Peligrosas**: Reemplazar eval() y el constructor Function con alternativas seguras (sentencias switch, tablas de búsqueda, operaciones en lista blanca). Para YAML, usar yaml.load() solo con FAILSAFE_SCHEMA o JSON_SCHEMA. Para archivos de configuración, preferir JSON sobre formatos que soportan ejecución de código. Usar entornos sandboxed (vm2, isolated-vm) si la ejecución dinámica de código es absolutamente necesaria.
4. **Implementar Defensa en Profundidad y Monitoreo**: Ejecutar procesos de aplicación con privilegios mínimos en entornos aislados (contenedores, VMs). Implementar Content Security Policy y sanitización de entrada. Monitorear patrones sospechosos de deserialización (payloads grandes, caracteres inusuales, intentos de ejecución). Usar verificaciones de integridad (firmas HMAC) para todos los datos serializados. Realizar auditorías de seguridad y pruebas de penetración regulares enfocadas en endpoints de deserialización.

1.9 Referencias

- OWASP Deserialization Cheat Sheet
- CWE-502: Deserialización de Datos No Confiables
- OWASP Top 10 2021 - A08 Fallos de Integridad del Software y los Datos
- Mejores Prácticas de Seguridad de Node.js
- Documentación de Carga Segura de YAML
- PortSwigger: Deserialización Insegura

Última Actualización: Enero 2026
Estado: Publicado
Idioma: Español