Contents

1 SEC-006: Insecure Deserialization

1.1 Severity

Critical

1.2 OWASP Reference

A08:2021 - Software and Data Integrity Failures

1.3 CWE References

- CWE-502: Deserialization of Untrusted Data

1.4 Description

Insecure deserialization vulnerabilities occur when an application deserializes untrusted data without proper validation, allowing attackers to manipulate serialized objects to achieve remote code execution, privilege escalation, or data tampering. In MCP servers that handle serialized data formats (JSON, XML, YAML, pickle, etc.), this vulnerability can lead to complete system compromise as attackers can craft malicious payloads that execute arbitrary code during the deserialization process.

1.5 Compliance Mapping

1.5.1 SOC2

- **CC6.1**: Logical and Physical Access Controls - The organization validates data integrity before processing.
- **CC6.6**: System Operations - The organization implements controls to prevent execution of malicious code.
- **CC7.1**: System Monitoring - The organization detects and responds to security events related to data processing.

- **CC8.1**: Change Management - The organization evaluates security implications of data processing mechanisms.

1.5.2 HIPAA

- **§164.312(c)(1)**: Integrity Controls - Implement policies to protect ePHI from improper alteration or destruction.
- **§164.312(c)(2)**: Mechanism to Authenticate ePHI - Implement electronic mechanisms to corroborate that ePHI has not been altered or destroyed.
- **§164.312(e)(1)**: Transmission Security - Implement technical security measures to guard against unauthorized access to ePHI transmitted over networks.

1.5.3 PCI DSS

- **6.2.4**: Bespoke and custom software is developed securely
- **6.5.8**: Insecure deserialization vulnerabilities are prevented through secure coding
- **8.3.2**: Data integrity is maintained through validation mechanisms
- **11.3.1**: Vulnerabilities related to deserialization are identified through security testing

1.6 Vulnerable Code Example

```
// □ INSECURE: Unsafe deserialization of untrusted data
const express = require('express');
const serialize = require('node-serialize');
const yaml = require('js-yaml');
const app = express();

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

// Vulnerable endpoint: deserializing user session data
app.post('/api/restore-session', (req, res) => {
  const sessionData = req.body;

  // DANGER: Deserializing untrusted data without validation
  // Attacker can inject malicious serialized objects with code execution
  const session = serialize.unserialize(sessionData);

  res.json({ message: 'Session restored', user: session.username });
});

// Vulnerable endpoint: YAML parsing without safe mode
app.post('/api/config', (req, res) => {
  const configYaml = req.body;

  // DANGER: js-yaml.load() can execute arbitrary code
```

```
  // Attacker input: "!!js/function 'return require(\"child_process\").exec(\"mali
  const config = yaml.load(configYaml);

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

// Vulnerable endpoint: eval() with JSON-like input
app.post('/api/calculate', (req, res) => {
  const expression = req.body;

  // DANGER: Using eval() on user input
  // Attacker input: "require('fs').readFileSync('/etc/passwd', 'utf8')"
  const result = eval(`(${expression})`);

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

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

  // DANGER: Creating functions from user input
  // Attacker can execute arbitrary code
  const transformFn = new Function('data', code);
  const transformed = transformFn(data);

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

app.listen(3000);
```

1.7 Secure Implementation

```
// ✅ SECURE: Safe data parsing with validation and restricted operations
const express = require('express');
const yaml = require('js-yaml');
const crypto = require('crypto');
const app = express();

app.use(express.json({ limit: '1mb' })); // Use JSON parser with size limit
app.use(express.text({ type: 'text/yaml', limit: '100kb' }));

// Secret key for HMAC signature validation (store securely)
const HMAC_SECRET = process.env.HMAC_SECRET;

// Helper function to create signed session token
function createSignedSession(sessionData) {
```

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

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

// Helper function to verify signed session token
function verifySignedSession(data, signature) {
  const expectedSignature = crypto
    .createHmac('sha256', HMAC_SECRET)
    .update(data)
    .digest('hex');

  // Use timing-safe comparison
  if (!crypto.timingSafeEqual(Buffer.from(signature), Buffer.from(expectedSignatur
    throw new Error('Invalid signature');
  }

  return JSON.parse(data);
}

// Secure endpoint: session restoration with signature verification
app.post('/api/restore-session', (req, res) => {
  const { data, signature } = req.body;

  if (!data || !signature) {
    return res.status(400).json({ error: 'Missing session data or signature' });
  }

  try {
    // Verify signature before parsing
    const session = verifySignedSession(data, signature);

    // Validate session structure
    if (!session.username || !session.userId || typeof session.userId !== 'number'
      return res.status(400).json({ error: 'Invalid session format' });
    }

    // Additional validation: check session expiration
    if (session.expiresAt < Date.now()) {
      return res.status(401).json({ error: 'Session expired' });
    }
```

```
    res.json({
      message: 'Session restored',
      user: session.username,
      userId: session.userId
    });

  } catch (error) {
    console.error('Session restoration error:', error);
    res.status(401).json({ error: 'Invalid session' });
  }
});

// Secure endpoint: YAML parsing with safe mode
app.post('/api/config', (req, res) => {
  const configYaml = req.body;

  // Validate YAML size
  if (configYaml.length > 10000) {
    return res.status(400).json({ error: 'Config too large' });
  }

  try {
    // Use safeLoad to prevent code execution
    // This only parses basic YAML types, not custom tags
    const config = yaml.load(configYaml, {
      schema: yaml.FAILSAFE_SCHEMA, // Most restrictive schema
      json: false // Disable JSON compatibility features
    });

    // Validate config structure against expected schema
    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: 'Invalid config keys' });
    }

    // Type validation
    if (config.timeout && typeof config.timeout !== 'number') {
      return res.status(400).json({ error: 'Invalid timeout type' });
    }

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

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

  } catch (error) {
    console.error('YAML parsing error:', error);
    res.status(400).json({ error: 'Invalid YAML format' });
  }
});

// Secure endpoint: mathematical calculations without eval()
app.post('/api/calculate', (req, res) => {
  const { operation, values } = req.body;

  // Whitelist allowed operations
  const allowedOperations = ['add', 'subtract', 'multiply', 'divide', 'average'];

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

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

  // Validate all values are numbers
  if (!values.every(v => typeof v === 'number' && !isNaN(v))) {
    return res.status(400).json({ error: 'All values must be numbers' });
  }

  let result;

  // Safe implementation using switch instead of 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('Division by zero');
        return quot / val;
      });
      break;
```

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

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

// Secure endpoint: data transformation with whitelisted operations
app.post('/api/transform', (req, res) => {
  const { operation, data } = req.body;

  // Whitelist of safe transformation operations
  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: 'Invalid transformation operation' });
  }

  if (typeof data !== 'string' || data.length > 10000) {
    return res.status(400).json({ error: 'Invalid data format or size' });
  }

  try {
    const transformed = allowedTransformations[operation](data);
    res.json({ operation, transformed });
  } catch (error) {
    console.error('Transformation error:', error);
    res.status(500).json({ error: 'Transformation failed' });
  }
});

app.listen(3000);
```

1.8 Remediation Steps

1. **Never Deserialize Untrusted Data Without Validation**: Avoid deserializing data from untrusted sources whenever possible. If deserialization is necessary, use safe parsing methods (JSON.parse() for JSON, yaml.safeLoad() for YAML with FAILSAFE_SCHEMA). Implement cryptographic signatures (HMAC) to verify data integrity before deserialization. Never use unsafe methods like eval(), Function

constructor, or node-serialize on user input.

2. **Implement Strict Input Validation and Type Checking**: Define explicit schemas for expected data structures using libraries like Joi, Yup, or JSON Schema validators. Validate data types, ranges, and formats before processing. Reject any data that doesn't conform to the expected schema. Use whitelists for allowed values rather than blacklists.

3. **Use Safe Alternatives to Dangerous Functions**: Replace eval() and Function constructor with safe alternatives (switch statements, lookup tables, whitelisted operations). For YAML, use yaml.load() with FAILSAFE_SCHEMA or JSON_SCHEMA only. For configuration files, prefer JSON over formats that support code execution. Use sandboxed environments (vm2, isolated-vm) if dynamic code execution is absolutely necessary.

4. **Implement Defense in Depth and Monitoring**: Run application processes with minimal privileges in isolated environments (containers, VMs). Implement Content Security Policy and input sanitization. Monitor for suspicious deserialization patterns (large payloads, unusual characters, execution attempts). Use integrity checks (HMAC signatures) for all serialized data. Conduct regular security audits and penetration testing focusing on deserialization endpoints.

1.9 References

- OWASP Deserialization Cheat Sheet
- CWE-502: Deserialization of Untrusted Data
- OWASP Top 10 2021 - A08 Software and Data Integrity Failures
- Node.js Security Best Practices
- YAML Safe Loading Documentation
- PortSwigger: Insecure Deserialization

Last Updated: January 2026
Status: Published
Language: English