Contents

1 SEC-005: XML External Entity (XXE)

1.1 Severity

High

1.2 OWASP Reference

A05:2021 - Security Misconfiguration

1.3 CWE References

- CWE-611: Improper Restriction of XML External Entity Reference

1.4 Description

XML External Entity (XXE) injection vulnerabilities occur when an application parses untrusted XML input without disabling external entity processing. This allows attackers to read arbitrary files from the server, perform Server-Side Request Forgery (SSRF) attacks, conduct denial-of-service attacks, or achieve remote code execution. In MCP servers that accept XML, SOAP, or SVG uploads/imports, XXE vulnerabilities can lead to complete system compromise and data exfiltration.

1.5 Compliance Mapping

1.5.1 SOC2

- **CC6.1**: Logical and Physical Access Controls - The organization implements access controls for file systems and sensitive data.
- **CC6.6**: System Operations - The organization prevents unauthorized execution of potentially malicious code.
- **CC7.2**: System Monitoring - The organization detects and monitors XML parsing attempts and security events.

1.5.2 HIPAA

- **§164.312(a)(1)**: Access Control - Implement technical policies to prevent unauthorized access to ePHI in files.
- **§164.312(c)(1)**: Integrity Controls - Protect ePHI from unauthorized alteration through secure XML processing.
- **§164.312(e)(1)**: Transmission Security - Implement controls to prevent ePHI disclosure through XXE exploitation.

1.5.3 PCI DSS

- **6.2.4**: All custom software must be developed securely with XML processing protections
- **6.5.1**: Injection flaws including XXE are addressed through secure coding practices
- **6.5.4**: Insecure direct object references via XXE are prevented
- **11.3.1**: XXE vulnerabilities are identified through regular vulnerability scanning

1.6 Vulnerable Code Example

```
// □ INSECURE: XML parsing without disabling external entities
const express = require('express');
const xml2js = require('xml2js');
const libxmljs = require('libxmljs');
const app = express();

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

// Vulnerable endpoint: XML data import using xml2js
app.post('/api/import-config', async (req, res) => {
  const xmlData = req.body;

  // DANGER: Default xml2js parser allows XXE attacks
  // Attacker input: <?xml version="1.0"?>
  // <!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
  // <config><user>&xxe;</user></config>
  const parser = new xml2js.Parser();

  try {
    const result = await parser.parseStringPromise(xmlData);
    res.json({ success: true, config: result });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// Vulnerable endpoint: SVG upload processing
app.post('/api/upload-svg', async (req, res) => {
```

```
  const svgData = req.body;

  // DANGER: libxmljs by default allows external entity processing
  try {
    const xmlDoc = libxmljs.parseXml(svgData);
    const root = xmlDoc.root();
    res.json({ success: true, svg: root.toString() });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

// Vulnerable endpoint: SOAP request handler
app.post('/api/soap-service', async (req, res) => {
  const soapRequest = req.body;

  // DANGER: No XXE protection for SOAP messages
  // Attacker can embed XXE payloads in SOAP requests
  try {
    const parser = new xml2js.Parser();
    const parsed = await parser.parseStringPromise(soapRequest);
    res.json({ response: 'SOAP processed' });
  } catch (error) {
    res.status(500).json({ error: 'Processing failed' });
  }
});

app.listen(3000);
```

1.7 Secure Implementation

```
// □ SECURE: XML parsing with XXE protection enabled
const express = require('express');
const xml2js = require('xml2js');
const libxmljs = require('libxmljs');
const app = express();

app.use(express.text({ type: 'application/xml', limit: '1mb' }));

// Secure endpoint: XML data import with XXE disabled
app.post('/api/import-config', async (req, res) => {
  const xmlData = req.body;

  // Validate XML size to prevent billion laughs attack
  if (xmlData.length > 1024 * 1024) { // 1MB max
    return res.status(400).json({ error: 'XML too large' });
  }
```

```
  try {
    // Create parser with XXE protections
    const parser = new xml2js.Parser({
      // Disable external entity processing - most important setting
      processEntities: false,

      // Disable DOCTYPE processing to prevent XXE
      doctype: false,

      // Prevent billion laughs DoS attack
      maxDepth: 20,

      // Limit expansion of entities
      maxAttributes: 50,

      // Disable namespace processing that could be exploited
      xmlns: false,

      // Trim values to prevent whitespace exploitation
      trim: true
    });

    const result = await parser.parseStringPromise(xmlData);

    // Additional validation of parsed structure
    if (!result.config) {
      return res.status(400).json({ error: 'Invalid XML structure' });
    }

    res.json({ success: true, config: result });

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

// Secure endpoint: SVG upload with XXE protection
app.post('/api/upload-svg', async (req, res) => {
  const svgData = req.body;

  // Validate size
  if (svgData.length > 5 * 1024 * 1024) { // 5MB max for SVG
    return res.status(400).json({ error: 'SVG too large' });
  }

  // Validate it starts with SVG tag
```

```
if (!svgData.includes('<svg')) {
  return res.status(400).json({ error: 'Not a valid SVG file' });
}

try {
  // Create parser with XXE protections disabled
  const parser = new libxmljs.SaxParser((cb) => {
    cb.onStartElement((elem, attrs) => {
      // Accept start element
    });
    cb.onError((msg) => {
      throw new Error(`XML Parse error: ${msg}`);
    });
  });

  // Enable XXE protection in libxmljs by disabling DTD processing
  const options = {
    dtdload: false,           // Don't load external DTDs
    dtdvalid: false,          // Don't validate against DTDs
    noent: false,             // Don't expand entities
    nonet: true,              // Don't access network
    nocdata: false            // OK to process CDATA sections
  };

  // Alternative: Use safer XML parsing with validation
  const xmlDoc = libxmljs.parseXml(svgData, {
    dtdload: false,
    dtdvalid: false,
    noent: false,
    nonet: true
  });

  // Validate root element is svg
  const root = xmlDoc.root();
  if (root.name() !== 'svg') {
    return res.status(400).json({ error: 'Root element must be <svg>' });
  }

  // Remove potentially dangerous elements
  const dangerousElements = ['script', 'iframe', 'embed', 'object'];
  dangerousElements.forEach(elem => {
    const elements = xmlDoc.find(`//${elem}`);
    elements.forEach(e => e.remove());
  });

  res.json({
    success: true,
    svg: root.toString().substring(0, 10000) // Limit response size
```

```
    });

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

// Secure endpoint: SOAP request handler with XXE protection
app.post('/api/soap-service', async (req, res) => {
  const soapRequest = req.body;

  // Validate size
  if (soapRequest.length > 2 * 1024 * 1024) { // 2MB max
    return res.status(400).json({ error: 'SOAP message too large' });
  }

  // Check for XXE patterns in raw XML (defense in depth)
  const xxePatterns = [
    /<!ENTITY\s+/i,
    /SYSTEM\s+/i,
    /PUBLIC\s+/i,
    /<!DOCTYPE\s+/i,
    /xsi:schemaLocation/i
  ];

  for (const pattern of xxePatterns) {
    if (pattern.test(soapRequest)) {
      return res.status(400).json({ error: 'Suspicious XML detected' });
    }
  }

  try {
    const parser = new xml2js.Parser({
      processEntities: false,
      doctype: false,
      maxDepth: 20,
      maxAttributes: 50,
      xmlns: false,
      trim: true,
      strict: true // Strict mode to reject malformed XML
    });

    const parsed = await parser.parseStringPromise(soapRequest);

    // Validate SOAP envelope structure
    if (!parsed['soap:Envelope'] && !parsed.Envelope) {
      return res.status(400).json({ error: 'Invalid SOAP message' });
```

```
    }

    res.json({
      success: true,
      response: 'SOAP processed securely',
      envelope: Object.keys(parsed)[0]
    });

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

// Secure endpoint: Generic XML validation endpoint
app.post('/api/validate-xml', async (req, res) => {
  const xmlData = req.body;

  // Size validation
  if (xmlData.length > 5 * 1024 * 1024) { // 5MB max
    return res.status(400).json({ error: 'XML too large' });
  }

  // Pattern-based XXE detection
  const xxeIndicators = [
    /<!ENTITY.*SYSTEM/is,
    /<!ENTITY.*PUBLIC/is,
    /<!DOCTYPE/i,
    /SYSTEM\s+["'](file|http|ftp):\/\//i
  ];

  for (const indicator of xxeIndicators) {
    if (indicator.test(xmlData)) {
      return res.status(400).json({
        error: 'XXE vulnerability detected',
        message: 'XML contains entity declarations or DOCTYPE - not allowed'
      });
    }
  }

  try {
    const parser = new xml2js.Parser({
      processEntities: false,
      doctype: false,
      maxDepth: 20,
      trim: true,
      strict: true
    });
```

```
    await parser.parseStringPromise(xmlData);
    res.json({ valid: true, message: 'XML is valid and safe' });

  } catch (error) {
    res.status(400).json({
      valid: false,
      error: error.message
    });
  }
});

app.listen(3000);
```

1.8 Remediation Steps

1. **Disable External Entity Processing in All XML Parsers**: Configure all XML parsing libraries to disable DTD processing, external entity expansion, and DOCTYPE parsing. For xml2js, set `processEntities: false` and `doctype: false`. For libxmljs, set `dtdload: false`, `dtdvalid: false`, and `noent: false`. For Node.js built-in parsers, use safe options or upgrade to versions with secure defaults.
2. **Implement XXE Detection and Prevention at Input Level**: Scan XML input for suspicious patterns (DOCTYPE declarations, ENTITY declarations, SYSTEM/PUBLIC keywords) before parsing. Reject XML that contains these indicators. Use strict XML parsing mode that rejects malformed input. Implement file upload restrictions (file type validation, size limits, format validation).
3. **Use Whitelist Validation and Safe XML Processing**: Define expected XML schemas using XSD validation libraries. Validate against schemas before processing. Use only necessary XML features - disable features like external DTDs, namespaces, or CDATA if not needed. Consider using safer alternatives to XML (JSON) when appropriate for the use case.
4. **Implement Defense in Depth and Monitoring**: Run XML processing in isolated containers or sandboxed environments. Implement rate limiting on XML parsing endpoints. Monitor for XXE exploitation attempts in logs (file access, network connections, error messages). Use Web Application Firewalls with XXE detection rules. Conduct regular security testing and code reviews of XML handling code.

1.9 References

- OWASP XXE Prevention Cheat Sheet
- CWE-611: Improper Restriction of XML External Entity Reference
- PortSwigger: XML External Entity Injection
- OWASP XML Security Cheat Sheet
- Node.js XML Parsing Security Guide

Last Updated: January 2026
Status: Published
Language: English