Contents

1 SEC-007: Path Traversal

1.1 Severity

High

1.2 OWASP Reference

A01:2021 - Broken Access Control

1.3 CWE References

- CWE-22: Improper Limitation of a Pathname to a Restricted Directory (‘Path Traversal’)
- CWE-23: Relative Path Traversal

1.4 Description

Path Traversal vulnerabilities occur when an application uses user-supplied input to construct file paths without proper validation and sanitization. This allows attackers to access files and directories outside the intended directory structure, potentially reading sensitive files (configuration files, private keys, source code), modifying or deleting files, or executing arbitrary code. In MCP servers that handle file operations, path traversal 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 restricts file system access to authorized resources only.
- **CC6.7**: Data Classification - Sensitive files are protected from unauthorized access through access controls.

- **CC7.2**: System Monitoring - The organization monitors file access patterns and detects suspicious activity.

1.5.2 HIPAA

- **§164.312(a)(1)**: Access Control - Implement technical policies to restrict access to ePHI files and directories.
- **§164.312(c)(1)**: Integrity Controls - Protect ePHI files from unauthorized access through secure file handling.
- **§164.312(c)(2)**: Mechanism to Authenticate ePHI - Verify authorized access to ePHI file locations.

1.5.3 PCI DSS

- **6.5.8**: Path traversal vulnerabilities are prevented through secure coding
- **2.2.4**: Configure system security parameters to prevent unauthorized file access
- **6.2.4**: All custom software is developed securely with proper file handling
- **11.3.1**: Path traversal vulnerabilities are identified through vulnerability scanning

1.6 Vulnerable Code Example

```
// □ INSECURE: Direct file path construction with user input
const express = require('express');
const fs = require('fs');
const path = require('path');
const app = express();

const UPLOAD_DIR = '/uploads';

// Vulnerable endpoint: file download with path traversal
app.get('/api/download/:filename', (req, res) => {
  const filename = req.params.filename;

  // DANGER: No path validation - attacker can use ../../../
  // Attacker input: "../../etc/passwd"
  const filepath = path.join(UPLOAD_DIR, filename);

  fs.readFile(filepath, (err, data) => {
    if (err) {
      return res.status(404).json({ error: 'File not found' });
    }
    res.download(data, filename);
  });
});

// Vulnerable endpoint: file preview
```

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

  // DANGER: Direct string concatenation
  // Attacker input: "../../../../etc/passwd"
  const filepath = UPLOAD_DIR + '/' + filename;

  fs.readFile(filepath, 'utf8', (err, data) => {
    if (err) {
      return res.status(404).json({ error: 'Not found' });
    }
    res.json({ preview: data.substring(0, 500) });
  });
});

// Vulnerable endpoint: file deletion
app.delete('/api/file/:id', (req, res) => {
  const { id } = req.params;

  // DANGER: No validation of file path
  // Attacker can delete arbitrary files
  const filepath = `/var/data/${id}.json`;

  fs.unlink(filepath, (err) => {
    if (err) {
      return res.status(500).json({ error: 'Delete failed' });
    }
    res.json({ success: true });
  });
});

app.listen(3000);
```

1.7 Secure Implementation

```
// ✅ SECURE: Strict file path validation and restrictions
const express = require('express');
const fs = require('fs');
const path = require('path');
const crypto = require('crypto');
const app = express();

const UPLOAD_DIR = path.resolve('/uploads');
const ALLOWED_EXTENSIONS = ['pdf', 'txt', 'jpg', 'png', 'json'];

// Helper function to safely resolve file path
function resolveSafePath(baseDir, userInput) {
```

```
  // Validate input format
  if (!userInput || typeof userInput !== 'string' || userInput.length > 255) {
    throw new Error('Invalid filename');
  }

  // Reject dangerous patterns
  if (userInput.includes('..') || userInput.includes('\\') || userInput.startsWith
    throw new Error('Invalid filename format');
  }

  // Get the base filename only (remove any path components)
  const basename = path.basename(userInput);

  // Verify the resolved path is within the base directory
  const resolvedPath = path.resolve(baseDir, basename);
  const resolvedBase = path.resolve(baseDir);

  if (!resolvedPath.startsWith(resolvedBase)) {
    throw new Error('Path traversal attempt detected');
  }

  return resolvedPath;
}

// Helper function to validate file extension
function validateExtension(filename) {
  const ext = path.extname(filename).substring(1).toLowerCase();
  if (!ALLOWED_EXTENSIONS.includes(ext)) {
    throw new Error(`File type .${ext} not allowed`);
  }
  return ext;
}

// Secure endpoint: file download with path validation
app.get('/api/download/:filename', (req, res) => {
  try {
    const filename = req.params.filename;

    // Validate extension
    validateExtension(filename);

    // Safely resolve path
    const filepath = resolveSafePath(UPLOAD_DIR, filename);

    // Additional check: verify file exists and is readable
    fs.accessSync(filepath, fs.constants.R_OK);

    // Check file size to prevent large file reads
```

```
    const stats = fs.statSync(filepath);
    if (stats.size > 100 * 1024 * 1024) { // 100MB max
      return res.status(400).json({ error: 'File too large' });
    }

    // Verify it's a regular file, not a directory or symlink
    if (!stats.isFile()) {
      return res.status(403).json({ error: 'Access denied' });
    }

    res.download(filepath, filename);

  } catch (error) {
    console.error('Download error:', error);
    res.status(400).json({ error: 'Cannot access file' });
  }
});

// Secure endpoint: file preview with validation
app.post('/api/preview-file', (req, res) => {
  try {
    const { filename } = req.body;

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

    // Validate extension
    validateExtension(filename);

    // Safely resolve path
    const filepath = resolveSafePath(UPLOAD_DIR, filename);

    // Check file accessibility
    fs.accessSync(filepath, fs.constants.R_OK);

    // Check file size
    const stats = fs.statSync(filepath);
    if (stats.size > 10 * 1024 * 1024) { // 10MB max for preview
      return res.status(400).json({ error: 'File too large for preview' });
    }

    // Verify it's a regular file
    if (!stats.isFile()) {
      return res.status(403).json({ error: 'Access denied' });
    }

    // Read with size limit
```

```
    const data = fs.readFileSync(filepath, 'utf8');
    const preview = data.substring(0, 1000);

    res.json({
      success: true,
      preview,
      size: stats.size
    });

  } catch (error) {
    console.error('Preview error:', error);
    res.status(400).json({ error: 'Cannot access file' });
  }
});

// Secure endpoint: file deletion with whitelist
app.delete('/api/file/:id', (req, res) => {
  try {
    const { id } = req.params;

    // Validate ID format (UUID or alphanumeric)
    if (!/^[a-zA-Z0-9-]{1,40}$/.test(id)) {
      return res.status(400).json({ error: 'Invalid file ID' });
    }

    // Construct filename with validated ID
    const filename = `${id}.json`;

    // Safely resolve path
    const filepath = resolveSafePath(UPLOAD_DIR, filename);

    // Check file exists before deleting
    if (!fs.existsSync(filepath)) {
      return res.status(404).json({ error: 'File not found' });
    }

    // Verify it's a regular file
    const stats = fs.statSync(filepath);
    if (!stats.isFile()) {
      return res.status(403).json({ error: 'Cannot delete this item' });
    }

    // Delete the file
    fs.unlinkSync(filepath);

    res.json({ success: true, message: 'File deleted' });

  } catch (error) {
```

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

// Secure endpoint: list files in directory
app.get('/api/files', (req, res) => {
  try {
    // Only list files in the upload directory
    const files = fs.readdirSync(UPLOAD_DIR);

    // Filter and validate each file
    const safeFiles = files.filter(file => {
      try {
        // Only include allowed file types
        const ext = path.extname(file).substring(1).toLowerCase();
        if (!ALLOWED_EXTENSIONS.includes(ext)) {
          return false;
        }

        // Verify it's a regular file
        const filepath = path.join(UPLOAD_DIR, file);
        const stats = fs.statSync(filepath);
        return stats.isFile();
      } catch (error) {
        return false;
      }
    }).map(file => {
      const filepath = path.join(UPLOAD_DIR, file);
      const stats = fs.statSync(filepath);
      return {
        name: file,
        size: stats.size,
        modified: stats.mtime
      };
    });

    res.json({ files: safeFiles });

  } catch (error) {
    console.error('List error:', error);
    res.status(500).json({ error: 'Cannot list files' });
  }
});

app.listen(3000);
```

1.8 Remediation Steps

1. **Use Path Resolution with Strict Validation**: Use `path.resolve()` to normalize paths and verify the resolved path is within the intended directory. Reject any input containing `..`, `/`, or `\`. Use `path.basename()` to extract only the filename. Validate against a strict whitelist of allowed filenames or patterns. Implement length limits (255 characters for most file systems).
2. **Implement File Type and Extension Whitelisting**: Only allow specific file extensions (.pdf, .txt, .json, etc.). Validate extensions on both upload and download operations. Verify MIME types match file extensions. Store files with a hash-based name (e.g., SHA256) instead of user-supplied names to completely eliminate path traversal risk.
3. **Use File ID Mapping and Immutable References**: Store files with unique identifiers (UUID, hash) instead of filenames. Maintain a database mapping of file IDs to actual file paths. This prevents any direct path manipulation and provides better access control. Generate file IDs server-side, never accept them from users.
4. **Implement Defense in Depth**: Verify files exist and are regular files (not directories or symlinks) before access. Check file sizes to prevent reading unexpectedly large files. Use principle of least privilege - run the application with minimal file system permissions. Implement file access logging for audit trails. Use operating system features (chroot, containerization) to restrict file system scope.

1.9 References

- OWASP Path Traversal
- CWE-22: Path Traversal
- OWASP Path Traversal Prevention Cheat Sheet
- PortSwigger: Path Traversal
- Node.js File System Security

Last Updated: January 2026
Status: Published
Language: English