Contents

1 SEC-002: Command Injection

1.1 Severity

Critical

1.2 OWASP Reference

A03:2021 - Injection

1.3 CWE References

- CWE-78: OS Command Injection
- CWE-77: Improper Neutralization of Special Elements used in a Command
- CWE-88: Argument Injection or Modification

1.4 Description

Command injection vulnerabilities occur when an application executes operating system commands constructed with untrusted user input without proper sanitization. In MCP servers, this allows attackers to execute arbitrary system commands with the privileges of the server process, potentially leading to full system compromise, data exfiltration, or lateral movement within the network. This is particularly dangerous in MCP implementations that provide file system access, code execution capabilities, or system integration features.

1.5 Compliance Mapping

1.5.1 SOC2

- **CC6.6**: Logical and Physical Access Controls - The organization implements controls to prevent unauthorized software from executing.
- **CC6.8**: System Operations - The organization restricts access to system configurations and system parameters to authorized personnel.

- **CC7.2**: System Monitoring - The organization monitors system components and validates system integrity.

1.5.2 HIPAA

- **§164.312(a)(1)**: Access Control - Technical policies and procedures must restrict access to electronic protected health information.
- **§164.312(b)**: Audit Controls - Implement hardware, software, and procedural mechanisms to record and examine activity.
- **§164.312(c)(1)**: Integrity Controls - Implement policies and procedures to protect ePHI from improper alteration or destruction.

1.5.3 PCI DSS

- **6.2.4**: Bespoke and custom software are developed securely, per PCI DSS requirements
- **6.5.1**: Injection flaws, particularly OS command injection, are prevented through secure coding practices
- **11.3.1**: External and internal vulnerabilities are identified via vulnerability scanning and penetration testing
- **11.6.1**: A change-detection mechanism is deployed to alert personnel to unauthorized modifications

1.6 Vulnerable Code Example

```
// □ INSECURE: Direct command execution with user input
const express = require('express');
const { exec } = require('child_process');
const app = express();

app.use(express.json());

// Vulnerable endpoint: file conversion tool
app.post('/api/convert-file', (req, res) => {
  const { filename, format } = req.body;

  // DANGER: User input directly concatenated into shell command
  // An attacker could input: filename = "file.pdf; rm -rf / #"
  const command = `convert ${filename} output.${format}`;

  exec(command, (error, stdout, stderr) => {
    if (error) {
      return res.status(500).json({ error: stderr });
    }
    res.json({ success: true, output: stdout });
  });
});
```

```
// Vulnerable endpoint: ping utility
app.get('/api/ping', (req, res) => {
  const { host } = req.query;

  // DANGER: No validation of host parameter
  // Attacker input: host = "google.com && cat /etc/passwd"
  exec(`ping -c 4 ${host}`, (error, stdout) => {
    res.send(stdout);
  });
});

app.listen(3000);
```

1.7 Secure Implementation

```
// ✅ SECURE: Safe command execution with input validation and parameterization
const express = require('express');
const { spawn } = require('child_process');
const path = require('path');
const app = express();

app.use(express.json());

// Whitelist of allowed file formats
const ALLOWED_FORMATS = ['png', 'jpg', 'pdf', 'webp'];
const ALLOWED_HOST_PATTERN = /^[a-zA-Z0-9.-]+$/;

// Helper function to validate filename
function sanitizeFilename(filename) {
  // Remove path traversal attempts and special characters
  const basename = path.basename(filename);
  // Only allow alphanumeric, dots, hyphens, underscores
  if (!/^[a-zA-Z0-9._-]+$/.test(basename)) {
    throw new Error('Invalid filename format');
  }
  return basename;
}

// Secure endpoint: file conversion with spawn (not exec)
app.post('/api/convert-file', (req, res) => {
  const { filename, format } = req.body;

  try {
    // Validate format against whitelist
    if (!ALLOWED_FORMATS.includes(format)) {
      return res.status(400).json({ error: 'Invalid format' });
```

```
    }

    // Sanitize filename
    const safeFilename = sanitizeFilename(filename);
    const outputFile = `output.${format}`;

    // Use spawn instead of exec - arguments are passed separately
    // This prevents shell injection because no shell is invoked
    const convertProcess = spawn('convert', [safeFilename, outputFile], {
      shell: false, // Critical: prevent shell interpretation
      timeout: 10000 // Prevent long-running processes
    });

    let stdout = '';
    let stderr = '';

    convertProcess.stdout.on('data', (data) => {
      stdout += data.toString();
    });

    convertProcess.stderr.on('data', (data) => {
      stderr += data.toString();
    });

    convertProcess.on('close', (code) => {
      if (code !== 0) {
        return res.status(500).json({ error: 'Conversion failed', details: stderr
      }
      res.json({ success: true, output: stdout, file: outputFile });
    });

    convertProcess.on('error', (error) => {
      res.status(500).json({ error: 'Process execution failed' });
    });

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

// Secure endpoint: ping utility with strict validation
app.get('/api/ping', (req, res) => {
  const { host } = req.query;

  // Validate host format (no special characters)
  if (!host || !ALLOWED_HOST_PATTERN.test(host)) {
    return res.status(400).json({ error: 'Invalid host format' });
  }
```

```
  // Additional length check
  if (host.length > 255) {
    return res.status(400).json({ error: 'Host name too long' });
  }

  // Use spawn with explicit arguments - no shell interpretation
  const pingProcess = spawn('ping', ['-c', '4', host], {
    shell: false,
    timeout: 10000
  });

  let output = '';

  pingProcess.stdout.on('data', (data) => {
    output += data.toString();
  });

  pingProcess.on('close', (code) => {
    res.json({ success: code === 0, output });
  });

  pingProcess.on('error', () => {
    res.status(500).json({ error: 'Ping execution failed' });
  });
});

app.listen(3000);
```

1.8 Remediation Steps

1. **Use Parameterized APIs Instead of Shell Commands**: Replace `child_process.exec()` with `child_process.spawn()` or `child_process.execFile()` and pass arguments as an array. Always set `shell: false` to prevent shell interpretation. This ensures arguments are passed directly to the executable without shell processing.
2. **Implement Strict Input Validation**: Create whitelists for allowed values (file formats, commands, parameters). Use regular expressions to validate input format and reject any input containing shell metacharacters (for example: `;`, `|`, `&`, `$`, `>`, `<`, `\`, and sequences like `\n`). Validate both the format and semantic meaning of input.
3. **Apply the Principle of Least Privilege**: Run the MCP server process with minimal system permissions. Use operating system features like chroot jails, containers, or sandboxing to limit the impact of successful command injection. Never run server processes as root or administrator.
4. **Implement Defense in Depth**: Use additional security layers including com-

mand execution logging, runtime application self-protection (RASP), system call filtering (seccomp on Linux), and monitoring for suspicious process creation patterns. Deploy intrusion detection systems to identify exploitation attempts.

1.9 References

- OWASP Command Injection
- CWE-78: OS Command Injection
- Node.js Child Process Documentation
- OWASP Command Injection Defense Cheat Sheet
- NIST SP 800-53 SI-10: Information Input Validation

Last Updated: January 2026
Status: Published
Language: English