Contents

1 SEC-002: Inyección de Comandos **1**
1.1 Severidad 1
1.2 Referencia OWASP 1
1.3 Referencias CWE 1
1.4 Descripción 1
1.5 Mapeo de Cumplimiento 1
1.6 Ejemplo de Código Vulnerable 2
1.7 Implementación Segura 3
1.8 Pasos de Remediación 5
1.9 Referencias 6

1 SEC-002: Inyección de Comandos

1.1 Severidad

Crítica

1.2 Referencia OWASP

A03:2021 - Inyección

1.3 Referencias CWE

- CWE-78: Inyección de Comandos del Sistema Operativo
- CWE-77: Neutralización Impropia de Elementos Especiales usados en un Comando
- CWE-88: Inyección o Modificación de Argumentos

1.4 Descripción

Las vulnerabilidades de inyección de comandos ocurren cuando una aplicación ejecuta comandos del sistema operativo construidos con entradas de usuario no confiables sin la sanitización adecuada. En servidores MCP, esto permite a los atacantes ejecutar comandos arbitrarios del sistema con los privilegios del proceso del servidor, potencialmente llevando a un compromiso completo del sistema, exfiltración de datos o movimiento lateral dentro de la red. Esto es particularmente peligroso en implementaciones MCP que proporcionan acceso al sistema de archivos, capacidades de ejecución de código o características de integración del sistema.

1.5 Mapeo de Cumplimiento

1.5.1 SOC2

- **CC6.6**: Controles de Acceso Lógico y Físico - La organización implementa controles para prevenir la ejecución de software no autorizado.

- **CC6.8**: Operaciones del Sistema - La organización restringe el acceso a configuraciones del sistema y parámetros del sistema al personal autorizado.
- **CC7.2**: Monitoreo del Sistema - La organización monitorea componentes del sistema y valida la integridad del sistema.

1.5.2 HIPAA

- **§164.312(a)(1)**: Control de Acceso - Las políticas y procedimientos técnicos deben restringir el acceso a la información de salud protegida electrónica.
- **§164.312(b)**: Controles de Auditoría - Implementar mecanismos de hardware, software y procedimientos para registrar y examinar la actividad.
- **§164.312(c)(1)**: Controles de Integridad - Implementar políticas y procedimientos para proteger la ePHI de alteración o destrucción indebida.

1.5.3 PCI DSS

- **6.2.4**: El software personalizado y a medida se desarrolla de forma segura, según los requisitos de PCI DSS
- **6.5.1**: Las fallas de inyección, particularmente la inyección de comandos del SO, se previenen mediante prácticas de codificación segura
- **11.3.1**: Las vulnerabilidades externas e internas se identifican mediante escaneo de vulnerabilidades y pruebas de penetración
- **11.6.1**: Se implementa un mecanismo de detección de cambios para alertar al personal sobre modificaciones no autorizadas

1.6 Ejemplo de Código Vulnerable

```
// □ INSEGURO: Ejecución directa de comandos con entrada de usuario
const express = require('express');
const { exec } = require('child_process');
const app = express();

app.use(express.json());

// Endpoint vulnerable: herramienta de conversión de archivos
app.post('/api/convert-file', (req, res) => {
  const { filename, format } = req.body;

  // PELIGRO: La entrada del usuario se concatena directamente en el comando shell
  // Un atacante podría enviar: 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 });
```

```
  });
});

// Endpoint vulnerable: utilidad de ping
app.get('/api/ping', (req, res) => {
  const { host } = req.query;

  // PELIGRO: Sin validación del parámetro host
  // Entrada del atacante: host = "google.com && cat /etc/passwd"
  exec(`ping -c 4 ${host}`, (error, stdout) => {
    res.send(stdout);
  });
});

app.listen(3000);
```

1.7 Implementación Segura

```
// ✅ SEGURO: Ejecución segura de comandos con validación de entrada y parametrizac
const express = require('express');
const { spawn } = require('child_process');
const path = require('path');
const app = express();

app.use(express.json());

// Lista blanca de formatos de archivo permitidos
const ALLOWED_FORMATS = ['png', 'jpg', 'pdf', 'webp'];
const ALLOWED_HOST_PATTERN = /^[a-zA-Z0-9.-]+$/;

// Función auxiliar para validar nombres de archivo
function sanitizeFilename(filename) {
  // Eliminar intentos de path traversal y caracteres especiales
  const basename = path.basename(filename);
  // Solo permitir alfanuméricos, puntos, guiones y guiones bajos
  if (!/^[a-zA-Z0-9._-]+$/.test(basename)) {
    throw new Error('Formato de nombre de archivo inválido');
  }
  return basename;
}

// Endpoint seguro: conversión de archivos con spawn (no exec)
app.post('/api/convert-file', (req, res) => {
  const { filename, format } = req.body;

  try {
    // Validar formato contra lista blanca
```

```
    if (!ALLOWED_FORMATS.includes(format)) {
      return res.status(400).json({ error: 'Formato inválido' });
    }

    // Sanitizar nombre de archivo
    const safeFilename = sanitizeFilename(filename);
    const outputFile = `output.${format}`;

    // Usar spawn en lugar de exec - los argumentos se pasan por separado
    // Esto previene la inyección shell porque no se invoca ningún shell
    const convertProcess = spawn('convert', [safeFilename, outputFile], {
      shell: false, // Crítico: prevenir interpretación shell
      timeout: 10000 // Prevenir procesos de larga duración
    });

    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: 'Conversión fallida', details: stderr
      }
      res.json({ success: true, output: stdout, file: outputFile });
    });

    convertProcess.on('error', (error) => {
      res.status(500).json({ error: 'Falló la ejecución del proceso' });
    });

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

// Endpoint seguro: utilidad de ping con validación estricta
app.get('/api/ping', (req, res) => {
  const { host } = req.query;

  // Validar formato de host (sin caracteres especiales)
  if (!host || !ALLOWED_HOST_PATTERN.test(host)) {
```

```
    return res.status(400).json({ error: 'Formato de host inválido' });
  }

  // Verificación adicional de longitud
  if (host.length > 255) {
    return res.status(400).json({ error: 'Nombre de host demasiado largo' });
  }

  // Usar spawn con argumentos explícitos - sin interpretación shell
  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: 'Ejecución de ping fallida' });
  });
});

app.listen(3000);
```

1.8 Pasos de Remediación

1. **Usar APIs Parametrizadas en Lugar de Comandos Shell**: Reemplazar `child_process.exec()` con `child_process.spawn()` o `child_process.execFile()` y pasar argumentos como un array. Siempre establecer `shell: false` para prevenir interpretación shell. Esto asegura que los argumentos se pasen directamente al ejecutable sin procesamiento shell.
2. **Implementar Validación Estricta de Entradas**: Crear listas blancas para valores permitidos (formatos de archivo, comandos, parámetros). Usar expresiones regulares para validar el formato de entrada y rechazar cualquier entrada que contenga metacaracteres shell (por ejemplo: `;`, `|`, `&`, `$`, `>`, `<`, `\` y secuencias como `\n`). Validar tanto el formato como el significado semántico de la entrada.
3. **Aplicar el Principio de Mínimo Privilegio**: Ejecutar el proceso del servidor MCP con permisos mínimos del sistema. Usar características del sistema operativo como jaulas chroot, contenedores o sandboxing para limitar el impacto de

una inyección de comandos exitosa. Nunca ejecutar procesos del servidor como root o administrador.

4. **Implementar Defensa en Profundidad**: Usar capas de seguridad adicionales incluyendo registro de ejecución de comandos, autoprotección de aplicaciones en tiempo de ejecución (RASP), filtrado de llamadas al sistema (seccomp en Linux) y monitoreo de patrones sospechosos de creación de procesos. Implementar sistemas de detección de intrusiones para identificar intentos de explotación.

1.9 Referencias

- OWASP Command Injection
- CWE-78: Inyección de Comandos del SO
- Documentación de Child Process de Node.js
- OWASP Command Injection Defense Cheat Sheet
- NIST SP 800-53 SI-10: Validación de Entrada de Información

Última Actualización: Enero 2026
Estado: Publicado
Idioma: Español