# CWE Detail – CWE-77

## Description

The product constructs all or part of a command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended command when it is sent to a downstream component.

## Extended Description

Many protocols and products have their own custom command language. While OS or shell command strings are frequently discovered and targeted, developers may not realize that these other command languages might also be vulnerable to attacks.

## Threat-Mapped Scoring

Score: 0.0

Priority: Unclassified

## Observed Examples (CVEs)

**•** CVE-2022-1509: injection of sed script syntax ("sed injection")

**•** CVE-2024-5184: API service using a large generative AI model allows direct prompt injection to leak hard-coded system prompts or execute other prompts.

**•** CVE-2020-11698: anti-spam product allows injection of SNMP commands into confiuration file

**•** CVE-2019-12921: image program allows injection of commands in "Magick Vector Graphics (MVG)" language.

**•** CVE-2022-36069: Python-based dependency management tool avoids OS command injection when generating Git commands but allows injection of optional arguments with input beginning with a dash (CWE-88), potentially allowing for code execution.

**•** CVE-1999-0067: Canonical example of OS command injection. CGI program does not neutralize "|" metacharacter when invoking a phonebook program.

**•** CVE-2020-9054: Chain: improper input validation (CWE-20) in username parameter, leading to OS command injection (CWE-78), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2021-41282: injection of sed script syntax ("sed injection")

**•** CVE-2019-13398: injection of sed script syntax ("sed injection")

## Related Attack Patterns (CAPEC)

* CAPEC-136
* CAPEC-15
* CAPEC-183
* CAPEC-248
* CAPEC-40
* CAPEC-43
* CAPEC-75
* CAPEC-76

## Modes of Introduction

**•** Implementation: Command injection vulnerabilities typically occur when: Data enters the application from an untrusted source. The data is part of a string that is executed as a command by the application.

**•** Implementation: REALIZATION: This weakness is caused during implementation of an architectural security tactic.

## Common Consequences

**•** Impact: Execute Unauthorized Code or Commands — Notes: If a malicious user injects a character (such as a semi-colon) that delimits the end of one command and the beginning of another, it may be possible to then insert an entirely new and unrelated command that was not intended to be executed. This gives an attacker a privilege or capability that they would not otherwise have.

## Potential Mitigations

**•** Architecture and Design: If at all possible, use library calls rather than external processes to recreate the desired functionality. (Effectiveness: N/A)

**•** Implementation: If possible, ensure that all external commands called from the program are statically created. (Effectiveness: N/A)

**•** Implementation: Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if the input is only expected to contain colors such as "red" or "blue." Do not rely exclusively on looking for malicious or malformed inputs. This is likely to miss at least one undesirable input, especially if the code's environment changes. This can give attackers enough room to bypass the intended validation. However, denylists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. (Effectiveness: N/A)

**•** Operation: Run time: Run time policy enforcement may be used in an allowlist fashion to prevent use of any non-sanctioned commands. (Effectiveness: N/A)

**•** System Configuration: Assign permissions that prevent the user from accessing/opening privileged files. (Effectiveness: N/A)

## Applicable Platforms

**•** None (Class: Not Language-Specific, Prevalence: Undetermined)

## Demonstrative Examples

**•** To avoid XSS risks, the code ensures that the response from the chatbot is properly encoded for HTML output. If the user provides CWE-77 and CWE-78, then the resulting prompt would look like:

**•** However, validate\_name() allows  
 filenames that begin with a "-". An adversary could  
 supply a filename like "-aR", producing the "ls -l -aR"  
 command (CWE-88), thereby getting a full recursive  
 listing of the entire directory and all of its  
 sub-directories. There are a couple possible mitigations for this  
 weakness. One would be to refactor the code to avoid  
 using system() altogether, instead relying on internal  
 functions. Another option could be to add a "--" argument  
 to the ls command, such as "ls -l --", so that any  
 remaining arguments are treated as filenames, causing  
 any leading "-" to be treated as part of a filename  
 instead of another option. Another fix might be to change the regular expression used in validate\_name to force the first character of the filename to be a letter or number, such as:

**•** Because the program runs with root privileges, the call to system() also executes with root privileges. If a user specifies a standard filename, the call works as expected. However, if an attacker passes a string of the form ";rm -rf /", then the call to system() fails to execute cat due to a lack of arguments and then plows on to recursively delete the contents of the root partition, leading to OS command injection (CWE-78).

**•** The problem here is that the program does not do any validation on the backuptype parameter read from the user. Typically the Runtime.exec() function will not execute multiple commands, but in this case the program first runs the cmd.exe shell in order to run multiple commands with a single call to Runtime.exec(). Once the shell is invoked, it will happily execute multiple commands separated by two ampersands. If an attacker passes a string of the form "& del c:\\dbms\\\*.\*", then the application will execute this command along with the others specified by the program. Because of the nature of the application, it runs with the privileges necessary to interact with the database, which means whatever command the attacker injects will run with those privileges as well.

## Notes

**•** Terminology: The "command injection" phrase carries different meanings, either as an attack or as a technical impact. The most common usage of "command injection" refers to the more-accurate OS command injection (CWE-78), but there are many command languages. In vulnerability-focused analysis, the phrase may refer to any situation in which the adversary can execute commands of their own choosing, i.e., the focus is on the risk and/or technical impact of exploitation. Many proof-of-concept exploits focus on the ability to execute commands and may emphasize "command injection." However, there are dozens of weaknesses that can allow execution of commands. That is, the ability to execute commands could be resultant from another weakness. To some, "command injection" can include cases in which the functionality intentionally allows the user to specify an entire command, which is then executed. In this case, the root cause weakness might be related to missing or incorrect authorization, since an adversary should not be able to specify arbitrary commands, but some users or admins are allowed. CWE-77 and its descendants are specifically focused on behaviors in which the product is intentionally building a command to execute, and the adversary can inject separators into the command or otherwise change the command being executed.

**•** Other: Command injection is a common problem with wrapper programs.