# CWE Detail – CWE-95

## Description

The product receives input from an upstream component, but it does not neutralize or incorrectly neutralizes code syntax before using the input in a dynamic evaluation call (e.g. "eval").

## Extended Description

N/A

## Threat-Mapped Scoring

Score: 0.0

Priority: Unclassified

## Observed Examples (CVEs)

**•** CVE-2024-4181: Framework for LLM applications allows eval injection via a crafted response from a hosting provider.

**•** CVE-2022-2054: Python compiler uses eval() to execute malicious strings as Python code.

**•** CVE-2021-22204: Chain: regex in EXIF processor code does not correctly determine where a string ends (CWE-625), enabling eval injection (CWE-95), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2021-22205: Chain: backslash followed by a newline can bypass a validation step (CWE-20), leading to eval injection (CWE-95), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2008-5071: Eval injection in PHP program.

**•** CVE-2002-1750: Eval injection in Perl program.

**•** CVE-2008-5305: Eval injection in Perl program using an ID that should only contain hyphens and numbers.

**•** CVE-2002-1752: Direct code injection into Perl eval function.

**•** CVE-2002-1753: Eval injection in Perl program.

**•** CVE-2005-1527: Direct code injection into Perl eval function.

**•** CVE-2005-2837: Direct code injection into Perl eval function.

**•** CVE-2005-1921: MFV. code injection into PHP eval statement using nested constructs that should not be nested.

**•** CVE-2005-2498: MFV. code injection into PHP eval statement using nested constructs that should not be nested.

**•** CVE-2005-3302: Code injection into Python eval statement from a field in a formatted file.

**•** CVE-2007-1253: Eval injection in Python program.

**•** CVE-2001-1471: chain: Resultant eval injection. An invalid value prevents initialization of variables, which can be modified by attacker and later injected into PHP eval statement.

**•** CVE-2007-2713: Chain: Execution after redirect triggers eval injection.

## Related Attack Patterns (CAPEC)

* CAPEC-35

## Attack TTPs

**•** T1027.009: Embedded Payloads (Tactics: defense-evasion)

**•** T1564.009: Resource Forking (Tactics: defense-evasion)

**•** T1027.006: HTML Smuggling (Tactics: defense-evasion)

## Modes of Introduction

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

**•** Implementation: This weakness is prevalent in handler/dispatch procedures that might want to invoke a large number of functions, or set a large number of variables.

## Common Consequences

**•** Impact: Read Files or Directories, Read Application Data — Notes: The injected code could access restricted data / files.

**•** Impact: Bypass Protection Mechanism — Notes: In some cases, injectable code controls authentication; this may lead to a remote vulnerability.

**•** Impact: Gain Privileges or Assume Identity — Notes: Injected code can access resources that the attacker is directly prevented from accessing.

**•** Impact: Execute Unauthorized Code or Commands — Notes: Code injection attacks can lead to loss of data integrity in nearly all cases as the control-plane data injected is always incidental to data recall or writing. Additionally, code injection can often result in the execution of arbitrary code or at least modify what code can be executed.

**•** Impact: Hide Activities — Notes: Often the actions performed by injected control code are unlogged.

## Potential Mitigations

**•** Architecture and Design: If possible, refactor your code so that it does not need to use eval() at all. (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)

**•** Implementation: Inputs should be decoded and canonicalized to the application's current internal representation before being validated (CWE-180, CWE-181). Make sure that your application does not inadvertently decode the same input twice (CWE-174). Such errors could be used to bypass allowlist schemes by introducing dangerous inputs after they have been checked. Use libraries such as the OWASP ESAPI Canonicalization control. Consider performing repeated canonicalization until your input does not change any more. This will avoid double-decoding and similar scenarios, but it might inadvertently modify inputs that are allowed to contain properly-encoded dangerous content. (Effectiveness: N/A)

**•** Implementation: For Python programs, it is frequently encouraged to use the ast.literal\_eval() function instead of eval, since it is intentionally designed to avoid executing code. However, an adversary could still cause excessive memory or stack consumption via deeply nested structures [REF-1372], so the python documentation discourages use of ast.literal\_eval() on untrusted data [REF-1373]. (Effectiveness: Discouraged Common Practice)

## Applicable Platforms

**•** Java (Class: None, Prevalence: Undetermined)

**•** JavaScript (Class: None, Prevalence: Undetermined)

**•** Python (Class: None, Prevalence: Undetermined)

**•** Perl (Class: None, Prevalence: Undetermined)

**•** PHP (Class: None, Prevalence: Undetermined)

**•** Ruby (Class: None, Prevalence: Undetermined)

**•** None (Class: Interpreted, Prevalence: Undetermined)

## Demonstrative Examples

**•** The script intends to take the 'action' parameter and invoke one of a variety of functions based on the value of that parameter - config\_file\_add\_key(), config\_file\_set\_key(), or config\_file\_delete\_key(). It could set up a conditional to invoke each function separately, but eval() is a powerful way of doing the same thing in fewer lines of code, especially when a large number of functions or variables are involved. Unfortunately, in this case, the attacker can provide other values in the action parameter, such as:

**•** The eval() function can take the user-supplied list and convert it into a Python list object, therefore allowing the programmer to use list comprehension methods to work with the data. However, if code is supplied to the eval() function, it will execute that code. For example, a malicious user could supply the following string:

## Notes

**•** Other: Factors: special character errors can play a role in increasing the variety of code that can be injected, although some vulnerabilities do not require special characters at all, e.g. when a single function without arguments can be referenced and a terminator character is not necessary.