# CWE Detail – CWE-94

## Description

The product constructs all or part of a code segment using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the syntax or behavior of the intended code segment.

## Extended Description

N/A

## Threat-Mapped Scoring

Score: 0.0

Priority: Unclassified

## Observed Examples (CVEs)

**•** CVE-2023-29374: Math component in an LLM framework translates user input into a Python
 expression that is input into the Python exec() method, allowing code
 execution - one variant of a "prompt injection" attack.

**•** CVE-2024-5565: Python-based library uses an LLM prompt containing user input to
 dynamically generate code that is then fed as input into the Python
 exec() method, allowing code execution - one variant of a "prompt
 injection" attack.

**•** 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-2020-8218: "Code injection" in VPN product, 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-2002-0495: Perl code directly injected into CGI library file from parameters to another CGI program.

**•** CVE-2005-1876: Direct PHP code injection into supporting template file.

**•** CVE-2005-1894: Direct code injection into PHP script that can be accessed by attacker.

**•** CVE-2003-0395: PHP code from User-Agent HTTP header directly inserted into log file implemented as PHP script.

## Related Attack Patterns (CAPEC)

* CAPEC-242
* CAPEC-35
* CAPEC-77

## 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.

## Common Consequences

**•** 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: When a product allows a user's input to contain code syntax, it might be possible for an attacker to craft the code in such a way that it will alter the intended control flow of the product. As a result, code injection can often result in the execution of arbitrary code. Code injection attacks can also lead to loss of data integrity in nearly all cases, since the control-plane data injected is always incidental to data recall or writing.

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

## Potential Mitigations

**•** Architecture and Design: Refactor your program so that you do not have to dynamically generate code. (Effectiveness: N/A)

**•** Architecture and Design: Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which code can be executed by your product. Examples include the Unix chroot jail and AppArmor. In general, managed code may provide some protection. This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise. Be careful to avoid CWE-243 and other weaknesses related to jails. (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. To reduce the likelihood of code injection, use stringent allowlists that limit which constructs are allowed. If you are dynamically constructing code that invokes a function, then verifying that the input is alphanumeric might be insufficient. An attacker might still be able to reference a dangerous function that you did not intend to allow, such as system(), exec(), or exit(). (Effectiveness: N/A)

**•** Testing: Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible. (Effectiveness: N/A)

**•** Testing: Use dynamic tools and techniques that interact with the product using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The product's operation may slow down, but it should not become unstable, crash, or generate incorrect results. (Effectiveness: N/A)

**•** Operation: Run the code in an environment that performs automatic taint propagation and prevents any command execution that uses tainted variables, such as Perl's "-T" switch. This will force the program to perform validation steps that remove the taint, although you must be careful to correctly validate your inputs so that you do not accidentally mark dangerous inputs as untainted (see CWE-183 and CWE-184). (Effectiveness: N/A)

**•** Operation: Run the code in an environment that performs automatic taint propagation and prevents any command execution that uses tainted variables, such as Perl's "-T" switch. This will force the program to perform validation steps that remove the taint, although you must be careful to correctly validate your inputs so that you do not accidentally mark dangerous inputs as untainted (see CWE-183 and CWE-184). (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

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

## Demonstrative Examples

**•** While the programmer intends for the MessageFile to only include data, an attacker can provide a message such as:

**•** 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

**•** Theoretical: Injection problems encompass a wide variety of issues -- all mitigated in very different ways. For this reason, the most effective way to discuss these weaknesses is to note the distinct features that classify them as injection weaknesses. The most important issue to note is that all injection problems share one thing in common -- i.e., they allow for the injection of control plane data into the user-controlled data plane. This means that the execution of the process may be altered by sending code in through legitimate data channels, using no other mechanism. While buffer overflows, and many other flaws, involve the use of some further issue to gain execution, injection problems need only for the data to be parsed. The most classic instantiations of this category of weakness are SQL injection and format string vulnerabilities.