# CWE Detail – CWE-502

## Description

The product deserializes untrusted data without sufficiently ensuring that the resulting data will be valid.

## Extended Description

N/A

## Threat-Mapped Scoring

Score: 0.0

Priority: Unclassified

## Observed Examples (CVEs)

**•** CVE-2019-12799: chain: bypass of untrusted deserialization issue (CWE-502) by using an assumed-trusted class (CWE-183)

**•** CVE-2015-8103: Deserialization issue in commonly-used Java library allows remote execution.

**•** CVE-2015-4852: Deserialization issue in commonly-used Java library allows remote execution. (KEV)

**•** CVE-2013-1465: Use of PHP unserialize function on untrusted input allows attacker to modify application configuration.

**•** CVE-2012-3527: Use of PHP unserialize function on untrusted input in content management system might allow code execution.

**•** CVE-2012-0911: Use of PHP unserialize function on untrusted input in content management system allows code execution using a crafted cookie value.

**•** CVE-2012-0911: Content management system written in PHP allows unserialize of arbitrary objects, possibly allowing code execution.

**•** CVE-2011-2520: Python script allows local users to execute code via pickled data.

**•** CVE-2012-4406: Unsafe deserialization using pickle in a Python script.

**•** CVE-2003-0791: Web browser allows execution of native methods via a crafted string to a JavaScript function that deserializes the string.

## Related Attack Patterns (CAPEC)

* CAPEC-586

## Modes of Introduction

**•** Architecture and Design: OMISSION: This weakness is caused by missing a security tactic during the architecture and design phase.

**•** Implementation: N/A

## Common Consequences

**•** Impact: Modify Application Data, Unexpected State — Notes: Attackers can modify unexpected objects or data that was assumed to be safe from modification. Deserialized data or code could be modified without using the provided accessor functions, or unexpected functions could be invoked.

**•** Impact: DoS: Resource Consumption (CPU) — Notes: If a function is making an assumption on when to terminate, based on a sentry in a string, it could easily never terminate.

**•** Impact: Varies by Context — Notes: The consequences can vary widely, because it depends on which objects or methods are being deserialized, and how they are used. Making an assumption that the code in the deserialized object is valid is dangerous and can enable exploitation. One example is attackers using gadget chains to perform unauthorized actions, such as generating a shell.

## Potential Mitigations

**•** Architecture and Design: If available, use the signing/sealing features of the programming language to assure that deserialized data has not been tainted. For example, a hash-based message authentication code (HMAC) could be used to ensure that data has not been modified. (Effectiveness: N/A)

**•** Implementation: When deserializing data, populate a new object rather than just deserializing. The result is that the data flows through safe input validation and that the functions are safe. (Effectiveness: N/A)

**•** Implementation: Explicitly define a final object() to prevent deserialization. (Effectiveness: N/A)

**•** Architecture and Design: Make fields transient to protect them from deserialization. An attempt to serialize and then deserialize a class containing transient fields will result in NULLs where the transient data should be. This is an excellent way to prevent time, environment-based, or sensitive variables from being carried over and used improperly. (Effectiveness: N/A)

**•** Implementation: Avoid having unnecessary types or gadgets (a sequence of instances and method invocations that can self-execute during the deserialization process, often found in libraries) available that can be leveraged for malicious ends. This limits the potential for unintended or unauthorized types and gadgets to be leveraged by the attacker. Add only acceptable classes to an allowlist. Note: new gadgets are constantly being discovered, so this alone is not a sufficient mitigation. (Effectiveness: N/A)

**•** Architecture and Design: Employ cryptography of the data or code for protection. However, it's important to note that it would still be client-side security. This is risky because if the client is compromised then the security implemented on the client (the cryptography) can be bypassed. (Effectiveness: N/A)

## Applicable Platforms

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

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

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

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

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

## Demonstrative Examples

**•** This code does not attempt to verify the source or contents of the file before deserializing it. An attacker may be able to replace the intended file with a file that contains arbitrary malicious code which will be executed when the button is pressed.

**•** Unfortunately, the code does not verify that the incoming data is legitimate. An attacker can construct a illegitimate, serialized object "AuthToken" that instantiates one of Python's subprocesses to execute arbitrary commands. For instance,the attacker could construct a pickle that leverages Python's subprocess module, which spawns new processes and includes a number of arguments for various uses. Since Pickle allows objects to define the process for how they should be unpickled, the attacker can direct the unpickle process to call Popen in the subprocess module and execute /bin/sh.

## Notes

**•** Maintenance: The relationships between CWE-502 and CWE-915 need further exploration. CWE-915 is more narrowly scoped to object modification, and is not necessarily used for deserialization.