A8: Insecure Deserialization

- Receipt of malicious serialized objects resulting in remote code execution.
- Attackers can build illegitimate object that execute commands within an infected application.

- Serialization:
  Before data is transmitted, the bits are often serialized so that they can be restored to the data's original structure.
- Deserialization:
  Reassembling a series of bits back into a file is called deserialization.

Exploitation of deserialization is somewhat difficult, as off the shelf exploits rarely work without changes or tweaks to the underlying exploit code.

This issue is included in the Top 10 based on an industry survey and not on quantifiable data.
Some tools can discover deserialization flaws, but human assistance is frequently needed to validate the problem. It is expected that prevalence data for deserialization flaws will increase as tooling is developed to help identify and address it.

The impact of deserialization flaws cannot be understated. These flaws can lead to remote code execution attacks, one of the most serious attacks possible.
The business impact depends on the protection needs of the application and data.

Is the Application Vulnerable?

Applications and APIs will be vulnerable if they deserialize hostile or tampered objects supplied by an attacker.

This can result in two primary types of attacks:

- Object and data structure related attacks where the attacker modifies application logic or achieves arbitrary remote code execution if there are classes available to the application that can change behavior during or after deserialization.

- Typical data tampering attacks, such as access-control-related attacks, where existing data structures are used but the content is changed.
  Serialization may be used in applications for:

- Remote- and inter-process communication (RPC/IPC)

- Wire protocols, web services, message brokers

- Caching/Persistence

- Databases, cache servers, file systems

- HTTP cookies, HTML form parameters, API authentication tokens

Example Attack Scenarios

Scenario #1: A React application calls a set of Spring Boot microservices. Being functional programmers, they tried to ensure that their code is immutable. The solution they came up with is serializing user state and passing it back and forth with each request. An attacker notices the "R00" Java object signature, and uses the Java Serial Killer tool to gain remote code execution on the application server.

Scenario #2: A PHP forum uses PHP object serialization to save a "super" cookie, containing the user's user ID, role, password hash, and other state:
**a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}**
An attacker changes the serialized object to give themselves admin privileges:
**a:4:{i:0;i:1;i:1;s:5:"Alice";i:2;s:5:"admin";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}**

How to Prevent

The only safe architectural pattern is not to accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types.

If that is not possible, consider one of more of the following:

- Implementing integrity checks such as digital signatures on any serialized objects to prevent hostile object creation or data tampering.
- Enforcing strict type constraints during deserialization before object creation as the code typically expects a definable set of classes. Bypasses to this technique have been demonstrated, so reliance solely on this is not advisable.
- Isolating and running code that deserializes in low privilege environments when possible.
- Logging deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.
- Restricting or monitoring incoming and outgoing network connectivity from containers or servers that deserialize.
- Monitoring deserialization, alerting if a user deserializes constantly.