A screenshot of a video game

Description automatically generated with medium confidence

**Description**

Unsafe Deserialization (also referred to as Insecure Deserialization) is a vulnerability wherein malformed and untrusted data input is used to compromise an application's logic, resulting in a denial of service or even arbitrary code execution once the input is deserialized. Although this isn't exactly a simple attack to employ, it featured in [OWASP's Top 10](https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A8-Insecure_Deserialization) most recent iteration due to the severity of impact upon compromise.

The process of converting an object state or data structure into a storable or transmissible format is called serialization. Deserialization is its opposite - the process of extracting the serialized data to reconstruct the original object version.

Unsafe Deserialization issues arise when an attacker is able to pass ad-hoc malicious data into user-supplied data to be deserialized. This could result in arbitrary object injection into the application that might influence the correct target behavior.

**Impact**

A successful Unsafe Deserialization attack can result in the full compromise of the confidentiality, integrity, and availability of the target system, and the oft-cited Equifax breach is probably the best example of the worst outcome that can arise. In Equifax's case, an unsafe Java deserialization attack leveraging the Struts 2 framework resulted in remote code execution, which, in turn, led to the largest data breach in history.

**Prevention**

It is important to consider any development project from an architectural standpoint to determine when and where serialization is necessary. If it is unnecessary, consider using a simpler format when passing data.

In cases where it is impossible to forego serialization without disrupting the application's operational integrity, developers can implement a range of defence-in-depth measures to mitigate the chances of being exploited.

* Use serialization that only permits primitive data types.
* Use a serialization library that provides cryptographic signature and encryption features to ensure serialized data are obtained untainted.
* Authenticate before deserializing.
* Use low privilege environments to isolate and run code that deserializes.

Finally, if possible, replace object serialization with data-only serialization formats, such as JSON.

**Testing**

Verify that serialization is not used when communicating with untrusted clients. If this is not possible, ensure that adequate integrity controls (and possibly encryption if sensitive data is sent) are enforced to prevent deserialization attacks including object injection.

* **OWASP ASVS**: [1.5.2, 5.5.1, 5.5.3](https://github.com/OWASP/ASVS/releases/download/v4.0.2_release/OWASP.Application.Security.Verification.Standard.4.0.2-en.pdf)

**.NET**

**Vulnerable Example**

The .NET framework offers several instances of deserialization. Developers will likely be familiar with the following example, where some untrusted binary data is *deserialized* to create some objects:

[**Serializable**]

**public** **class** **SomeClass**

{

**public** **string** SomeProperty { **get**; **set**; }

**public** **double** SomeOtherProperty { **get**; **set**; }

}

**class** **Program**

{

**static** **void** **Main**(**string**[] args)

{

BinaryFormatter binaryFormatter = **new** BinaryFormatter();

MemoryStream memoryStream = **new** MemoryStream(File.ReadAllBytes("untrusted.file"));

SomeClass obj = (SomeClass)binaryFormatter.Deserialize(memoryStream);

Console.WriteLine(obj.SomeProperty);

Console.WriteLine(obj.SomeOtherProperty);

}

}

The above program merrily deserializes not only instances of SomeClass (even though a class cast error is raised for other objects), but also might be enough to trigger dangerous behaviors. For example, a malicious user could leverage publicly available tools such as [ysoserial.net](https://github.com/pwntester/ysoserial.net) to easily craft payloads that exploit the presence of external libraries, and thus build a chain of gadgets that eventually lead to RCE.

Alternatively, an attacker with knowledge of the source code of the application could attempt to locate dangerous classes in the code base. For example, suppose that somewhere in the application, the following class is defined:

[**Serializable**]

**public** **class** **DangerousClass**

{

**private** **string** path;

**public** **DangerousClass**(String path) {

**this**.path = path;

}

**public** ~DangerousClass() {

File.Delete(path)

}

}

The attacker is then able to build such objects locally using an arbitrary path as a parameter, serialize it, and finally feed it to the vulnerable application. When said object is eventually removed from memory by the garbage collector, the attacker gains the ability to delete arbitrary files in the system.

**Prevention**

Never pass user-supplied input to BinaryFormatter; the documentation states this explicitly:

The BinaryFormatter type is dangerous and is not recommended for data processing. Applications should stop using BinaryFormatter as soon as possible, even if they believe the data they're processing to be trustworthy. BinaryFormatter is insecure and can't be made secure.

When possible, developers are encouraged to use other forms of data serialization, such as XML, JSON, or the BinaryReader and BinaryWriter classes. The latter is the recommended approach for binary serialization. For example, in the above scenario, the serialization phase could be implemented as:

**var** someObject = **new** SomeClass();

someObject.SomeProperty = "some value";

someObject.SomeOtherProperty = 3.14;

**using** (BinaryWriter writer = **new** BinaryWriter(File.Open("untrusted.file", FileMode.Create)))

{

writer.Write(someObject.SomeProperty);

writer.Write(someObject.SomeOtherProperty);

}

And in turn, the deserialization phase as:

**var** someObject = **new** SomeClass();

**using** (BinaryReader reader = **new** BinaryReader(File.Open("untrusted.file", FileMode.Open)))

{

someObject.SomeProperty = reader.ReadString();

someObject.SomeOtherProperty = reader.ReadDouble();

}

**References**

[OWASP - Deserialization Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html#java)  
[Wikipedia - Serialization](https://en.wikipedia.org/wiki/Serialization)  
[Microsoft - BinaryFormatter security guide](https://docs.microsoft.com/en-us/dotnet/standard/serialization/binaryformatter-security-guide)  
[Black Hat - Breaking .NET Through Serialization](https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf)