Deserialization Cheat Sheet

Introduction

This article is focused on providing clear, actionable guidance for safely deserializing untrusted data in your applications.

What is Deserialization

Serialization is the process of turning some object into a data format that can be restored later. People often serialize objects in order to save them to storage, or to send as part of communications.

Deserialization is the reverse of that process, taking data structured from some format, and rebuilding it into an object. Today, the most popular data format for serializing data is JSON. Before that, it was XML.

However, many programming languages offer a native capability for serializing objects. These native formats usually offer more features than JSON or XML, including customizability of the serialization process.

Unfortunately, the features of these native deserialization mechanisms can be repurposed for malicious effect when operating on untrusted data. Attacks against deserializers have been found to allow denial-of-service, access control, and remote code execution (RCE) attacks.

Guidance on Deserializing Objects Safely

The following language-specific guidance attempts to enumerate safe methodologies for deserializing data that can't be trusted.

PHP

WhiteBox Review

Check the use of unserialize() function and review how the external parameters are accepted. Use a safe, standard data interchange format such as JSON (via `json_decode()` and `json_encode()`) if you need to pass serialized data to the user.

Python

BlackBox Review

If the traffic data contains the symbol dot `.` at the end, it's very likely that the data was sent in serialization.

WhiteBox Review

The following API in Python will be vulnerable to serialization attack. Search code for the pattern below.

1. The uses of `pickle/c_pickle/_pickle` with `load/loads`:

```
import pickle
data = """ cos.system(S'dir')tR. """
pickle.loads(data)
```

1. Uses of `PyYAML` with `load`:

```
import yaml
document = "!!python/object/apply:os.system ['ipconfig']"
print(yaml.load(document))
```

1. Uses of `jsonpickle` with `encode` or `store` methods.

Java

The following techniques are all good for preventing attacks against deserialization against Java's Serializable format.

Implementation advices:

- In your code, override the `ObjectInputStream#resolveClass()` method to prevent arbitrary classes from being deserialized. This safe behavior can be wrapped in a library like SerialKiller.
- Use a safe replacement for the generic `readObject()` method as seen here. Note that this addresses "billion laughs" type attacks by checking input length and number of objects deserialized.

WhiteBox Review

Be aware of the following Java API uses for potential serialization vulnerability.

1. `XMLdecoder` with external user defined parameters

2. `XStream` with `fromXML` method
(xstream version <= v1.46 is vulnerable to the serialization issue)

3. `ObjectInputStream` with `readObject`

4. Uses of `readObject`, `readObjectNodData`, `readResolve` or `readExternal`

5. `ObjectInputStream.readUnshared`

6. `Serializable`

BlackBox Review

If the captured traffic data include the following patterns may suggest that the data was sent in Java serialization streams

- `AC ED 00 05` in Hex
- `rO0` in Base64
- `Content-type` header of an HTTP response set to `application/x-java-serialized-object`

Prevent Data Leakage and Trusted Field Clobbering

If there are data members of an object that should never be controlled by end users during deserialization or exposed to users during serialization, they should be declared as the `transient` keyword (section *Protecting Sensitive Information*).

For a class that defined as Serializable, the sensitive information variable should be declared as `private transient`.

For example, the class myAccount, the variable 'profit' and 'margin' were declared as transient to avoid to be serialized:

```
public class myAccount implements Serializable
{
    private transient double profit; // declared transient

    private transient double margin; // declared transient
    ....
```

Prevent Deserialization of Domain Objects

Some of your application objects may be forced to implement Serializable due to their hierarchy. To guarantee that your application objects can't be deserialized, a `readObject()` method should be declared (with a `final` modifier) which always throws an exception:

```
private final void readObject(ObjectInputStream in) throws java.io.IOException {
    throw new java.io.IOException("Cannot be deserialized");
}
```

Harden Your Own java.io.ObjectInputStream

The `java.io.ObjectInputStream` class is used to deserialize objects. It's possible to harden its behavior by subclassing it. This is the best solution if:

- You can change the code that does the deserialization
- You know what classes you expect to deserialize

The general idea is to override `ObjectInputStream.html#resolveClass()` in order to restrict which classes are allowed to be deserialized.

Because this call happens before a `readObject()` is called, you can be sure that no deserialization activity will occur unless the type is one that you wish to allow.

A simple example of this shown here, where the the `LookAheadObjectInputStream` class is guaranteed not to deserialize any other type besides the `Bicycle` class:

```
public class LookAheadObjectInputStream extends ObjectInputStream {

    public LookAheadObjectInputStream(InputStream inputStream) throws IOException
{
        super(inputStream);
    }

    /**
    * Only deserialize instances of our expected Bicycle class
    */
    @Override
    protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException,
ClassNotFoundException {
        if (!desc.getName().equals(Bicycle.class.getName())) {
            throw new InvalidClassException("Unauthorized deserialization
attempt", desc.getName());
        }
        return super.resolveClass(desc);
    }
}
```

More complete implementations of this approach have been proposed by various community members:

- NibbleSec - a library that allows creating lists of classes that are allowed to be deserialized

- IBM - the seminal protection, written years before the most devastating exploitation scenarios were envisioned.
- Apache Commons IO classes

Harden All java.io.ObjectInputStream Usage with an Agent

As mentioned above, the `java.io.ObjectInputStream` class is used to deserialize objects. It's possible to harden its behavior by subclassing it. However, if you don't own the code or can't wait for a patch, using an agent to weave in hardening to `java.io.ObjectInputStream` is the best solution.

Globally changing `ObjectInputStream` is only safe for block-listing known malicious types, because it's not possible to know for all applications what the expected classes to be deserialized are. Fortunately, there are very few classes needed in the blocklist to be safe from all the known attack vectors, today.

It's inevitable that more "gadget" classes will be discovered that can be abused. However, there is an incredible amount of vulnerable software exposed today, in need of a fix. In some cases, "fixing" the vulnerability may involve re-architecting messaging systems and breaking backwards compatibility as developers move towards not accepting serialized objects.

To enable these agents, simply add a new JVM parameter:

```
-javaagent:name-of-agent.jar
```

Agents taking this approach have been released by various community members:

- rO0 by Contrast Security

A similar, but less scalable approach would be to manually patch and bootstrap your JVM's ObjectInputStream. Guidance on this approach is available here.

.Net CSharp

WhiteBox Review

Search the source code for the following terms:

1. `TypeNameHandling`
2. `JavaScriptTypeResolver`

Look for any serializers where the type is set by a user controlled variable.

BlackBox Review

Search for the following base64 encoded content that starts with:

```
AAEAAAD/////
```

Search for content with the following text:

1. `TypeObject`
2. `$type:`

General Precautions

Don't allow the datastream to define the type of object that the stream will be deserialized to. You can prevent this by for example using the `DataContractSerializer` or `XmlSerializer` if at all possible.

Where `JSON.Net` is being used make sure the `TypeNameHandling` is only set to `None`.

```
TypeNameHandling = TypeNameHandling.None
```

If `JavaScriptSerializer` is to be used then do not use it with a `JavaScriptTypeResolver`.

If you must deserialise data streams that define their own type, then restrict the types that are allowed to be deserialized. One should be aware that this is still risky as many native .Net types potentially dangerous in themselves. e.g.

```
System.IO.FileInfo
```

`FileInfo` objects that reference files actually on the server can when deserialized, change the properties of those files e.g. to read-only, creating a potential denial of service attack.

Even if you have limited the types that can be deserialised remember that some types have properties that are risky. `System.ComponentModel.DataAnnotations.ValidationException`, for example has a property `Value` of type `Object`. if this type is the type allowed for deserialization then an attacker can set the `Value` property to any object type they choose.

Attackers should be prevented from steering the type that will be instantiated. If this is possible then even `DataContractSerializer` or `XmlSerializer` can be subverted e.g.

```
// Action below is dangerous if the attacker can change the data in the database
var typename = GetTransactionTypeFromDatabase();

var serializer = new DataContractJsonSerializer(Type.GetType(typename));

var obj = serializer.ReadObject(ms);
```

Execution can occur within certain .Net types during deserialization. Creating a control such as the one shown below is ineffective.

```
var suspectObject = myBinaryFormatter.Deserialize(untrustedData);

//Check below is too late! Execution may have already occurred.
if (suspectObject is SomeDangerousObjectType)
{
    //generate warnings and dispose of suspectObject
}
```

For `BinaryFormatter` and `JSON.Net` it is possible to create a safer form of allow-list control using a custom `SerializationBinder`.

Try to keep up-to-date on known .Net insecure deserialization gadgets and pay special attention where such types can be created by your deserialization processes. **A deserializer can only instantiate types that it knows about**.

Try to keep any code that might create potential gadgets separate from any code that has internet connectivity. As an example `System.Windows.Data.ObjectDataProvider` used in WPF applications is a known gadget that allows arbitrary method invocation. It would be risky to have this a reference to this assembly in a REST service project that deserializes untrusted data.

Known .NET RCE Gadgets

- `System.Configuration.Install.AssemblyInstaller`
- `System.Activities.Presentation.WorkflowDesigner`
- `System.Windows.ResourceDictionary`
- `System.Windows.Data.ObjectDataProvider`
- `System.Windows.Forms.BindingSource`
- `Microsoft.Exchange.Management.SystemManager.WinForms.ExchangeSettingsProvider`
- `System.Data.DataViewManager, System.Xml.XmlDocument/XmlDataDocument`
- `System.Management.Automation.PSObject`

Language-Agnostic Methods for Deserializing Safely

Using Alternative Data Formats

A great reduction of risk is achieved by avoiding native (de)serialization formats. By switching to a pure data format like JSON or XML, you lessen the chance of custom deserialization logic being repurposed towards malicious ends.

Many applications rely on a data-transfer object pattern that involves creating a separate domain of objects for the explicit purpose data transfer. Of course, it's still possible that the application will make security mistakes after a pure data object is parsed.

Only Deserialize Signed Data

If the application knows before deserialization which messages will need to be processed, they could sign them as part of the serialization process. The application could then to choose not to deserialize any message which didn't have an authenticated signature.

Mitigation Tools/Libraries

- Java secure deserialization library
- SWAT (Serial Whitelist Application Trainer)
- NotSoSerial

Detection Tools

- Java deserialization cheat sheet aimed at pen testers
- A proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization.
- Java De-serialization toolkits
- Java de-serialization tool
- .Net payload generator
- Burp Suite extension
- Java secure deserialization library
- Serianalyzer is a static bytecode analyzer for deserialization
- Payload generator
- Android Java Deserialization Vulnerability Tester
- Burp Suite Extension
  - JavaSerialKiller
  - Java Deserialization Scanner
  - Burp-ysoserial
  - SuperSerial
  - SuperSerial-Active