UNIVERSITÀ DELLA CALABRIA
il Campus per eccellenza

Insecure deserialization + Server-side template injection

Mario Alviano

Main References

Bug Bounty Bootcamp – Chapters 14 and 16

https://portswigger.net/web-security/deserialization

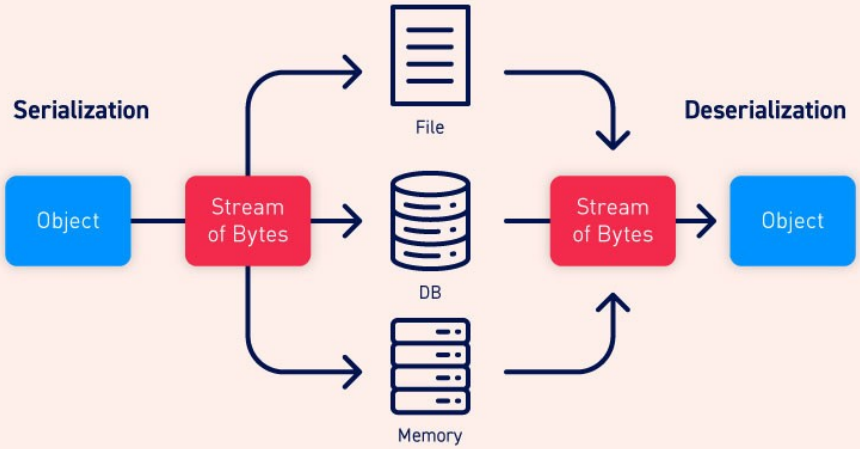
https://portswigger.net/web-security/server-side-template-injection

OWASP Top Ten

A broad consensus about the most critical security risks to web applications

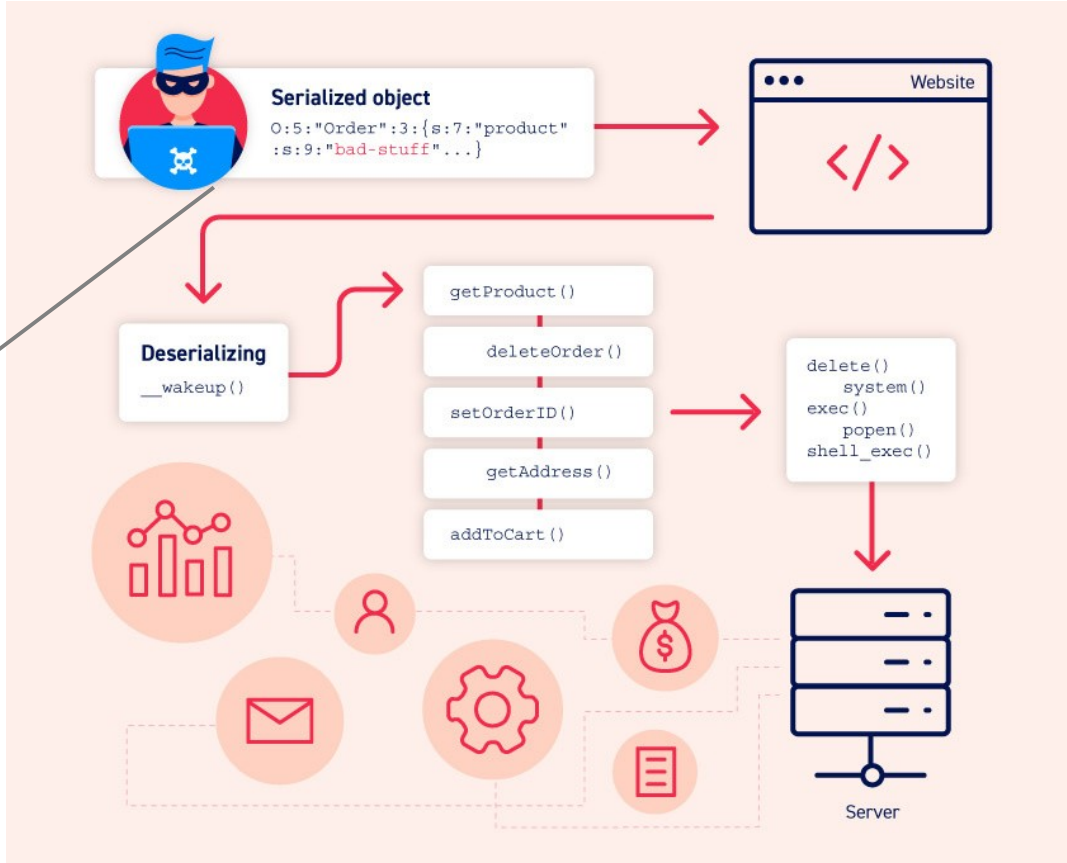
Insecure deserialization

Many programming languages have serialization and deserialization mechanisms to store and transfer objects. Often such mechanisms are unsafe.



Why do you want users to send their objects?

There is not a lot to say…
DON'T DO IT!



PHP serialization format

Don’t use serialize() and deserialize()

Use json_encode() and json_decode() instead!

```
<?php
❶ class User{
    public $username;
    public $status;
  }
❷ $user = new User;
❸ $user->username = 'vickie';
❹ $user->status = 'not admin';
❺ echo serialize($user);
?>
```

```
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}
```

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

Object of type User with 2 properties

First property name is the string “username”

First property value is the string “vickie”

Second property name is the string “status”

Second property value is the string “not admin”

Object tampering

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

If the user can send any string to be unserialized on the server side, who prevent them from modifying the object?

The only way to be sure that the user is sending back the same string you created is by adding some form of cryptographic signature.

Property-Oriented Programming (POP) chains

Some **magic methods** are called automatically.
- __constructor() and __wakeup() are used during instantiation.
- __destruct() is called when the object is not needed anymore
- __call() is invoked when an undefined method is called

If these methods interpret some properties of the unserialized object as code, there is RCE.

Note that users may tamper also the class name, hence it's sufficient to have a single vulnerable class to have problems.

Even if there is no vulnerable class, there may be enough classes enabling several small operation via their properties.
By chaining calls to such properties one can achieve RCE.

There are tools for finding such gadgets and combining them to obtain RCE, like PHP Generic Gadget Chains (PHPGGC).

Server-Side Template Injection (SSTI)

Template engines are designed to generate web pages by combining fixed templates with volatile data.

Server-side template injection attacks can occur when user input is concatenated directly into a template, rather than passed in as data.

Template Engines

There are many!
- Jinja, Django, Mako (Python)
- Smarty, Twig (PHP)
- Apache FreeMarker, Apache Velocity (Java)

Jinja is a popular template language for Python.

Literal HTML content

Python expressions

Loop statement

Conditional statement

```
<html>
  <body>
❶ <h1>{{ list_title }}</h1>
  <h2>{{ list_description }}</h2>
❷ {% for item in item_list %}
    {{ item }}
    {% if not loop.last %},{% endif %}
  {% endfor %}
  </body>
</html>
```

```python
from jinja2 import Template

with open("example.jinja") as template_file:
    template = Template(template_file.read())
    print(template.render(
        list_title = "Chapter Contents",
        list_description = "Here are the contents of chapter 16.",
        item_list = [
            "Mechanisms Of Template Injection",
            "Preventing Template Injection",
            "Hunting For Template Injection",
            "Escalating Template Injection",
            "Automating Template Injection",
            "Find Your First Template Injection!",
        ],
    ))
```

Data binding

example.jinja

```
<html>
  <body>
❶ <h1>{{ list_title }}</h1>
  <h2>{{ list_description }}</h2>
❷ {% for item in item_list %}
    {{ item }}
    {% if not loop.last %},{% endif %}
  {% endfor %}
  </body>
</html>
```

```
<html>
  <body>
  <h1>Chapter Contents</h1>
  <h2>Here are the contents of chapter 16.</h2>
  Mechanisms Of Template Injection,
  Preventing Template Injection,
  Hunting For Template Injection,
  Escalating Template Injection,
  Automating Template Injection,
  Find Your First Template Injection!
  </body>
</html>
```

Correct use of templates

```
from jinja2 import Template
with open('example.jinja') as f:
    tmpl = Template(f.read())
print(tmpl.render(
   ❶ list_title = user_input.title,
   ❷ list_description = user_input.description,
   ❸ item_list = user_input.list,
))
```

Bind data from user. The template engine takes care of escaping content to prevent XSS.

Incorrect use of templates

Template injection vulnerabilities happen when a user is able to inject input into templates without proper sanitization.

```python
from jinja2 import Template

name = request.GET("name")
template = Template(f"""
<html>
<body>
    <h1>Hello {name}!</h1>
</body>
</html>
""")
print(template.render())
```

It works…

```
GET /display_name?name=Vickie
Host: example.com
```

```
<html>
<body>
    <h1>Hello Vickie!</h1>
</body>
</html>
```

Incorrect use of templates

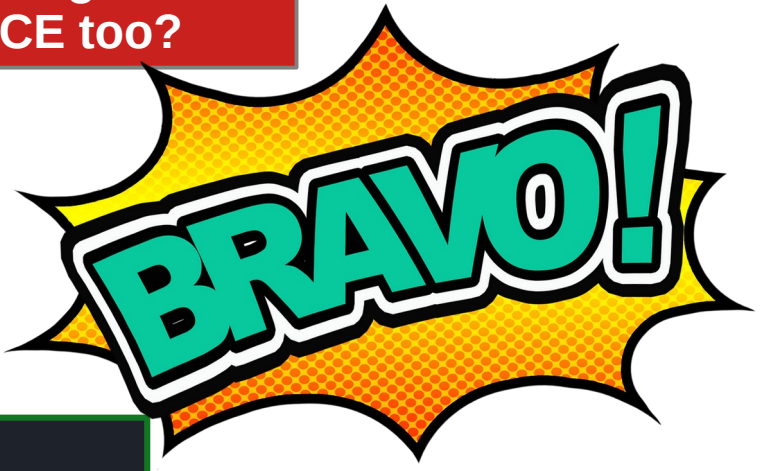
Template injection vulnerabilities happen when a user is able to inject input into templates without proper sanitization.

```python
from jinja2 import Template

name = request.GET("name")
template = Template(f"""
<html>
<body>
    <h1>Hello {name}!</h1>
</body>
</html>
""")
print(template.render())
```

It works… under the assumption that name is just a name!

XSS was not enough… so you added RCE too?




```
GET /display_name?name=Vickie
Host: example.com
```

It's just the expected path!

```html
<html>
<body>
    <h1>Hello Vickie!</h1>
</body>
</html>
```

```python
from jinja2 import Template

name = request.GET("name")
template = Template(f"""
<html>
<body>
    <h1>Hello {name}!</h1>
</body>
</html>
""")
print(template.render())
```

{name} is going to be expanded with {{1+1}}, which will be interpreted as an expression by the Jinja2 template engine!

Note that 1+1 is evaluated server-side.

```
GET /display_name?name={{1+1}}
Host: example.com
```

Unexpected path!

```
<html>
<body>
    <h1>Hello 2!</h1>
</body>
</html>
```

Prevention

- Don’t process user templates server-side
- If you have to, sandbox and disable dangerous modules
- Implement allowlist for allowed attributes

Use data binding as much as possible.
Don’t concatenate user strings with the template!

Questions