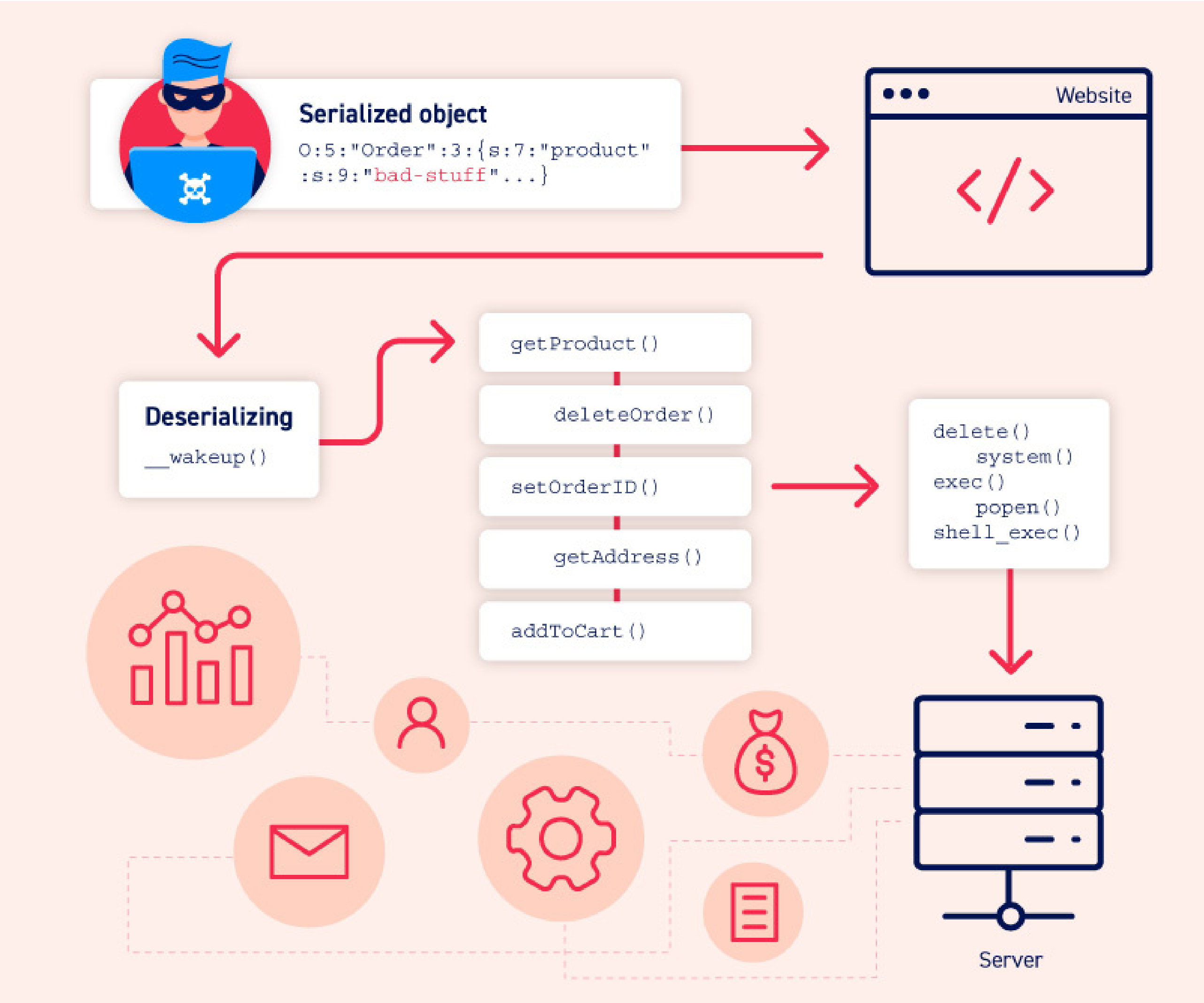
INSECURE DESERIALIZATION

Many programming languages have serialization and deserialization mechanisms to store and transfer objects.
In some cases these mechanisms can open the doors to problems and possible attacks.



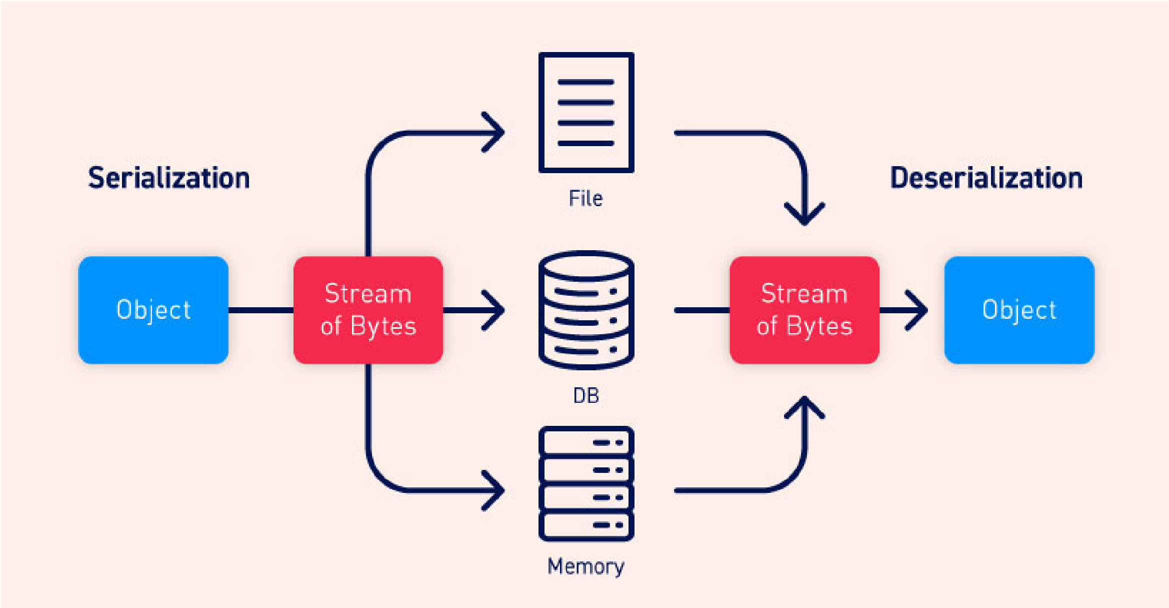
- in this case the attacker changes the php serialization of an object injecting some malicious stuff

WHAT IS SERIALIZATION?

Serialization is the process of converting complex data structures such as objects and their fields into a "flatter" format that can be sent and received as a sequential stream of bytes.

For example JSON is a way to serialize object generally in a safe way.

SERIALIZATION vs DESERIALIZATION



Deserialization is the process of restoring objects into the original state in which they were serialized from byte stream.

Different languages serialize objects in different ways.

What is insecure deserialization?

The insecure deserialization happens when user-controllable data is deserialized by a website.

This allows users to manipulate the serialized object in order to pass malicious data to the application code.

It is also possible to replace a serialized object with an object of an enterily different class.

How do insecure deserialization vulnerabilities arise?

Insecure deserialization arises because there is a lack of understanding how dangerous deserializing user-controllable data can be.

User input should never be deserialized at all.

It can also arise when object are often assumed to be trustworthy.

What is the impact of insecure deserialization?

It allows an attacker to reuse existing application code in harmful ways, resulting in numerous other vulnerabilities, often remote code execution.

Even in cases where remote code execution is not possible, **insecure deserialization can lead to privilege escalation, arbitrary file access, and denial-of-service attacks.**

How to identify insecure deserialization

PHP serialization format

PHP uses a human-readable string format:

- **letters represent the data type of the entry**
- **numbers represent length of the entry**

For example we have the class User with two attributes:

```
$user->name = "carlos";

$user->isLoggedIn = true;
```

when the object is serialized it becomes:

O:4:"User":2:{s:4:"name":s:6:"carlos"; s:10:"isLoggedIn":b:1;}

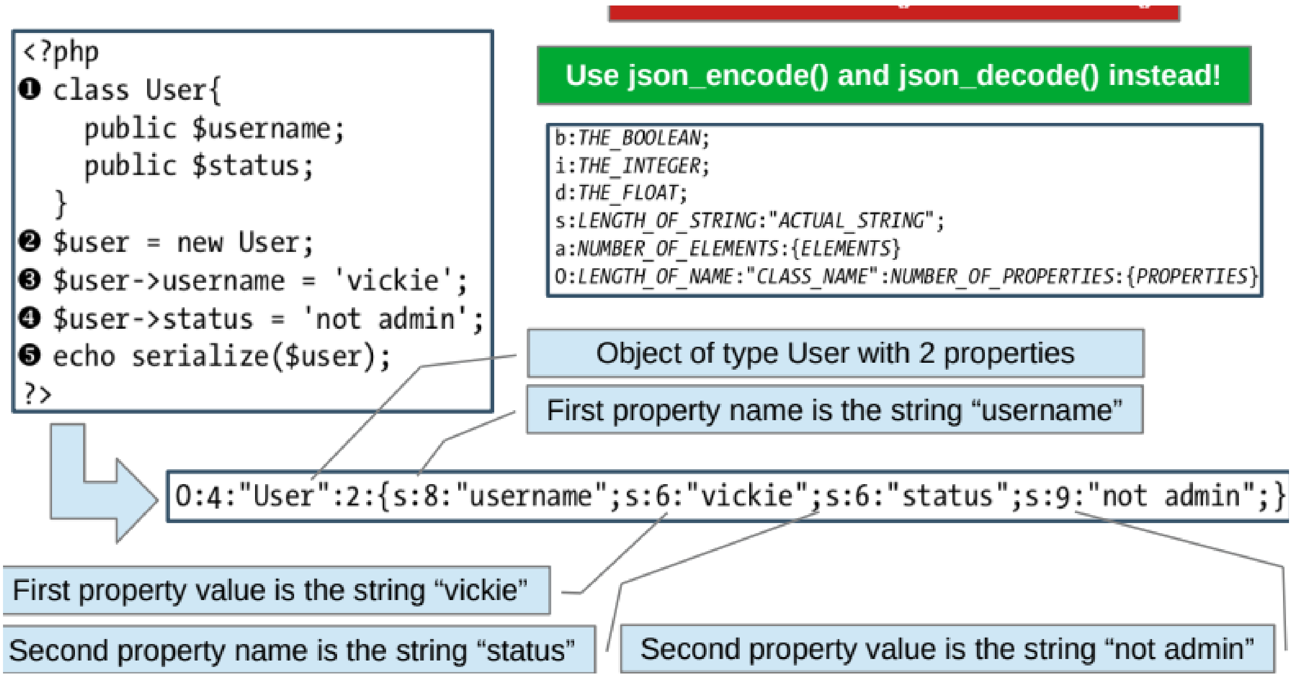
This can be interpreted as follows:

- `O:4:"User"` → An object with the 4-character class name `"User"`
- `:2` → the object has 2 attributes
- `s:4:"name"` → The key of the first attribute is a 4-character string `"name"`
- `s:6:"carlos"` → The value of the first attribute is the 6-character string `"carlos"`

•`s:10:"isLoggedIn"` → The key of the second attribute is the 10-character string `"isLoggedIn"`

•`b:1` → The value of the second attribute is the boolean value `true`

PHP uses two native methods that are serialize() and unserialize()



The idea is to avoid serialize() and unserialize() but instead use json_encode() and json_decode()

what an attacker can do is the Object tampering, where the serialized object is changed in order to perform a malicious action. For example:

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

- in this case the attacker can obtain the admin functionalities, if the backend is based only on this object serialization

The only way to be sure that the user is sending back the string the server created is to use a cryptographic signature

Modifying data types

PHP-based logic is particularly vulnerable to this kind of manipulation due to the behavior of its loose comparison operator (==) when comparing different data types.

For example, if you perform a loose comparison between an integer and a string, PHP will attempt to convert the string to an integer, meaning that 5 == "5" evaluates to true.

Unusually, this also works for any alphanumeric string that starts with a number.

Therefore, 5 == "5 of something" is in practice treated as 5 == 5.

This becomes even stranger when comparing a string the integer 0:

0 == "Example string"

- **this is true because there are no numbers in the string so it is treated as 0**

Let’s imagine this behavior:

$login = unserialize($_COOKIE)

if ($login['acess_token'] == $access_token) {

// log in successfully

}

if the access_token is sent into a serialized object then the attacker can manipulate it in this way:

O:4:"User":2:{s:8:"username";s:13:"administrator";s:12:"access_token";i:0;}

- it is changed in the integer 0 with i:0 so the login will be successful

Property-Oriented Programming (POP) chains

Magic methods

They are methods that are invoked automatically whenever a particular event or scenario occurs.

Magic methods are a common feature of object-oriented programming in various languages.

They are sometimes indicated by prefixing or surrounding the method name with double-underscores.

One of the most common examples in PHP is __construct(), which is invoked whenever an object of the class is instantiated, similar to Python's __init__.

Some languages have magic methods that are invoked automatically during the deserialization process.

For example, PHP's unserialize() method looks for and invokes an object's __wakeup() magic method.

In Java deserialization, the same applies to the ObjectInputStream.readObject() method, which is used to read data from the initial byte stream and essentially acts like a constructor for "re-initializing" a serialized object.

If these methods interpret some properties of the userialized object as code then there is the door for a RCE.

Even if there is no vulnerable class maybe there can be enogh classes that allows small operation via their properties, and if the attacker can chain them he can achieve RCE.

There are some tools that can be used to find gadgets and to combine them to achieve RCE (for example PHP Generic Gadget Chains)

Injecting arbitrary objects

Deserialization methods do not typically check what they are deserializing.

This means that you can pass in objects of any serializable class that is available to the website, and the object will be deserialized.

This effectively allows an attacker to create instances of arbitrary classes.

If an attacker has access to the source code, they can study all of the available classes in detail.

To construct a simple exploit, they would look for classes containing deserialization magic methods, then check whether any of them perform dangerous operations on controllable data.

Preventing

1. avoid the user input data deserialization, if it is needed validate it or use a cryptographic signature.

2. expose to serialization only the fields that doesn't contain sensitive data.

SERVER-SIDE TEMPLATE INJECTION

Template engines are designed to generate web pages by combining fixed templates and volatile data, in order to reuse the template instead of write always a equal new page.

Let's imagine the page for showing a product on amazon, it is always the same, so we can use the template in order to fill the page with the different products values but using always the same page template.

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

This is a very huge problem because what the attacker inject is executed on server-side.

This allows attackers to inject arbitrary template directives in order to manipulate the template engine, often enabling them to take complete control of the server.

What are the most used template engines?

There are many template engines:

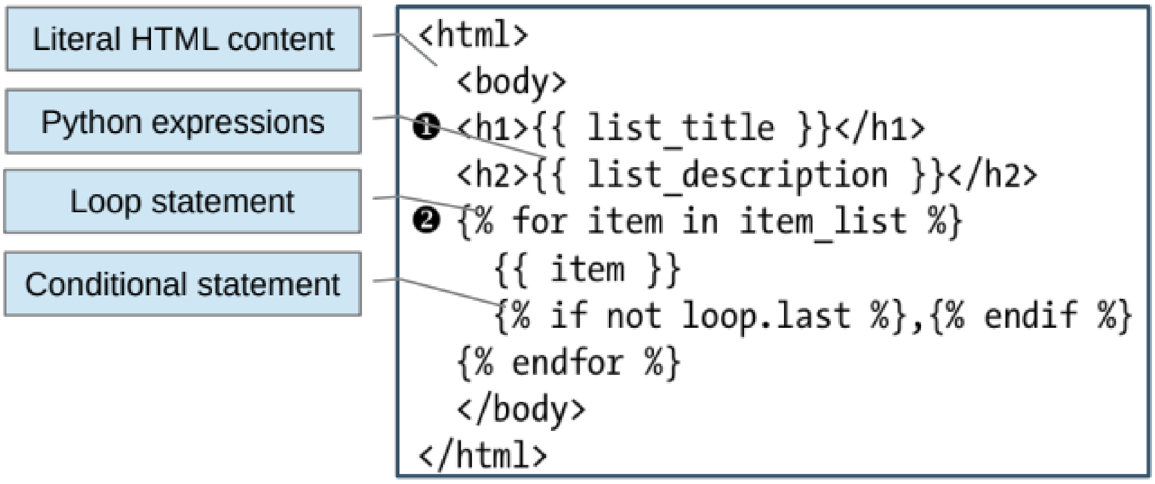
- Jinja, Django, Mako for Python
- Smarty, Twig for PHP
- Apache FreeMaker, Apache Velocity for Java

The most used one for Python is nowadays Jinja.

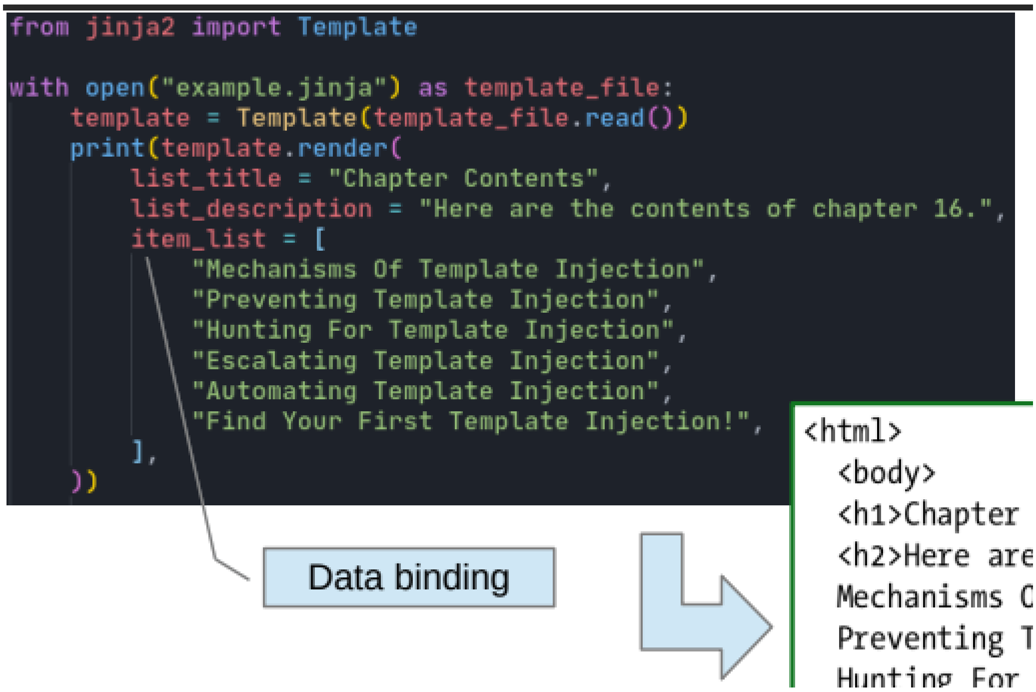
What is the impact of server-side template injection?

At the severe end of the scale, an attacker can potentially achieve remote code execution, taking full control of the back-end server and using it to perform other attacks on internal infrastructure.

But in general an attacker can perform numerous attacks such as read sensitive data and arbitrary files on the server.



In this case we cana have a back end that does:



In this case we will obtain in the page rendering:

```
<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>
```

The correct use of templates is to avoid the concatenation of strings obtained by the users but use only the data binding.

This because the template in the rendering phase is able to escape the data in oder to prevent attacks and for example XSS. It is not able to do it if we concatenate strings.

```
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,
))
```

- in this case the template is used in a safe way

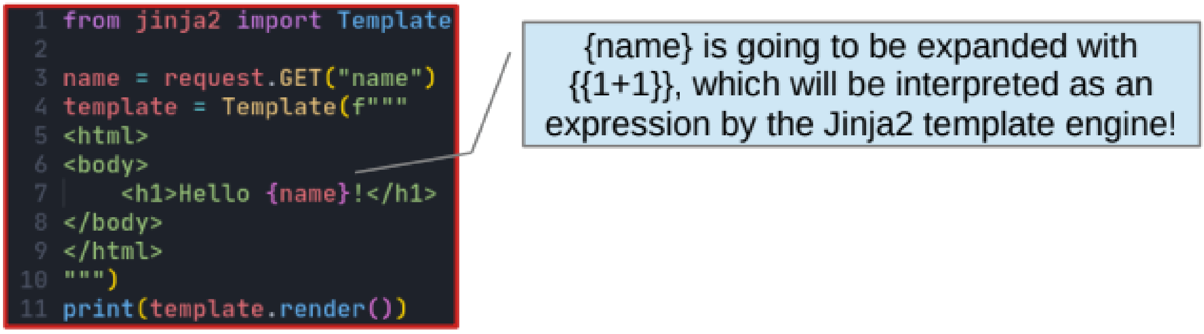
In this other case it is very dangerous:

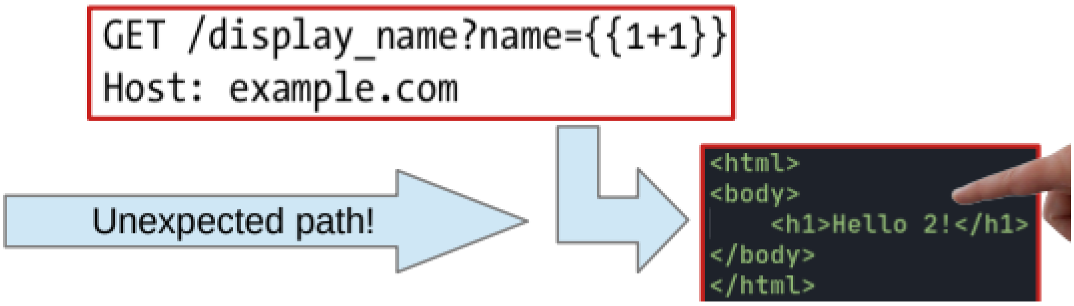
```
from jinja2 import Template

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

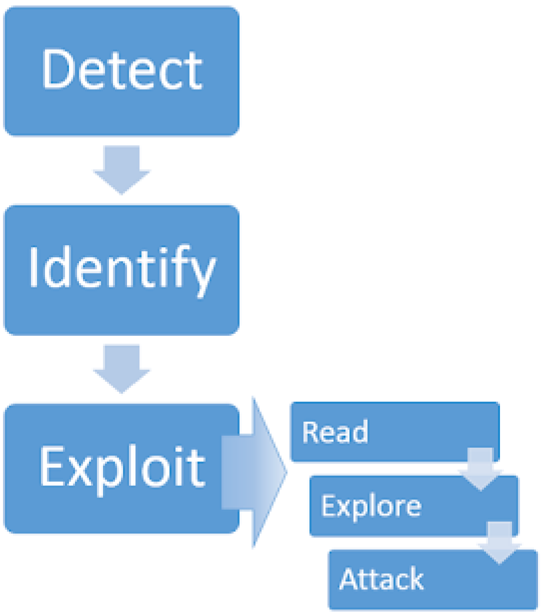
- it works only under the assumption that name is just a name.

If fact in this case we can run arbitrary code or force the server to intepret expressions:





Constructing a server-side template injection attack



Detect

The simplest initial approach is to try fuzzing the template by injecting a sequence of special characters commonly used in template expressions, such as ${{<%[%'"}}%\.

If an exception is raised, this indicates that the injected template syntax is potentially being interpreted by the server in some way.

Even if fuzzing did suggest a template injection vulnerability, you still need to identify its context in order to exploit it.

Plaintext context

During auditing, we might test for server-side template injection by requesting a URL such as: http://vulnerable-website.com/?username=${7*7}

If the resulting output contains Hello 49 there is a good proof of concept for a server-side template injection vulnerability.

Identify

Simply submitting invalid syntax is often enough because the resulting error message will tell you exactly what the template engine is, and sometimes even which version.

For example, the invalid expression <%=foobar%> triggers the following response from the Ruby-based ERB engine:

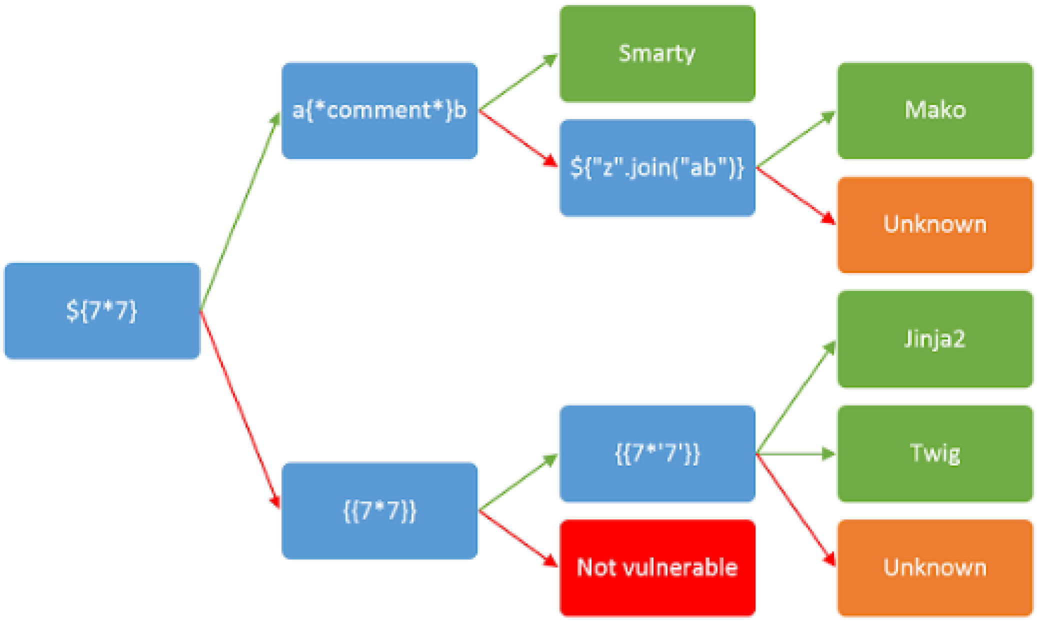
```
(erb):1:in `<main>': undefined local variable or method `fobar' for main:Object (NameError)

from /usr/lib/ruby/2.5.0/erb.rb:876:in `eval'

from /usr/lib/ruby/2.5.0/erb.rb:876:in `result'

from -e:4:in `<main>'
```

Otherwise, you'll need to manually test different language-specific payloads and study how they are interpreted by the template engine.



Exploit

After detecting that a potential vulnerability exists and successfully identifying the template engine, you can begin trying to find ways of exploiting it.

Prevention

1. don’t process user templates on the server-side, is better to process them on client-side
2. if we have to process on server-side, disable dangerous modules
3. implement allow lists for allowed attributes

Use data binding as much as possible and don’t concatenate user strings with the template

OWASP Top Ten

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