



10 RULES

FOR

SAFER

CODE

RULE #1

EVAL IS EVIL

Don't trust **strings** supposed to contain **expressions** or **code**

(even your own!)

“eval” breaks the barrier between **code** and **data**

```
result = self.env.ref('account.%s' % (xml_id)).read()[0]  
invoice_domain = eval(result['domain'])
```

Is this safe?

Maybe... it depends.

Is it a good idea?

No, because eval() is **not** necessary!

There are **safer** and **smarter** ways to **parse** data in **PYTHON**

Given this
string

"42"

"[1,2,3,true]"

'{"widget": "monetary"}'

"[1,2,3,True]"

"{'widget': 'monetary'}"

Parse it
like this

`int(x)`
`float(x)`

`json.loads(x)`

`ast.literal_eval(x)`

There are **safer** and **smarter** ways to **parse** data in **JAVASCRIPT**

Given this
string

"42"

"[1,2,3,true]"

'{"widget": "monetary"}'

Parse it
like this

`parseInt(x)`
`parseFloat(x)`

`JSON.parse(x)`

If you must eval **parameters** use a **safe** eval method

Show your meaning!

PYTHON

Import as "**safe_eval**", not as "eval"!

```
# YES  
from odoo.tools import safe_eval  
res = safe_eval('foo', {'foo': 42});
```

```
# NO  
from odoo.tools import safe_eval as eval  
res = eval('foo', {'foo': 42});
```

Alias built-in eval as "**unsafe_eval**"

```
# YES  
unsafe_eval = eval  
res = unsafe_eval(trusted_code);
```

```
# NO!  
res = eval(trusted_code);
```

If you must eval **parameters** use a **safe** eval method

Do not use the built-in JS eval!

JAVASCRIPT

```
// py.js is included by default
py.eval('foo', {'foo': 42});

// require("web.pyeval") for
// domains/contexts/groupby evaluation
pyeval.eval('domains', my_domain);
```

50%

of **vulnerabilities** found every year include
remote code execution injected via

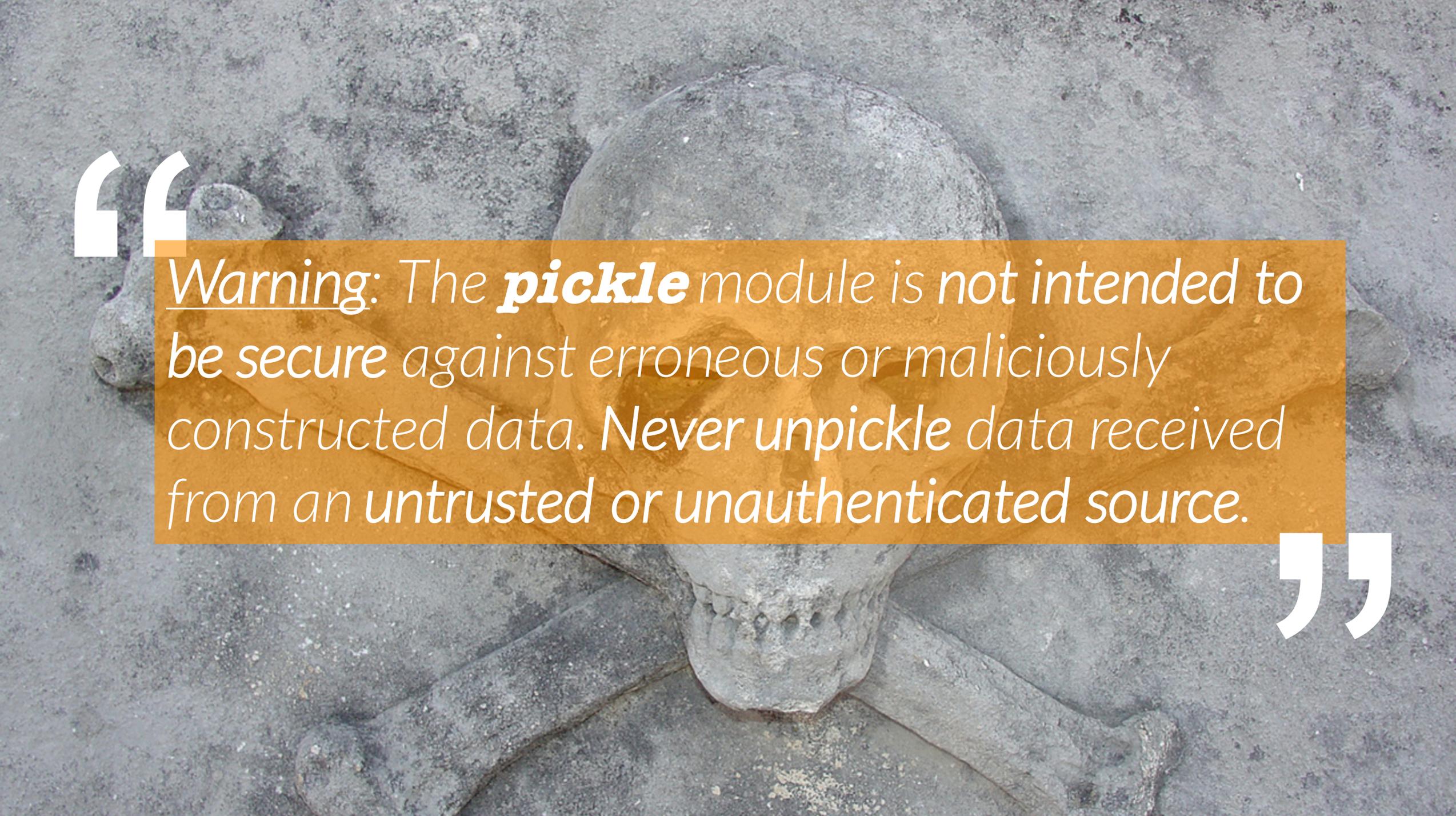
unsafe eval



RULE #2

**YOU SHALL NOT
PICKLE**

Don't use it. Ever. Use JSON.

A skull and crossbones symbol is centered on a dark, textured background. The skull is positioned in the center, with two long, dark bones crossing behind it to form an 'X' shape. The skull is light gray and appears to be made of stone or a similar material. The background is a dark, mottled gray with some lighter spots and textures.

“

Warning: The ***pickle*** module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

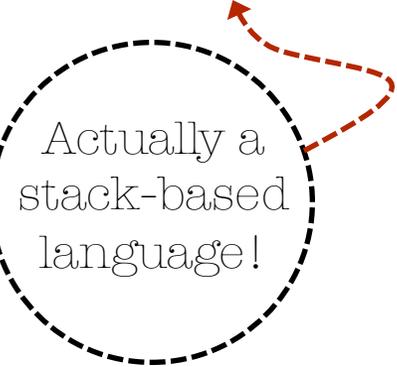
”

Python's pickle serialization is:

+unsafe +not portable

+unreadable

```
pickle.dumps({"widget": "monetary"}) == "(dp0\nS'widget'\np1\nS'monetary'\np2\ns."
```



Actually a
stack-based
language!

Pickle is **unsafe** Seriously.

```
>>> yummy = "cos\nsystem\n(S'cat /etc/shadow | head -n 5'\ntR.'\ntR."  
>>> pickle.loads(yummy)  
root:$6$m7ndoM3p$JRVXomVQFn/KH81DEePpX98usSoESUnm13e6N1f.:14951:0:99999:7:::  
daemon:x:14592:0:99999:7:::  
(...  
>>>
```

Use JSON instead!

```
json.dumps({"widget": "monetary"}) == '{"widget": "monetary"}'
```



+safe +portable
+readable

RULE #3

USE THE CURSOR WISELY

Use the **ORM API**. And when you can't, use **query parameters**.

SQL injection is a classical privilege escalation vector

The **ORM** is here to help you build safe queries:

```
self.search(domain)
```

Psycopg can also help you do that, if you tell it what is **code** and what is **data**:

```
query = """SELECT * FROM res_partner  
        WHERE id IN %s"""  
self._cr.execute(query, (tuple(ids),))
```

SQL code

SQL data parameters



Learn the API to avoid hurting
yourself
and
other people!



This method is vulnerable to SQL injection

```
def compute_balance_by_category(self, categ='in'):  
    query = """SELECT sum(debit-credit)  
                FROM account_invoice_line l  
                JOIN account_invoice i ON (l.invoice_id = i.id)  
                WHERE i.categ = '%s_invoice'  
                GROUP BY i.partner_id """  
    self._cr.execute(query % categ)  
    return self._cr.fetchall()
```

What if someone calls it with

```
categ = """in_invoice'; UPDATE res_users  
SET password = 'god' WHERE id=1; SELECT  
sum(debit-credit) FROM account_invoice_line  
WHERE name = """
```

This method is **still** vulnerable to SQL injection

Now
private!



```
def _compute_balance_by_category(self, categ='in'):  
    query = """SELECT sum(debit-credit)  
                FROM account_invoice_line l  
                JOIN account_invoice i ON (l.invoice_id = i.id)  
                WHERE i.categ = '%s_invoice'  
                GROUP BY i.partner_id """  
    self._cr.execute(query % categ)  
    return self._cr.fetchall()
```

Better, but it could still be called
indirectly!

This method is **still** vulnerable to SQL injection

```
def _compute_balance_by_category(self, categ='in'):  
    assert categ in ('in', 'out')  
    query = """SELECT sum(debit-credit)  
                FROM account_invoice_line l  
                JOIN account_invoice i ON (l.invoice_id = i.id)  
                WHERE i.categ = '%s_invoice'  
                GROUP BY i.partner_id """  
    self._cr.execute(query % categ)  
    return self._cr.fetchall()
```

Now checked
with assert!

Better, but **assert** can be optimized
out and **ignored**
(e.g. in Windows builds)

This method is **safe** against SQL injection

```
def _compute_balance_by_category(self, categ='in'):
```

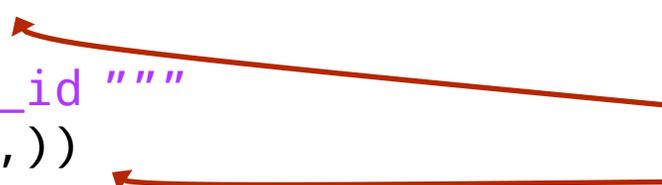
```
    categ = '%s_invoice' % categ
```

```
    query = """SELECT sum(debit-credit)
                FROM account_invoice_line l
                JOIN account_invoice i ON (l.invoice_id = i.id)
                WHERE i.categ = %s
                GROUP BY i.partner_id """
```

```
    self._cr.execute(query, (categ,))
```

```
    return self._cr.fetchall()
```

Separates **code**
and **parameters**!



RULE #4

FIGHT XSS-FORCE

(T-RAW, UPLOADS, HTML FIELDS, ...)

So many **XSS vectors** – gotta **watch** 'em all