10 RULES FOR SAFER CODE

Olivier Dony
@odony

Yet *another* security issue?!

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

PYTHON

Show your meaning!

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

JAVASCRIPT

Do not use the built-in JS eval!

```
// 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%

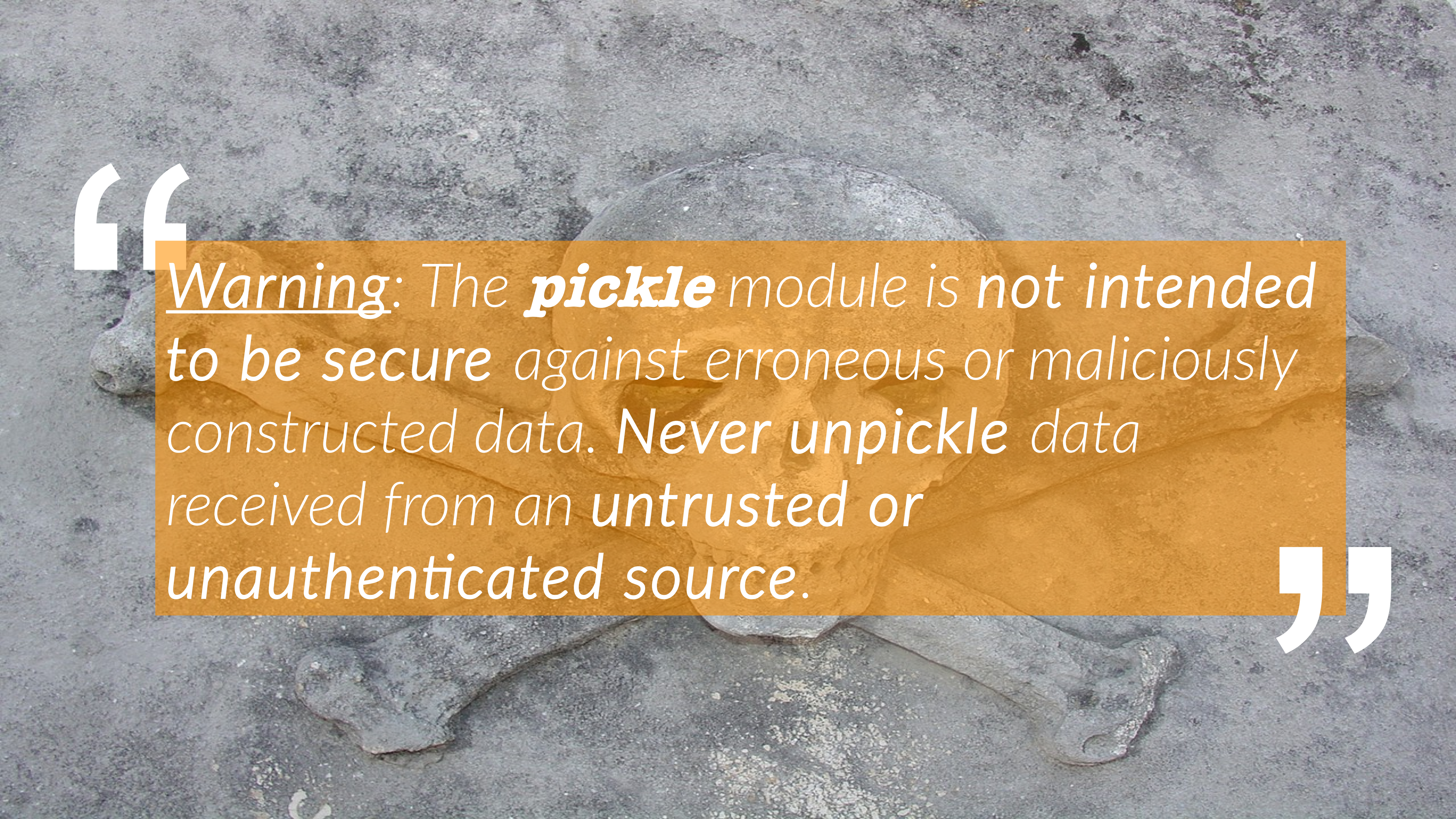
of vulnerabilities found every year include remote code execution injected via

unsafe eval

Rule #2

YOU SHALL NOT PICKLE

Don’t use it. Ever. Use JSON.



“*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
+human-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/KH81DEePpX98usSoESUnml3e6Nlf.:14951:0:99999:7:::
daemon:x:14592:0:99999:7:::
(…)
>>>
```

Use JSON instead!

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

+safe +portable
+human-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 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

So many XSS vectors – gotta watch 'em all

Browsers blur the distinction between code and data!

Most XSS errors are trivial:
QWeb templates

t-raw vs **t-esc** / **t-field**

Only use it to insert **HTML** code that has been prepared and **escaped** by the framework.

Never use it to insert **text**.

For **everything else**, use:

- **t-esc**: variables, URL parameters, …
- **t-field**: record data

Most XSS errors are trivial:
DOM manipulations (JQuery)

`$elem.html(value)` vs `$elem.text(value)`

Only use it to insert **HTML** code that has been prepared and **escaped** by the framework.

Never use it to insert **text**.

For **everything else**, use:

- **t-esc**: variables, URL parameters, ...
- **t-field**: record data

Some XSS are less obvious: **callbacks**

```
@http.route('/web/binary/upload', type='http', auth="user")
@serialize_exception
def upload(self, callback, ufile):
    out = """<script language="javascript" type="text/javascript">
                var win = window.top.window;
                win.jQuery(win).trigger(%s, %s);
            </script>"""
    # (...)
    return out % (json.dumps(callback), json.dumps(args))
```

JSON escaping is not sufficient to prevent XSS,
because of the way browsers parse documents!

```
/web/binary/upload?callback=</script><script>alert('This works!');//
```

Some XSS are less obvious: **uploads**

Users can often upload **arbitrary files** : contact forms, email gateway, etc.

Upon download, browsers will happily detect the file type and **execute** anything that **remotely** looks like HTML, even if you return it with an image mime-type !

A valid SVG image file!

```
<svg xmlns="http://www.w3.org/2000/svg">
  <script type="text/javascript">alert('XSS!!');</script>
</svg>
```

Rule #5

GUARD PASSWORDS & TOKENS FIERCELY

Secure all user and API tokens, and don’t leak them

Where should we store precious tokens for external APIs?

On the `res.users` record of the user!

On the `res.company` record!

On the record representing the API endpoint, like the `acquirer` record!

In the `ir.config_parameter` table!

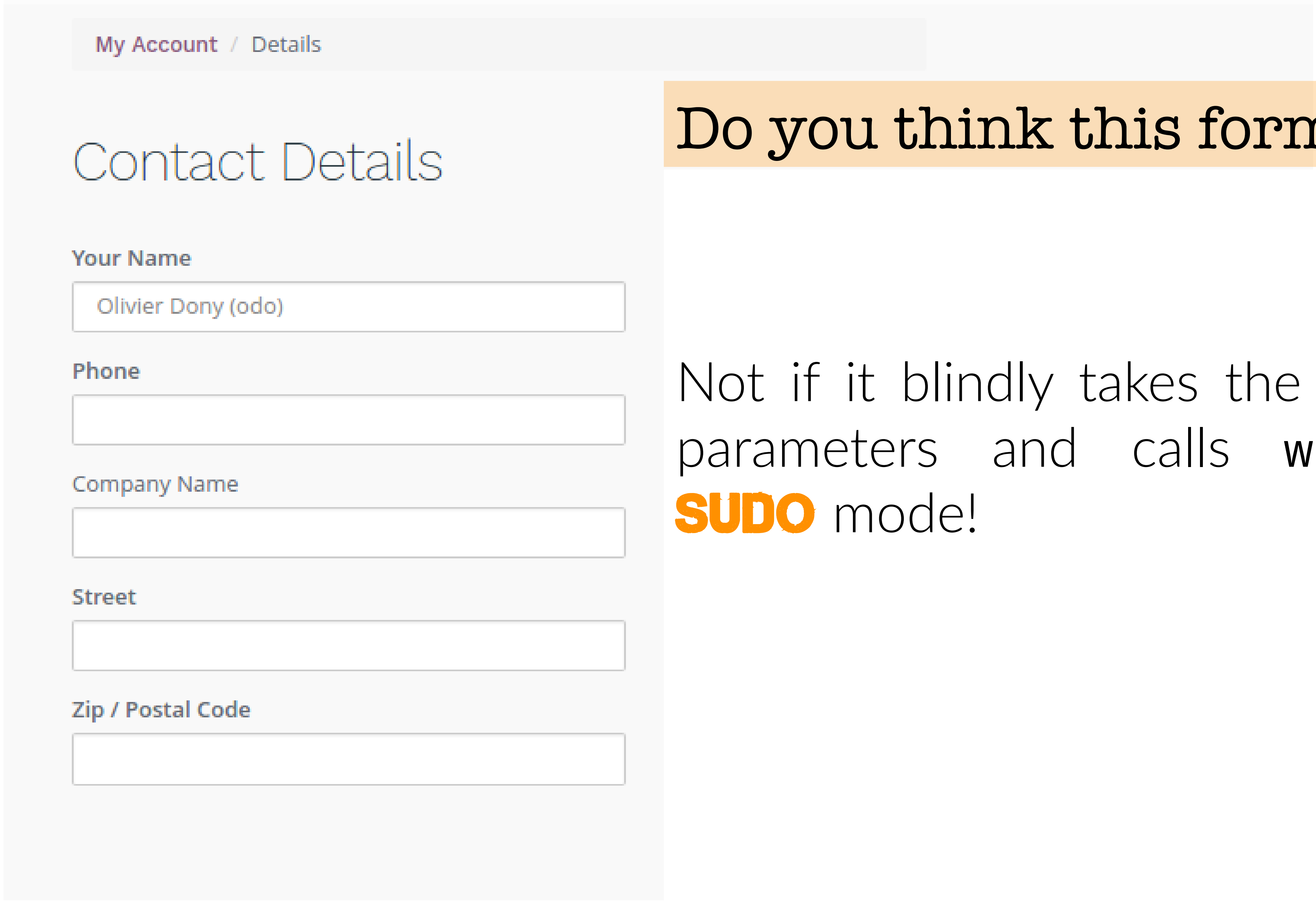
Wherever it makes the most sense, as long as it's **not readable** by anyone!
(field-level group, ACLs, ICP groups, etc.)

Avoid leaking user cookies and Session ID

Rule #6

DO NOT OVER-SUDO IT

Review 2x your sudo() usage, particularly controllers/public methods


My Account / Details

Contact Details

Your Name	Olivier Dony (odo)
Phone	
Company Name	
Street	
Zip / Postal Code	

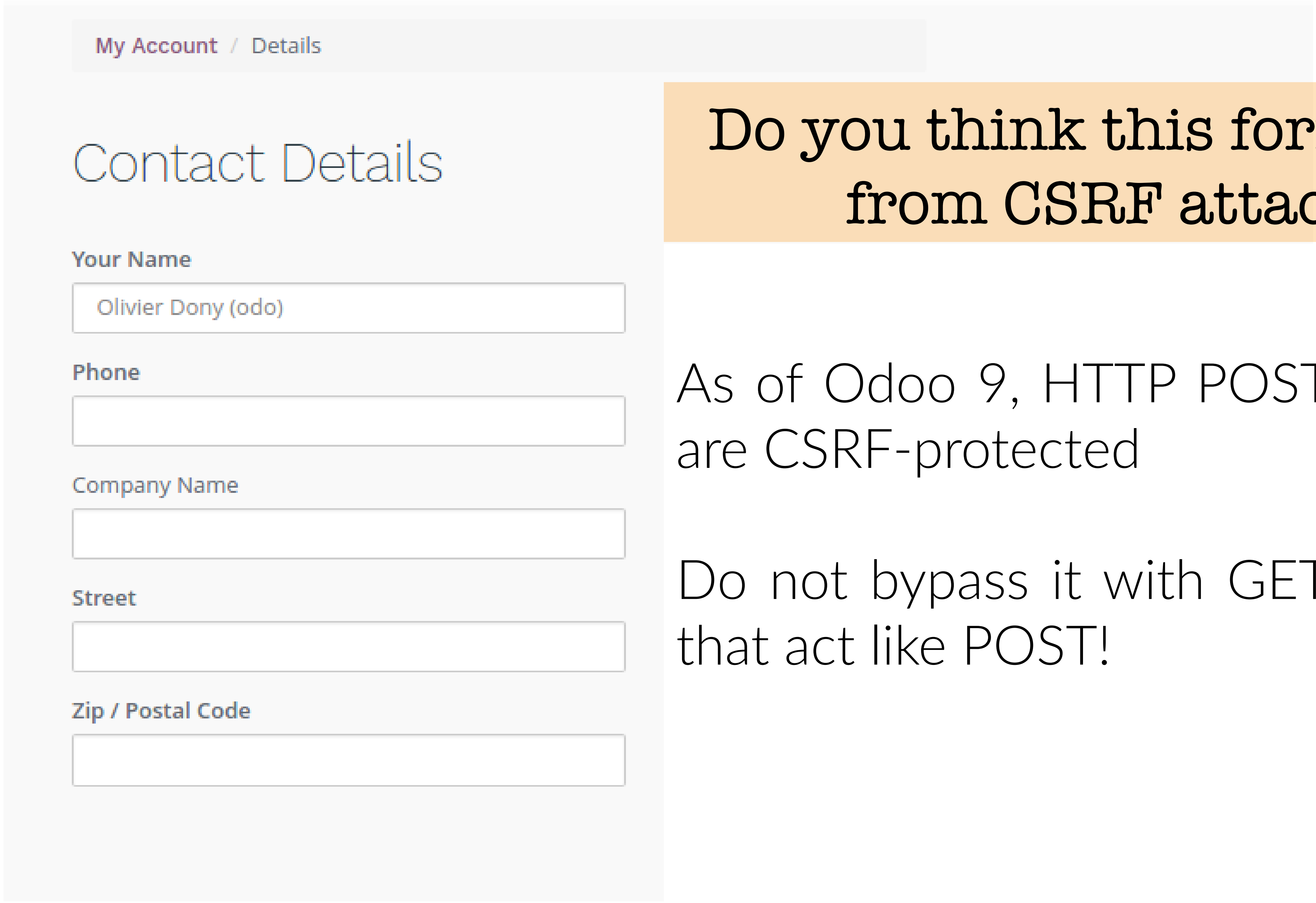
Do you think this form is safe?

Not if it blindly takes the form POST parameters and calls `write()` in **SUDO** mode!

Rule #7

CSRF TOKENS FOR WEBSITE FORMS

HTTP Posts **require** CSRF tokens since v9.0


My Account / Details

Contact Details

Your Name	Olivier Dony (odo)
Phone	
Company Name	
Street	
Zip / Postal Code	

Do you think this form is safe from CSRF attacks?

As of Odoo 9, HTTP POST controllers are CSRF-protected

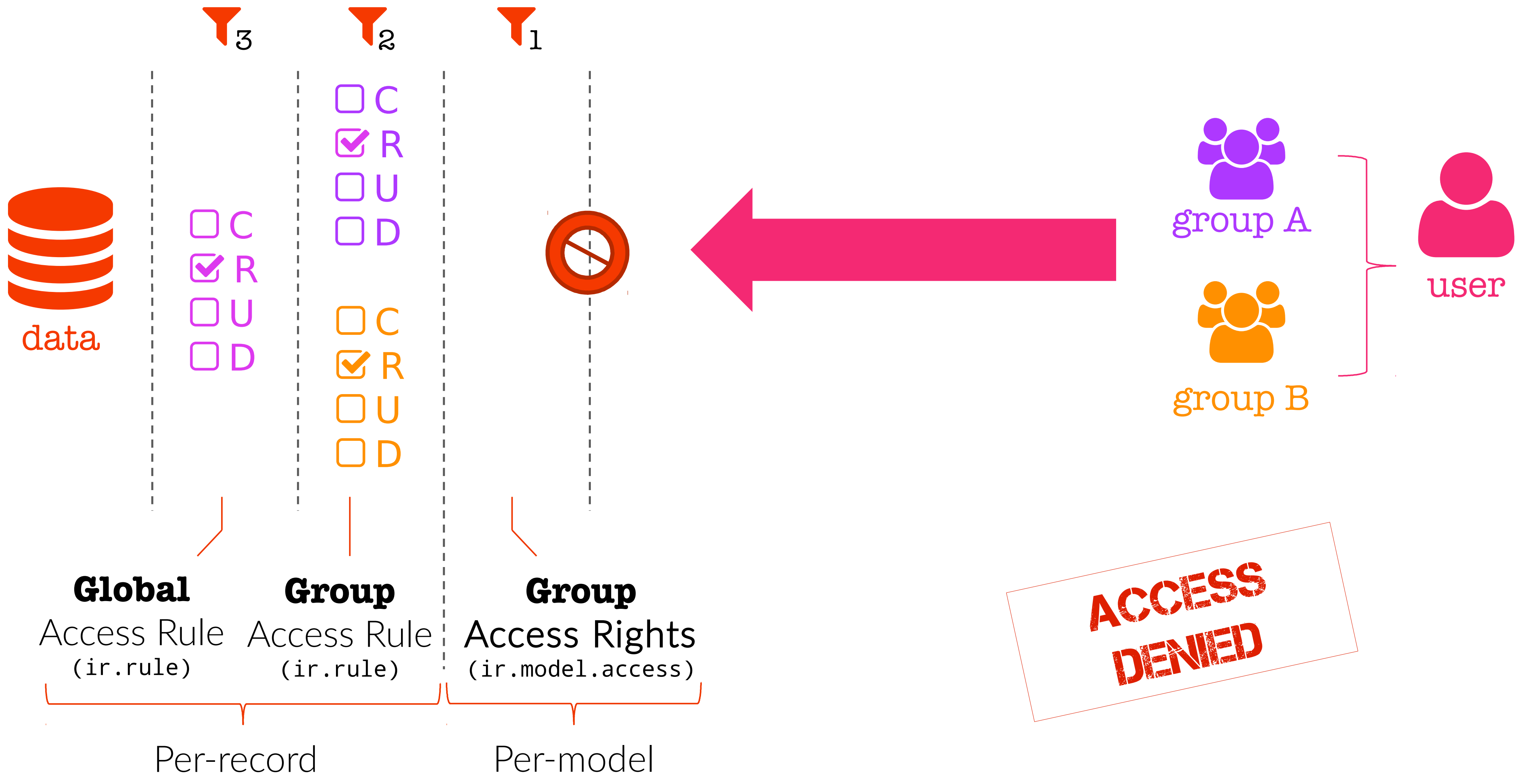
Do not bypass it with GET controllers that act like POST!

Rule #8

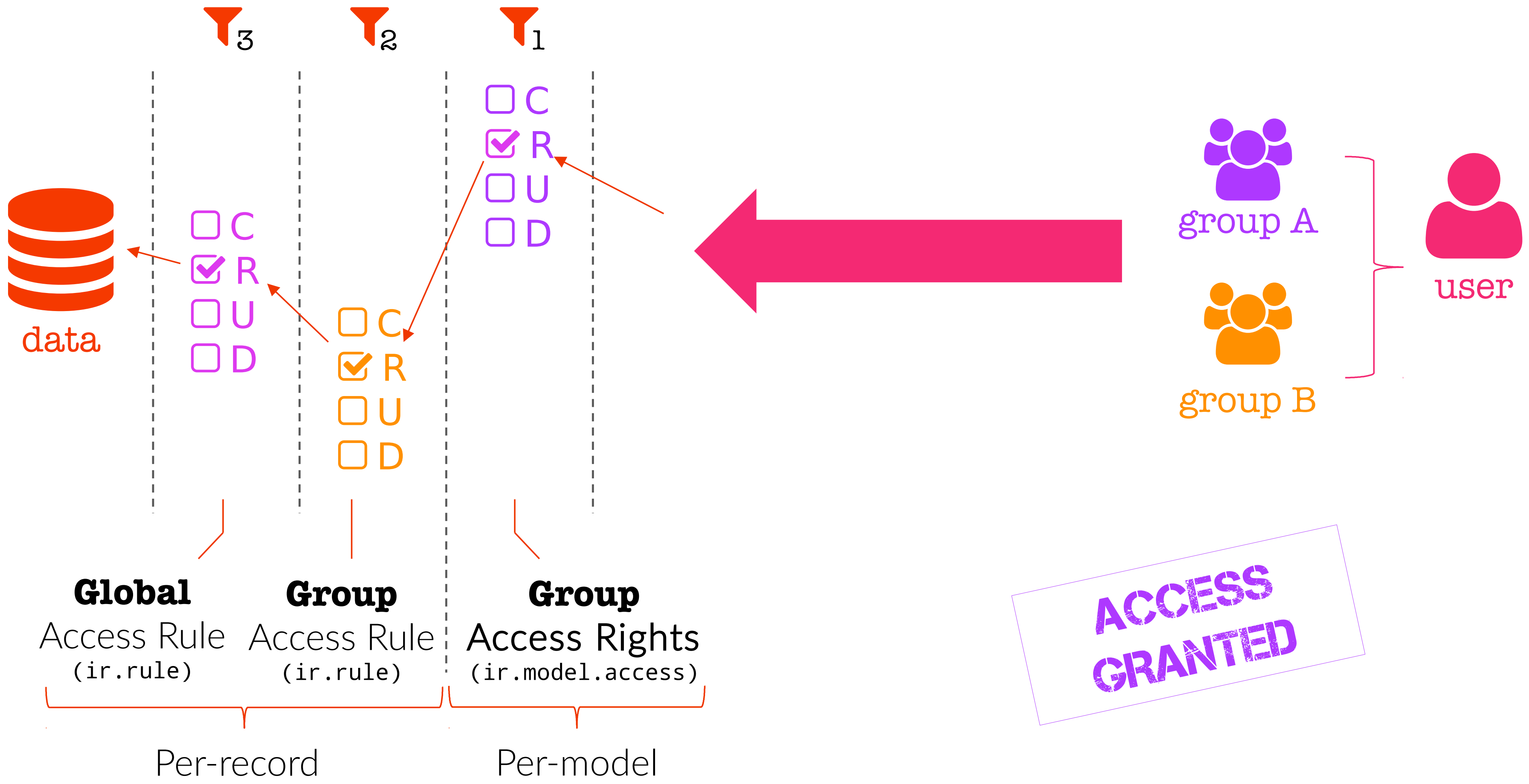
MASTER THE RULES

Odoo ACL and Rules are not trivial, be sure to understand them

Odoo ACLs are not always trivial

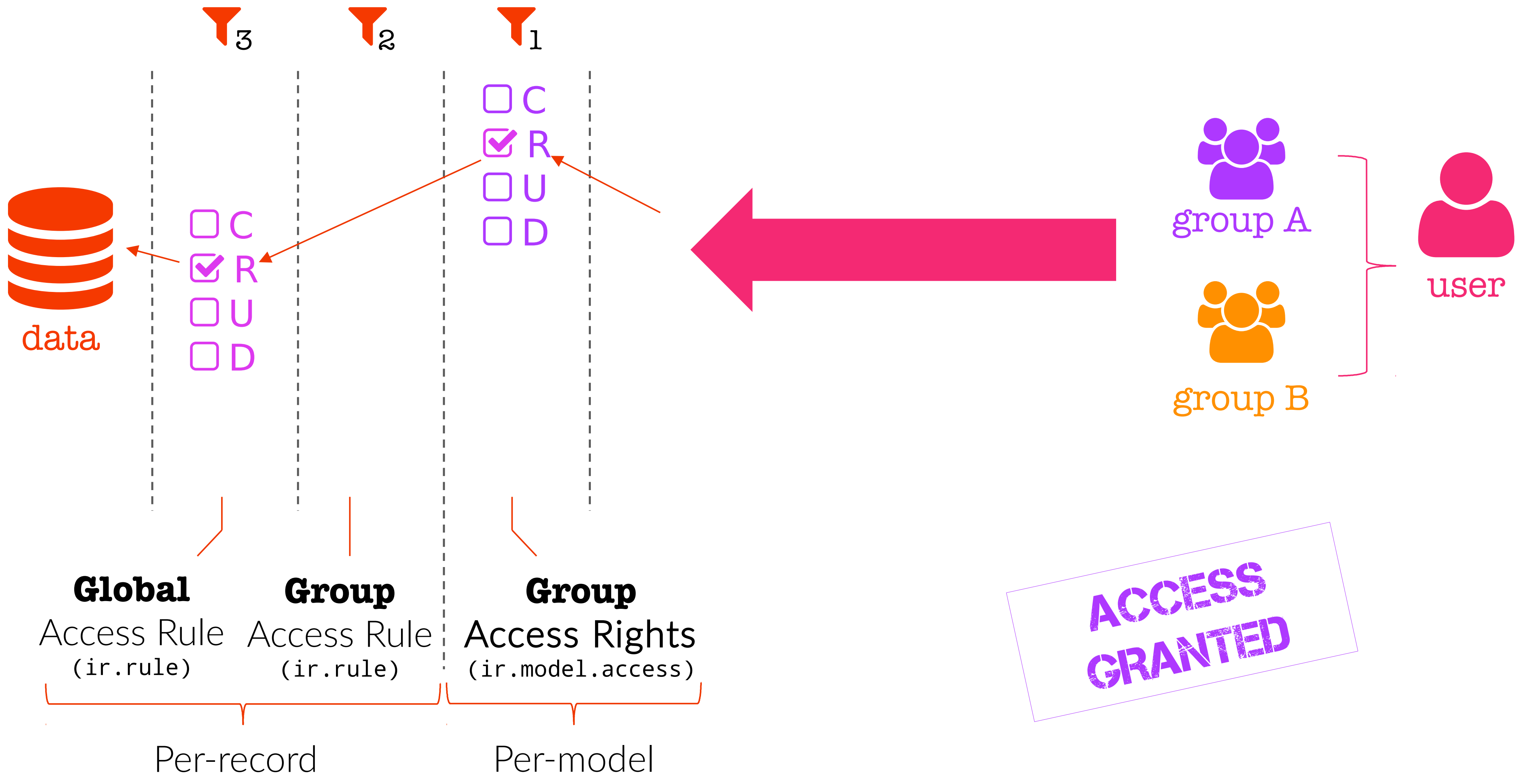

3
2
1
data
C
R
U
D
C
R
U
D
C
R
U
D
group A
group B
user
Global Access Rule (ir.rule)
Group Access Rule (ir.rule)
Group Access Rights (ir.model.access)
Per-record
Per-model

Odoo ACLs are not always trivial

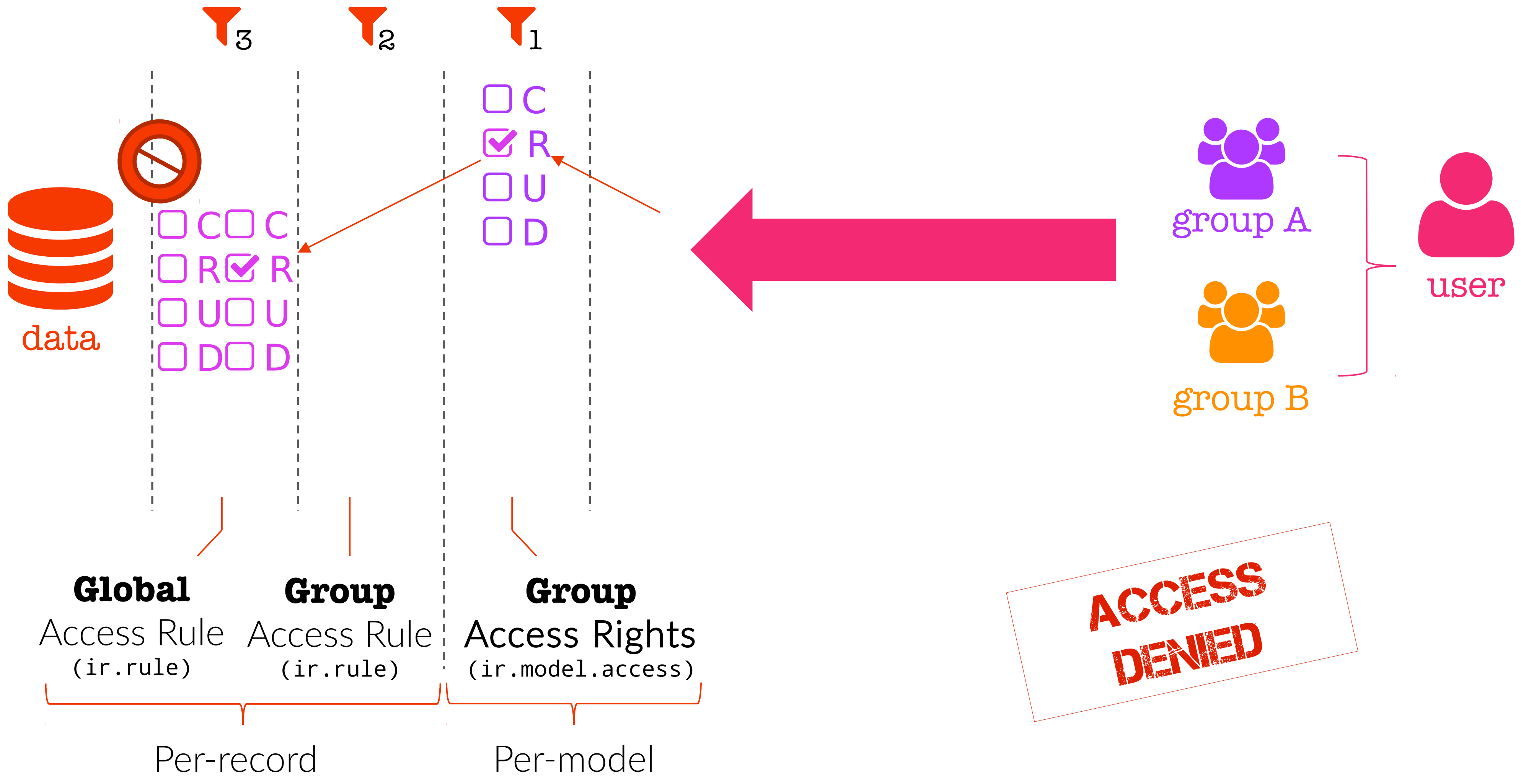


ACCESS GRANTED

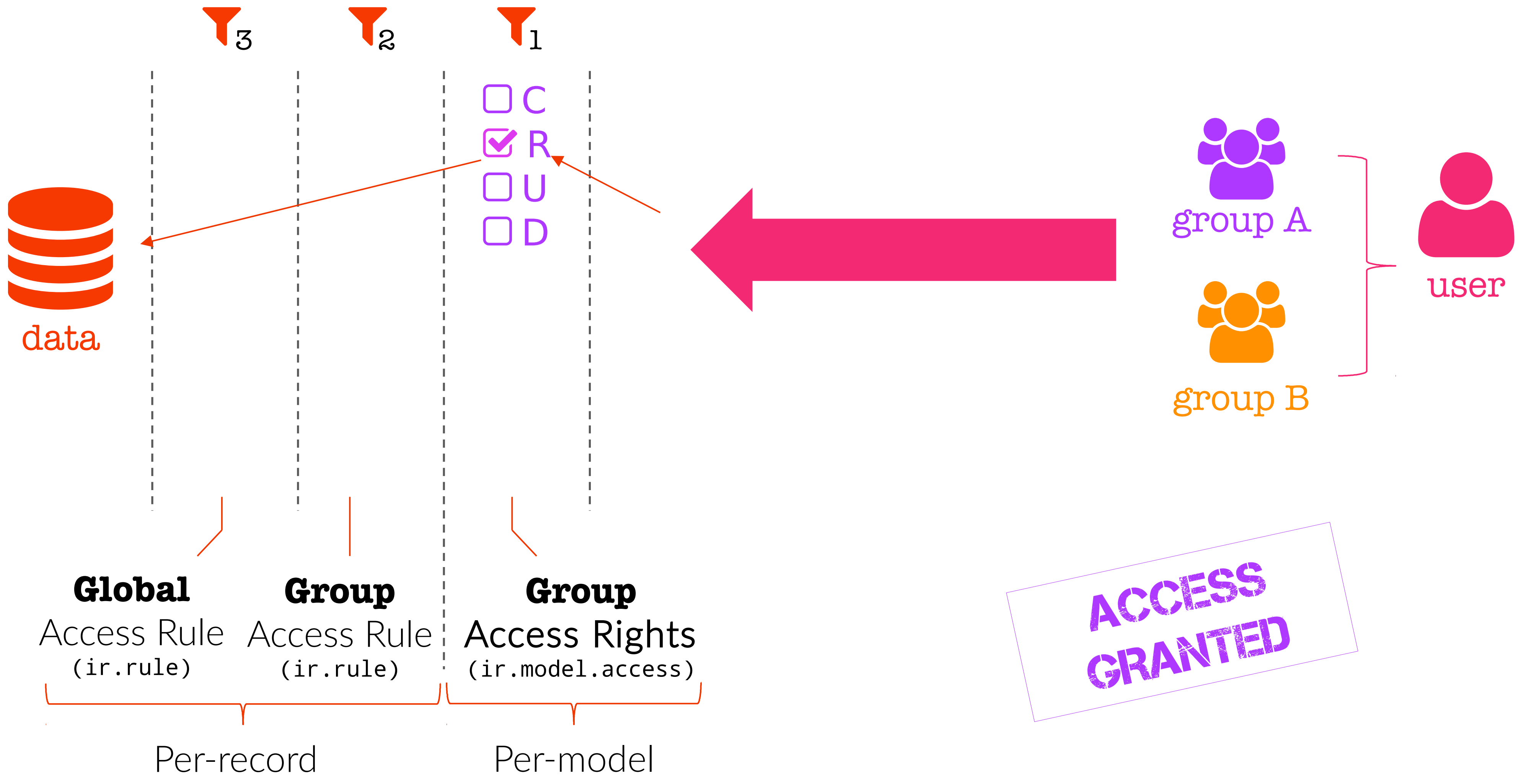
Odoo ACLs are not always trivial



Odoo ACLs are not always trivial



Odoo ACLs are not always trivial



ACCESS GRANTED

Rule #9

GETATTR IS NOT YOUR FRIEND

There are better and safer alternatives

Do **NOT** do this:

```
def _get_it(self, field='partner_id'):
    return getattr(record, field)
```

By passing arbitrary field values, an attacker could gain access to dangerous methods!

Try this instead:

```
def _get_it(self, field='partner_id'):
    return record[field]
```

This will only work with valid field values

Rule #10

OPEN WITH CARE

Do NOT open(), urlopen(), requests.post(), … an arbitrary URL/Path!

Summary

1. EVAL IS EVIL
2. YOU SHALL NOT PICKLE
3. USE THE CURSOR WISELY
4. FIGHT XSS
5. GUARD PASSWORDS & TOKENS FIERCELY
6. DO NOT OVER-SUDO IT
7. CSRF TOKENS FOR WEBSITE FORMS
8. MASTER THE RULES
9. GETATTR IS NOT YOUR FRIEND
10. OPEN WITH CARE

Thank you !

security@odoo.com