Security in Odoo

Aside from manually managing access using custom code, Odoo provides two main data-driven mechanisms to manage or restrict access to data.
Both mechanisms are linked to specific users through *groups*: a user belongs to any number of groups, and security mechanisms are associated to groups, thus applying security mechamisms to users.

Access Control

Managed by the `ir.model.access` records, defines access to a whole model.
Each access control has a model to which it grants permissions, the permissions it grants and optionally a group.
Access controls are additive, for a given model a user has access all permissions granted to any of its groups: if the user belongs to one group which allows writing and another which allows deleting, they can both write and delete.
If no group is specified, the access control applies to all users, otherwise it only applies to the members of the given group.
Available permissions are creation (`perm_create`), searching and reading (`perm_read`), updating existing records (`perm_write`) and deleting existing records (`perm_unlink`)

Record Rules

Record rules are conditions that records must satisfy for an operation (create, read, update or delete) to be allowed. It is applied record-by-record after access control has been applied.
A record rule has:

- a model on which it applies
- a set of permissions to which it applies (e.g. if `perm_read` is set, the rule will only be checked when reading a record)
- a set of user groups to which the rule applies, if no group is specified the rule is *global*
- a domain (orm.html#reference-orm-domains) used to check whether a given record matches the rule (and is accessible) or does not (and is not accessible). The domain is evaluated with two variables in context: `user` is the current user's record and `time` is the time module (https://docs.python.org/3/library/time.html)

Global rules and group rules (rules restricted to specific groups versus groups applying to all users) are used quite differently:

Global rules are subtractive, they *must all* be matched for a record to be accessible

Group rules are additive, if *any* of them matches (and all global rules match) then the record is accessible

This means the first *group rule* restricts access, but any further *group rule* expands it, while *global rules* can only ever restrict access (or have no effect).

> **⚠ Warning**
> record rules do not apply to the Superuser account

Field Access

An ORM **`Field`** (orm.html#odoo.fields.Field) can have a **`groups`** attribute providing a list of groups (as a comma-separated string of external identifiers (../glossary.html#term-external-identifiers)).

If the current user is not in one of the listed groups, he will not have access to the field:

restricted fields are automatically removed from requested views

restricted fields are removed from **`fields_get()`** (orm.html#odoo.models.Model.fields_get) responses

attempts to (explicitly) read from or write to restricted fields results in an access error

Security Pitfalls

As a developer, it is important to understand the security mechanisms and avoid common mistakes leading to insecure code.

Unsafe Public Methods

Any public method can be executed via a RPC call (../webservices/odoo.html#webservices-odoo-calling-methods) with the chosen parameters. The methods starting with a `_` are not callable from an action button or external API.

On public methods, the record on which a method is executed and the parameters can not be trusted, ACL being only verified during CRUD operations.

```
# this method is public and its arguments can not be trusted
def action_done(self):
    if self.state == "draft" and self.user_has_groups('base.manager'):
        self._set_state("done")

# this method is private and can only be called from other python methods
def _set_state(self, new_state):
    self.sudo().write({"state": new_state})
```

Making a method private is obviously not enough and care must be taken to use it properly.

Bypassing the ORM

You should never use the database cursor directly when the ORM can do the same thing! By doing so you are bypassing all the ORM features, possibly the automated behaviours like translations, invalidation of fields, **`active`**, access rights and so on.

And chances are that you are also making the code harder to read and probably less secure.

```
# very very wrong
self.env.cr.execute('SELECT id FROM auction_lots WHERE auction_id in (' + ','.join(map(s
auction_lots_ids = [x[0] for x in self.env.cr.fetchall()]

# no injection, but still wrong
self.env.cr.execute('SELECT id FROM auction_lots WHERE auction_id in %s '\
           'AND state=%s AND obj_price > 0', (tuple(ids), 'draft',))
auction_lots_ids = [x[0] for x in self.env.cr.fetchall()]

# better
auction_lots_ids = self.search([('auction_id','in',ids), ('state','=','draft'), ('obj_pr
```

SQL injections

Care must be taken not to introduce SQL injections vulnerabilities when using manual SQL queries. The vulnerability is present when user input is either incorrectly filtered or badly quoted, allowing an attacker to introduce undesirable clauses to a SQL query (such as circumventing filters or executing **`UPDATE`** or **`DELETE`** commands).

The best way to be safe is to never, NEVER use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string.

The second reason, which is almost as important, is that it is the job of the database abstraction layer (psycopg2) to decide how to format query parameters, not your job! For example psycopg2 knows that when you pass a list of values it needs to format them as a comma-separated list, enclosed in parentheses !

```
# the following is very bad:
#   - it's a SQL injection vulnerability
#   - it's unreadable
#   - it's not your job to format the list of ids
self.env.cr.execute('SELECT distinct child_id FROM account_account_consol_rel ' +
           'WHERE parent_id IN ('+','.join(map(str, ids))+')')

# better
self.env.cr.execute('SELECT DISTINCT child_id '\
           'FROM account_account_consol_rel '\
           'WHERE parent_id IN %s',
           (tuple(ids),))
```

This is very important, so please be careful also when refactoring, and most importantly do not copy these patterns!
Here is a memorable example to help you remember what the issue is about (but do not copy the code there). Before continuing, please be sure to read the online documentation of pyscopg2 to learn of to use it properly:

- The problem with query parameters (http://initd.org/psycopg/docs/usage.html#the-problem-with-the-query-parameters)
- How to pass parameters with psycopg2 (http://initd.org/psycopg/docs/usage.html#passing-parameters-to-sql-queries)
- Advanced parameter types (http://initd.org/psycopg/docs/usage.html#adaptation-of-python-values-to-sql-types)
- Psycopg documentation (https://www.psycopg.org/docs/sql.html)

Unescaped field content

When rendering content using JavaScript and XML, one may be tempted to use a **`t-raw`** to display rich-text content. This should be avoided as a frequent XSS (https://en.wikipedia.org/wiki/Cross-site_scripting) vector.
It is very hard to control the integrity of the data from the computation until the final integration in the browser DOM. A **`t-raw`** that is correctly escaped at the time of introduction may no longer be safe at the next bugfix or refactoring.

```
QWeb.render('insecure_template', {
    info_message: "You have an <strong>important</strong> notification",
})


<div t-name="insecure_template">
    <div id="information-bar"><t t-raw="info_message" /></div>
</div>
```

The above code may feel safe as the message content is controlled but is a bad practice that may lead to unexpected security vulnerabilities once this code evolves in the future.

```
// XSS possible with unescaped user provided content !
QWeb.render('insecure_template', {
    info_message: "You have an <strong>important</strong> notification on " \
        + "the product <strong>" + product.name + "</strong>",
})
```

While formatting the template differently would prevent such vulnerabilities.

```
QWeb.render('secure_template', {
    message: "You have an important notification on the product:",
    subject: product.name
})
```

```
<div t-name="secure_template">
    <div id="information-bar">
        <div class="info"><t t-esc="message" /></div>
        <div class="subject"><t t-esc="subject" /></div>
    </div>
</div>


.subject {
    font-weight: bold;
}
```

Evaluating content

Some may want to `eval` to parse user provided content. Using `eval` should be avoided at all cost. A safer, sandboxed, method `safe_eval` can be used instead but still gives tremendous capabilities to the user running it and must be reserved for trusted privileged users only as it breaks the barrier between code and data.

```
# very bad
domain = eval(self.filter_domain)
return self.search(domain)

# better but still not recommended
from odoo.tools import safe_eval
domain = safe_eval(self.filter_domain)
return self.search(domain)

# good
from ast import literal_eval
domain = literal_eval(self.filter_domain)
return self.search(domain)
```

Parsing content does not need `eval`

Language	Data type	Suitable parser
Python	int, float, etc.	int(), float()
Javascript	int, float, etc.	parseInt(), parseFloat()
Python	dict	json.loads(), ast.literal_eval()
Javascript	object, list, etc.	JSON.parse()

Accessing object attributes

If the values of a record needs to be retrieved or modified dynamically, one may want to use the `getattr` and `setattr` methods.

```
# unsafe retrieval of a field value
def _get_state_value(self, res_id, state_field):
    record = self.sudo().browse(res_id)
    return getattr(record, state_field, False)
```

This code is however not safe as it allows to access any property of the record, including private attributes or methods.
The `__getitem__` of a recordset has been defined and accessing a dynamic field value can be easily achieved safely:

```
# better retrieval of a field value
def _get_state_value(self, res_id, state_field):
    record = self.sudo().browse(res_id)
    return record[state_field]
```

The above method is obviously still too optimistic and additional verifications on the record id and field value must be done.