谈一谈ORM的安全

说到ORM安全，很多人能想到的仅仅就是ORM会提供一些直接执行SQL语句的接口。当然不止，我把ORM的SQL注入漏洞大致分为4个等级。

一、ORM中直接提供的Raw SQL方法。

这个不多说了，基本所有ORM都会提供原生的SQL语句执行过程，如果这个SQL语句被控制，当然就是注入点。这个等级的，如Laravel的ORM（Eloquent）的DB::raw方法。

二、ORM中提供支持参数化查询的Raw SQL方法

这个就相对安全一点，通常类似

```
DB::query('select * from `user` where `id`=? or `username`=?',
[$_GET['id'], $_GET['username']])
```

因为内部使用参数化查询的方法来执行具体SQL语句，所以上述代码是不能注入的。

但万事也无绝对，以前很多同学问我是不是使用了参数化查询就可以杜绝SQL注入了，答案当然是否定的。举个小例子，绑定参数的位置其实是有限制的，诸如order by xxx、select xxx等位置是不能绑定参数的。

另外，由于参数化查询方法不够灵活，所以有些ORM的部分操作是不经由ORM的。举个小例子，由于insert..into语句中程序不知道values的具体参数个数，所以有部分ORM并没有在insert语句中使用ORM，结果就是导致注入。案例：Edusoho一处SQL注入漏洞（demo站演示）

这个等级的ORM方法，有如Laravel的ORM（Eloquent）的whereRaw方法：

```
DB::table('users')
            ->whereExists(function ($query) {
                $query->select(DB::raw(1))
                      ->from('orders')
                      ->whereRaw('orders.user_id = users.id');
            })
            ->get();
```

我们可以对其进行参数绑定，如：`->whereRaw("(CONCAT(clans.name,' ',ibf_members.members_display_name) like '%?%')", $searchString);`。类似的还有Django的raw方法：`Person.objects.raw('SELECT * FROM myapp_person')`，它也是支持参数绑定的：

Passing parameters into `raw()`

If you need to perform parameterized queries, you can use the **`params`** argument to **`raw()`**:

```
>>> lname = 'Doe'
>>> Person.objects.raw('SELECT * FROM myapp_person WHERE last_name = %s',
[lname])
```

对于这类支持参数绑定的ORM方法，挖掘起来就是两个原则：

1. 找不支持绑定的位置（如select xxx）
2. 找没有进行绑定的拼接（如insert into .. values xxx）

三、ORM自带方法存在SQL拼接过程

这个是ORM问题里最突出的一个。随便说个例子，老的CodeIgniter、ThinkPHP都是没有过滤`order_by()`函数的，也就是说，一旦`order_by`函数的参数被控制，将导致一个order by注入漏洞。

上面例子中这种案例数不胜数，我就不展开说了，具体可以自己慢慢发掘。另一个很典型的是，很多ORM会混搭Raw SQL和安全的方法，典型案例就是ThinkPHP的where方法（案例：WooYun-2014-88251）：

code 区域

字符串方式

字符串方式条件即以字符串的方式将条件作为 where() 方法的参数，例子：

```
$Dao = M("User");
$List = $Dao->where('uid<10 AND email="Jack@**.**.**.**"')->find();
```

实际执行的 SQL 为：

```
SELECT * FROM user WHERE uid<10 AND email="Jack@**.**.**.**" LIMIT 1
```

字符串方式设定的条件即为实际 SQL 执行的条件，也是最接近原生 SQL 的方式，ThinkPHP 不会对条件做任何（类型上的）检查。

where方法可以传入（键，值）对进行查询，也可以传入Raw SQL，这样的混搭通常会造成极大的安全问题。ThinkPHP中另一个混搭，就是exp的问题，详情是这个众所周知的漏洞《ThinkPHP框架架构上存在SQL注入》。

Laravel的混搭：

Specifying A Select Clause

Of course, you may not always want to select all columns from a database table. Using the `select` method, you can specify a custom `select` clause for the query:

```
$users = DB::table('users')->select('name', 'email as user_email')->get();
```

Django中也存在此类问题，Django ORM正常使用是不会存在SQL拼接的，但如果你在ORM中混搭了extra函数，就可能有SQL注入了：

```
q = Entry.objects.extra(select={'is_recent': "pub_date > '%s'"
% request.GET.get('pub_date')})
q = q.extra(order_by = ['-is_recent'])
```

CodeIgniter的query builder其实也存在这个问题，我就不具体说了，给个图大家自己去找找看：

你应该注意到 $this->db->where() 方法的使用，它可以为你设置 WHERE 子句。 你也可以直接使用字符串形式设置 WHERE 子句：

```
$this->db->update('mytable', $data, "id = 4");
```

这个等级的ORM方法，举几个例子就是上述说到的ThinkPHP/CodeIgniter/Django。

四、ORM中谜一般的“符号型”注入方法

第四种就比较玄妙，最早的起源可能和nosql有关，说具体点就是Mongodb。大家可以先看看Mongodb的注入，这篇文章《Mongodb注入攻击》，顺便想念一下作者和发布这篇文章的平台。

Mongodb的注入核心就在于，用户控制了查询过程中的“符号”，比如Mongodb中 `$eq` 代表“等于”符号，而 `$ne` 就代表“不等于”。默认的符号是 `$eq` ，而如果将符号指定为 `$ne` ，就颠覆、反转了默认查询的逻辑条件，造成注入攻击。

这类注入攻击通常很难被发现，因为它不包含任何SQL语句拼接的过程，有的只是对于逻辑的修改。在传统数据库的ORM里，这种攻击也存在。比如ThinkPHP的这个案例《ThinkPHP架构设计不合理极易导致SQL注入》。

ThinkPHP也是支持传入eq/neq/gt...等操作符来控制SQL查询的条件的：

此时的攻击利用了php可以传递数组参数的一个特性。

当传入的url为：

http://127.0.0.1/2.php?username=test&password=test

执行了语句：

```
db.test.find({username:'test',password:'test'});
```

如果此时传入的url如下:

```
http://127.0.0.1/2.php?username[xx]=test&password=test
```

则$username就是一个数组，也就相当于执行了php语句：

```
$data = array(
'username'=>array('xx'=>'test'),
'password'=>'test');
```

而mongodb对于多维数组的解析使最终执行了如下语句：

```
db.test.find({username:{'xx':'test'},password:'test'});
```

利用此特性，我们可以传入数据，是数组的键名为一个操作符（大于，小于，等于，不等于等等），完成一些攻击者预期的查询。

这和mongodb的注入很类似，我就不多说了。有一些同学可能会将《ThinkPHP框架架构上存在SQL注入》和这个漏洞混淆在一块，其实仔细看，他们是两个不同的漏洞：前者是存在SQL语句拼接过程的，注入也存在于拼接过程中；而后者不存在SQL语句的拼接，注入漏洞完全是由引入操作符导致的逻辑错误造成的。

Django的ORM也是一个将符号写作字符串(gt/lt/gte/lte/in/exact)的案例，甚至加入了很多功能更为强大的“符号”（contains/regex/startswith/endswith/range），如果你能控制查询的“键”，也将可以使用这些“符号”来进行注入。

我在Pwnhub上出的Django注入的题目就是使用了这个思路。用户控制了Django ORM中的filter方法的参数，导致可以引入一系列的符号与查询方法：

```
class LoginView(JsonResponseMixin, generic.TemplateView):
    template_name = 'login.html'

    def post(self, request, *args, **kwargs):
        data = json.loads(request.body.decode())
        stu = models.Student.objects.filter(**data).first()
        if not stu or stu.passkey != data['passkey']:
            return self._jsondata('账号或密码错误', 403)
        else:
            request.session['is_login'] = True
            return self._jsondata('登录成功', 200)
```

题目的标准解是通过使用 `group__secret__regex='pwnhub{flag:.*}'` 来进行盲注，使用的是Django提供的两个功能：1是关联表的查询 2是regex正则匹配方法。

具体题目的细节与思路，之后请关注Pwnhub的官方解答与我的博客。

控制符号的难点与我欣赏的做法

其实在ORM中，如何控制Where语句符号，其实算一个比较难的问题。就如上述的ThinkPHP和Django，如果开发者一不小心就可能导致错误。

但ThinkPHP和Django的符号控制难度又有差别，ThinkPHP的符号是存放在“值”中的，因为“值”是很容易被用户控制的，所以被注入的概率大大上升；Django的符号是放在“键”中，所以基本只要没有很脑残的写出 `.filter(**data)` 这样的代码，是不会有注入漏洞的。

类似Django的还有CodeIgniter的where方法，它也是将符号存放在“键”中：

2. **自定义 key/value 方式:**

为了控制比较，你可以在第一个参数中包含一个比较运算符：

```
$this->db->where('name !=', $name);
$this->db->where('id <', $id); // Produces: WHERE name != 'Joe' AND id < 45
```

Laravel是将符号存放在where的第二个参数：

Of course, you may use a variety of other operators when writing a `where` clause:

```
$users = DB::table('users')
                ->where('votes', '>=', 100)
                ->get();

$users = DB::table('users')
                ->where('votes', '<>', 100)
                ->get();

$users = DB::table('users')
                ->where('name', 'like', 'T%')
                ->get();
```

You may also pass an array of conditions to the `where` function:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

但他做的更好的一点是将一些符号直接作为函数名了，如between和in（相比起来，Django是将in也作为了一个“操作符”看待）：

whereIn / whereNotIn

The `whereIn` method verifies that a given column's value is contained within the given array:

```
$users = DB::table('users')
                    ->whereIn('id', [1, 2, 3])
                    ->get();
```

The `whereNotIn` method verifies that the given column's value is **not** contained in the given array:

```
$users = DB::table('users')
                    ->whereNotIn('id', [1, 2, 3])
                    ->get();
```

其实我最为欣赏的，还是Python的ORM ——— Sqlalchemy。Sqlalchemy重写了Python原生的操作符，我们可以直接像写Python运算一样写SQL语句，不再用担心符号被用户控制的问题：

Most Python operators, as it turns out, produce a SQL expression here, like equals, not equals, etc.:

```
>>> print(users.c.id != 7)
users.id != :id_1

>>> # None converts to IS NULL
>>> print(users.c.name == None)
users.name IS NULL

>>> # reverse works too
>>> print('fred' > users.c.name)
users.name < :name_1
```