

Django下防御Race Condition漏洞

原创 phith0n 代码审计 2023-03-19 14:08 发表于新加坡

收录于合集

#django 1 #条件竞争 1

今天下午在v2ex上看到一个帖子，讲述自己因为忘记加分布式锁导致了公司的损失：



我曾在《从Pwnhub诞生聊Django安全编码》一文中描述过关于商城逻辑所涉及的安全问题，其中就包含并发漏洞（Race Condition）的防御，但当时说的比较简洁，也没有演示实际的攻击过程与危害。今天就以v2ex上这个帖子的场景来讲讲，常见的存在漏洞的Django代码，与我们如何正确防御竞争漏洞的方法。

0x01 Playground搭建

首先，使用我这个[Django-Cookiecutter](#)脚手架创建一个项目，项目名是Race Condition Playground。

创建两个新的Model：

```

class User(AbstractUser):
    username = models.CharField('username', max_length=256)
    email = models.EmailField('email', blank=True, unique=True)
    money = models.IntegerField('money', default=0)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['username']

    class Meta(AbstractUser.Meta):
        swappable = 'AUTH_USER_MODEL'
        verbose_name = 'user'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.username

class WithdrawLog(models.Model):
    user = models.ForeignKey('User', verbose_name='user', on_delete=models.SET_NULL, null=True)
    amount = models.IntegerField('amount')

    created_time = models.DateTimeField('created time', auto_now_add=True)
    last_modify_time = models.DateTimeField('last modify time', auto_now=True)

    class Meta:
        verbose_name = 'withdraw log'
        verbose_name_plural = 'withdraw logs'

    def __str__(self):
        return str(self.created_time)

```

一个是User表，用以储存用户，其中money字段是这个用户的余额；一个是WithdrawLog表，用以储存提取的日志。我们假设公司财务会根据这个日志表来向用户打款，那么只要成功在这个表中插入记录，则说明攻击成功。

然后，我们编写一个WithdrawForm，其字段amount，表示用户此时想提取的余额，必须是整数：

```

class WithdrawForm(forms.Form):
    amount = forms.IntegerField(min_value=1)

    def __init__(self, *args, **kwargs):
        self.user = kwargs.pop('user', None)
        super().__init__(*args, **kwargs)

    def clean_amount(self):
        amount = self.cleaned_data['amount']
        if amount > self.user.money:
            raise forms.ValidationError('insufficient user balance')

        return amount

```

我将检查用户余额的逻辑放在 `WithdrawForm.clean_amount` 中，如果发现用户要提取的金额大于用户的余额，则抛出一个 `forms.ValidationError` 异常。

最后我们编写一个用于提现的View：

```
class BaseWithdrawView(LoginRequiredMixin, generic.FormView):
    template_name = 'form.html'
    form_class = forms.WithdrawForm

    def get_form_kwargs(self):
        kwargs = super().get_form_kwargs()
        kwargs['user'] = self.request.user
        return kwargs

class WithdrawView1(BaseWithdrawView):
    success_url = reverse_lazy('ucenter:withdraw1')

    def form_valid(self, form):
        amount = form.cleaned_data['amount']
        self.request.user.money -= amount
        self.request.user.save()
        models.WithdrawLog.objects.create(user=self.request.user, amount=amount)

        return redirect(self.get_success_url())
```

这个 `WithdrawView1` 非常简单，因为使用了django的 `generic.FormView`，所以Django在接收到POST请求后会正常使用form的方法进行检查（包含上面提到的余额充足的检查），检查通过后执行 `form_valid()` 函数完成提现操作：

- 对 `request.user.money` 进行自减
- 在 `WithdrawLog` 中添加一条新记录

最后再添加一些必要的前端、路由、Admin等即可完工。

跑起来Web应用，访问后台，给自己的账户设置余额为10：

The screenshot shows the Django administration interface. On the left is a sidebar with navigation links: 'Home', 'User Center', 'User', and 'phith0n'. Below these are sections for 'AUTHENTICATION AND AUTHORIZATION' (Groups), 'USER CENTER' (User, Withdraw logs), and 'Permissions'. The main content area is titled 'Change user' for the user 'phith0n'. It contains fields for 'Username' (phith0n), 'Email' (phith0n.ph2f@gmail.com), and 'Password' (algorithm: pbkdf2_sha256 iterations: 390000 salt: HmAddQ***** hash: LOXbdm*****). A red box highlights the 'Money' field, which contains the value '10'. Below the 'Money' field is a 'Permissions' section.

然后来到前台，输入Amount即可进行提现。我们尝试输入100提交，此时会因为余额不足而报错：

Withdraw money

Username

Username
phith0n

Money
10

Amount
100

insufficient user balance

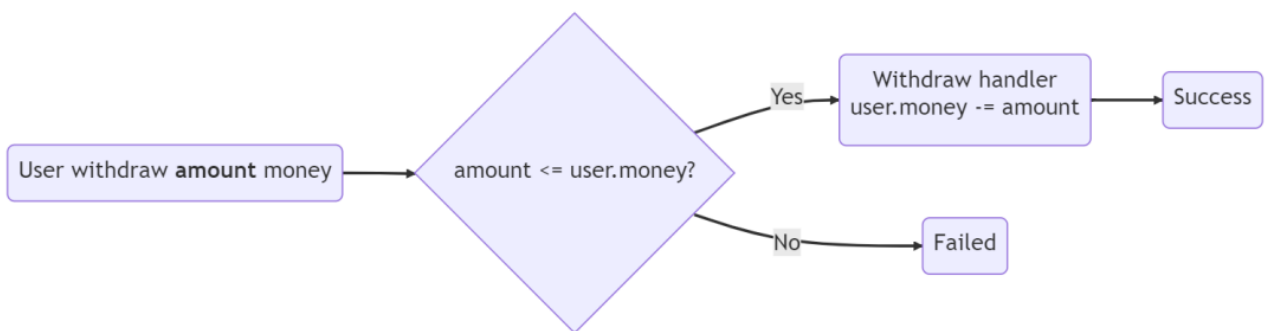
提交

运行正常，我们可以开始进行实验。

0x02 无锁无事务时的竞争攻击

观察我前面写的 `WithdrawView1`，可以发现整个操作即没有使用事务，也没有加锁，理论上是存在Race Condition漏洞的。

Race Condition的原理很简单，下图是用户提现时的流程：

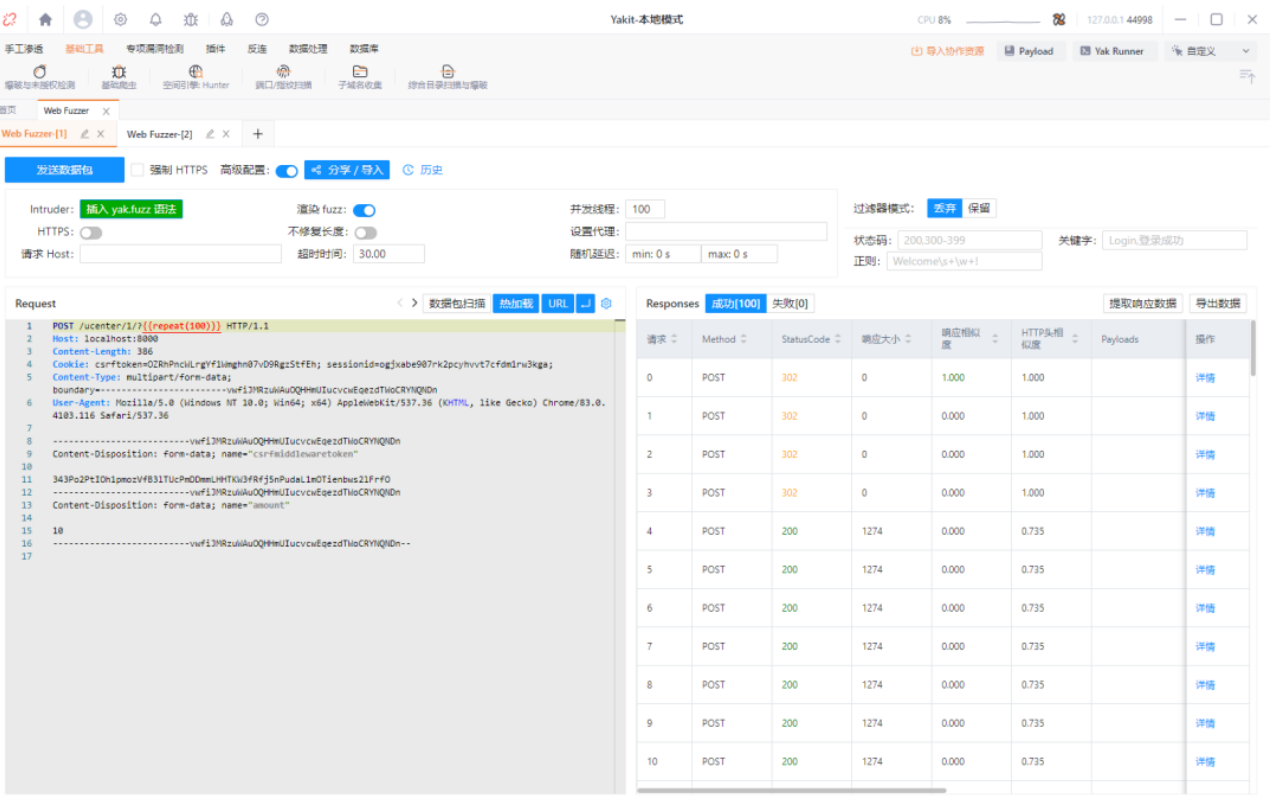


在经过 `amount <= user.money` 检查后，服务端执行提现操作，本无可厚非。但如果某个用户同时发起两次提现请求，在第一个请求经过检查到达Withdraw handler之前，此时该用户的 `user.money` 是还没有减少的；此时第二个请求如果也经过了检查，两个请求同时到达Withdraw handler，就会导致 `user.money -= amount` 执行两次。

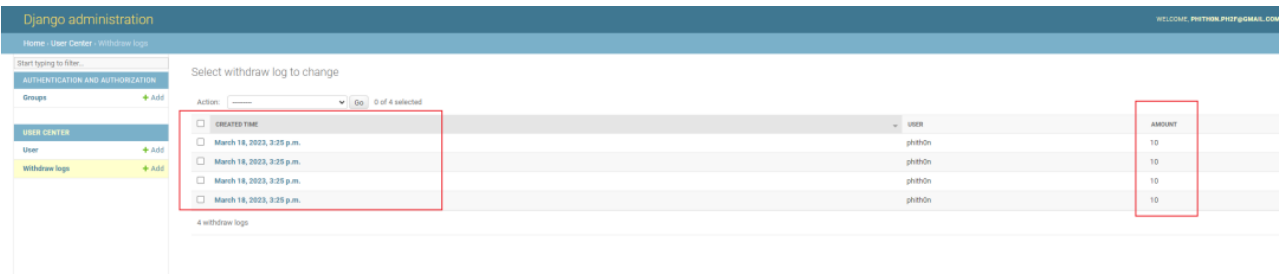
那么如果用户的余额只够提取一次，这里执行了两次，就实现了竞争攻击，造成了资产损失的结果。

测试方法也很简单，我们下载Yakit，新建一个Web Fuzzer，贴入提现的数据包。这里，我账户余额是10，我设置的提现金额amount也为10。正常情况下，我只能提现一次，第二次就会因为余额不足而失败。

然后我在数据包中添加 `{{repeat(100)}}`，并把并发线程调高到100发送，此时Yakit就会使用100个线程重复发送100次这个数据包：



见上图，发送结果里，前4个请求返回了302跳转，说明提现成功。我们来到后台Withdraw Log页面，有4个提现日志：



这意味着，我余额虽然只有10，但我成功提现了4次，也就是40元，造成的资产损失为30元。

0x03 无锁有事务时的竞争攻击

很多Django初学者会认为，这种情况只要我们加上事务就可以解决了。

原因也比较有趣，Django里增加事务的操作名字叫 `transaction.atomic`，atomic嘛就是“原子”的意思，原子操作不是可以解决并发问题吗？

我们也可以来做试验，新编写一个 `WithdrawView2`，加上 `@transaction.atomic` 修饰符：

```
class WithdrawView2(BaseWithdrawView):
    success_url = reverse_lazy('ucenter:withdraw2')

    @transaction.atomic
    def form_valid(self, form):
        amount = form.cleaned_data['amount']
        self.request.user.money -= amount
        self.request.user.save()
        models.WithdrawLog.objects.create(user=self.request.user, amount=amount)

    return redirect(self.get_success_url())
```

同样使用Yakit做测试，结果和刚才并无区别：

Request	Responses
1 POST /ucenter/2/?[repeat(100)] HTTP/1.1 Host: localhost:8080 Content-Length: 386 Cookie: csrftoken=OZRhPncILrgVflmgHn07vD9RgzStFeh; sessionId=ogjxabe907rk2pcy/hvvt7cfdmiru3kga; Content-Type: multipart/form-data; boundary=-----vuf13JRzuluAuQ9HmU1ucvucEeqzdTuoCRyHQIDn User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36	请求 Method Status Code 响应大小 响应相似度 HTTP头相似度 Payloads 操作
2	0 POST 302 0 1.000 1.000 详情
3	1 POST 302 0 0.000 1.000 详情
4	2 POST 302 0 0.000 1.000 详情
5	3 POST 302 0 0.000 1.000 详情
6	4 POST 200 1274 0.000 0.733 详情
7	5 POST 200 1274 0.000 0.733 详情
8	6 POST 200 1274 0.000 0.733 详情
9	7 POST 200 1274 0.000 0.733 详情
10	8 POST 200 1274 0.000 0.733 详情
11	9 POST 200 1274 0.000 0.733 详情
12	10 POST 200 1274 0.000 0.733 详情

后台也是4条提现记录：

CREATED TIME	USER	AMOUNT
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phithon	10

这也可以说明， `transaction.atomic` 并无处理并发的能力，只是保证当前上下文中的数据库操作在出错的时候能够回滚。

0x04 悲观锁加事务防御Race Condition

Django在ORM里提供了对数据库Select for Update的支持，在PostgreSQL、Mysql、Oracle三个数据库中都可以使用，结合Where语句，可以实现行级的锁。

使用 `SELECT FOR UPDATE` 获取到的数据库记录，不会再被其他事务获取。比如，我查询 `id = 1` 的用户，在提交事务前，其他事务执行同一个SQL语句就会block：

```
START transaction;
SELECT * FROM user WHERE id = 1 FOR UPDATE;
COMMIT;
```

这样就可以保证我们在同一个事务内执行的操作的原子性，这是一个典型的悲观锁。“悲观锁”的意思是，我们先假设其他线程会修改数据，所以在操作数据库前就加锁，直到当前线程释放锁后，其他线程才能再次获取这个锁。

我们使用 `select_for_update()` 来实现一个 `WithdrawView3`：

```
class WithdrawView3(BaseWithdrawView):
    success_url = reverse_lazy('ucenter:withdraw3')

    def get_form_kwargs(self):
        kwargs = super().get_form_kwargs()
        kwargs['user'] = self.user
        return kwargs

    @transaction.atomic
    def dispatch(self, request, *args, **kwargs):
        self.user = get_object_or_404(models.User.objects.select_for_update().all(), pk=self.request.user.pk)
        return super().dispatch(request, *args, **kwargs)

    def form_valid(self, form):
        amount = form.cleaned_data['amount']
        self.user.money -= amount
        self.user.save()
        models.WithdrawLog.objects.create(user=self.user, amount=amount)

        return redirect(self.get_success_url())
```

对于当前这个场景，我们可以再次尝试使用Yakit进行竞争攻击：

Request

<>

数据包扫描

热加载

URL

🔍

🔗

1

POST /ucenter/3/2/((repeat(100))) HTTP/1.1

2

Host: localhost:8080

3

Content-Length: 386

4

Cookie: csrftoken=Q2RhPncuLrgYf1imgn87V09Rg1zStFEn; sessionId=ogjxabe987rk2pcyhvt7cfm1rw3kga;

5

Content-Type: multipart/form-data;

6

boundary=-----vuf13HRzuAuQ@HMuIucvcuEgezdThioCRYMqIDn

7

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4183.116 Safari/537.36

8

-----vuf13HRzuAuQ@HMuIucvcuEgezdThioCRYMqIDn

9

Content-Disposition: form-data; name="csrfmiddlewaretoken"

10

343Po2PI0hIpmozVF831TucPnDmLHHTXU3FrF5SnPudL1nOTlenbus2lFrFO

11

-----vuf13HRzuAuQ@HMuIucvcuEgezdThioCRYMqIDn

12

Content-Disposition: form-data; name="amount"

13

10

-----vuf13HRzuAuQ@HMuIucvcuEgezdThioCRYMqIDn--

14

15

16

17

Responses

成功[100]

失败[0]

提取响应数据

导出数据

请求	Method	Status Code	响应大小	响应相似度	HTTP头相似度	Payloads	操作
0	POST	302	0	1.000	1.000		详情
1	POST	200	1275	0.000	0.733		详情
2	POST	200	1274	0.000	0.733		详情
3	POST	200	1274	0.000	0.733		详情
4	POST	200	1274	0.000	0.733		详情
5	POST	200	1274	0.000	0.733		详情
6	POST	200	1274	0.000	0.733		详情
7	POST	200	1274	0.000	0.733		详情
8	POST	200	1274	0.000	0.733		详情
9	POST	200	1274	0.000	0.733		详情
10	POST	200	1274	0.000	0.733		详情

可见，此时返回包只有一个302响应了，再查看后台也只成功添加一条提现记录：

Select withdraw log to change

Action: 0 of 9 selected

CREATED TIME	USER	AMOUNT
<input type="checkbox"/> March 18, 2023, 4:05 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phithon	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phithon	10

9 withdraw logs

这意味着程序是按照预期运行，没有发生Race Condition问题。

0x05 乐观锁加事务防御Race Condition

我们观察上述的 `WithdrawView3` 代码，其实会发现一个问题，如果有大量读操作的场景下，使用悲观锁会有性能问题。因为每次访问这个view都会锁住当前用户对象，此时其他要使用这个用户的场景（如查看用户主页）也会卡住。

另外，也不是所有数据库都支持 `select for update`，我们也可以尝试使用乐观锁来解决Race Condition的问题。

乐观锁的意思就是，我们不假设其他进程会修改数据，所以不加锁，而是到需要更新数据的时候，再使用数据库自身的UPDATE操作来更新数据库。因为UPDATE语句本身是原子操作，所以也可以用来防御并发问题。

我们新增一个 `WithdrawView4`：

```
class WithdrawView4(BaseWithdrawView):
    success_url = reverse_lazy('ucenter:withdraw4')

    @transaction.atomic
    def form_valid(self, form):
        amount = form.cleaned_data['amount']
        rows = models.User.objects.filter(pk=self.request.user, money__gte=amount).update(money=F('money')-amount)
        if rows > 0:
            models.WithdrawLog.objects.create(user=self.request.user, amount=amount)

        return redirect(self.get_success_url())
```

代码基本是从 `WithdrawView2` 来的，只是将其中的 `self.request.user.money -= amount` 改成了用update，并且在修改数据行数大于0的情况下再添加提现日志。

此时，假设有多个提现请求同时到达update语句，因为update本身的原子性，执行第一次update后，用户的余额已经减少amount，再执行第二次update时，`money__gte=amount` 这个条件就不会成功，就不会再次减少amount了。

使用Yakit进行测试，只有一次302返回：

Request: 数据扫描 添加URL 提取响应数据 导出数据


```
1 POST /ucenter/4/ HTTP/1.1
2 Host: localhost:8080
3 Content-Length: 386
4 Cookie: csrfToken=02RnPncVLrgVf1mgHn87vD9RgzStfEh; sessionId=ogjsabe907rk2pcyhvt7cfdm1ru3kga;
5 Content-Type: multipart/form-data;
6 boundary=-----vuf13RZuIAuQ@HmUJucvceqezdTioCRYNQiDn
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36
7
8 -----vuf13RZuIAuQ@HmUJucvceqezdTioCRYNQiDn
9 Content-Disposition: form-data; name="csrfmiddlewaretoken"
10
11 343Po2Pt1Oh1pmozVf831TUCPeDmLHMTK3fRfj5nPuDaLiMOTienbvs2lFrFO
12 -----vuf13RZuIAuQ@HmUJucvceqezdTioCRYNQiDn
13 Content-Disposition: form-data; name="amount"
14
15 10
16 -----vuf13RZuIAuQ@HmUJucvceqezdTioCRYNQiDn--
17
```

请求	Method	Status Code	响应大小	响应相似度	HTTP头相似度	Payloads	操作
0	POST	302	0	1.000	1.000		详情
1	POST	200	1275	0.000	0.733		详情
2	POST	200	1275	0.000	0.733		详情
3	POST	200	1274	0.000	0.733		详情
4	POST	200	1274	0.000	0.733		详情
5	POST	200	1274	0.000	0.733		详情
6	POST	200	1274	0.000	0.733		详情
7	POST	200	1274	0.000	0.733		详情

查看后台，也只成功添加一个提现日志，符合预期：

Select withdraw log to change

Action: Go 0 of 10 selected

CREATED TIME	USER	AMOUNT
<input checked="" type="checkbox"/> March 18, 2023, 3:23 p.m.	phith0n	10
<input type="checkbox"/> March 18, 2023, 4:05 p.m.	phith0n	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phith0n	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phith0n	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phith0n	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phith0n	10
<input type="checkbox"/> March 18, 2023, 3:54 p.m.	phith0n	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phith0n	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phith0n	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phith0n	10
<input type="checkbox"/> March 18, 2023, 3:25 p.m.	phith0n	10

10 withdraw logs

乐观锁的优点就是不会锁住数据库记录，也就不会影响其他线程查询该用户。

0x06 总结

本文主要从v2ex一个帖子的例子入手，阐述了如何使用Yakit进行Race Condition攻击，以及在Django中如何使用悲观锁和乐观锁对该攻击进行防御。

本文涉及的代码，可以在<https://github.com/phith0n/race-condition-playground>找到。