FIXES

For fixing Gifting a card attack and the XSS attack, we had sanitized the "director" variable within buy.html in views.py. We limited the variable to be at most 12 characters long and check if it is comprised of only alphabetic characters.

```
65
66 def buy_card_view(request, prod_num=0):
67     if request.method == 'GET':
68         context = {"prod_num" : prod_num}
69         director = request.GET.get('director', None)
70         if director is not None:
71             if director.isalpha() and len(director) < 12:
72                 # KG: Wait, what is this used for? Need to check the template.
73                 context['director'] = director
74         if prod_num != 0:
75             try:
76                 prod = Product.objects.get(product_id=prod_num)
77             except:
```

To fix the SQL injection where we get a user's hashed password, we also sanitize the inputs for the query in views.py. If the signature is not alphanumeric, it could contain sql or scripting elements, so we automatically raise an exception to prevent any attacks.

```
208         # KG: data seems dangerous.
209         print(card_data.strip())
210         signature = json.loads(card_data)['records'][0]['signature']
211         # signatures should be pretty unique, right?
212         if not signature.isalnum():
213             card_query = Card.object.raw('')
214             user_cards = Card.objects.raw('')
215             raise exception("Invalid signature")
216         else:
217             card_query = Card.objects.raw('select id from LegacySite_card where data LIKE \'%%%s%%\'' % signature)
218             user_cards = Card.objects.raw('select id, count(*) as count from LegacySite_card where LegacySite_card.user_id
    = %s' % str(request.user.id))
```

Finally, to fix the exploit regarding the faulty salts, we should use a randomized salt per password and store that salt alongside the password, therefore each hashed salt and password combination would be unique.