Kevin Zhang
Application Security CS GY 9163

Homework 2: When a Wreck Reaches the World Wide Web

On NYU Classes, submit a link to your GitHub repository. The repository should be **private**, and you should add the instructor/TA's GitHub account as a contributor to give them access for grading. For this section, your instructor is: **Kevin Gallagher**, GitHub ID **`kcg295`**. Your TA is: **Evan Richter**, GitHub ID **`evanrichter`**

Author note: Please note that all comments to code created by me are indicated by **#KZ:/ or <!-KZ:--->.** This is to assist in grading and locating changed parts of the code.

Part 0: Setting up Your Environment

Setting up Django

- Used pip to install Django 3.2.2
- Installed Python 3.9.5
- Utilized Virtual box

Some additional tools that may be useful for this assignment (but are not necessary) are **sqlite**, **burp suite**, the **python requests library**, and the **web development console** of your favorite browser. **If you are runing a *NIX system, these tools should be pre-installed and/or available in your distribution's package manager**. **However, we will be checking for signed commits, since they are security relevant**.

Part 1: Auditing and Test Cases

Files added to repository

- "GiftcardSite/"
- "LegacySite/"
- "Images/"
- "Templates/"
- "Manage.py"

Attacks were created for each case to exploit vulnerabilities throughout the website. Please see attached bugs.txt to demonstrate the 1) bug exploited, and 2) a description and fix for the bug.

1. ***One* attack, that exploits a XSS (cross-site scripting) vulnerability.**
    a. Bug: Identifies a safe string ("director") in gift.html, allowing XSS attacks
    b. Description: In our XSS attack, we take advantage of the fact that we can inject HTML into a GET Request. If we utilize a script as part of the director, we can engage in an XSS attack. We utilize curl within the .sh file in order to run a very small script. Specifically, we target the file views.py located in the LegacySite folder. The "gift" is referencing the gift_card_view function. Our goal is to have the target browser session run a script with our payload. In the XSS attack, we utilize the unencoded data to the browser, allowing us to inject scripts or processes. In order to patch

this vulnerability, we located the issue at Line 60-62 of gift.html - if we take away the "safe tag," we are able to trigger Django to automatically check the variable against API.

2. ***One* attack that allows you to force another user to gift a gift card to your account without their knowledge.**
    a. Bug: Gift form does not validate crsf, post requests go unchallenged
    b. Description: To force a user to gift a gift card, we utilize a Cross-Site Request Forgery (CRSF) based attack, which will force the target to execute code against their will. In this case, we will be taking advantage of sending a POST message to the webpage. For the CRSF attack, we target the gift section of the Legacy Site (sepcifically views.gift_card_view). Our goal is that by using an HTML file with the "form" section included as part of the code, we post a request to the target website, while masked by a normal HTML file. We forge the POST request and create a value for a gift card ($5000) to give to our account - this creates a dangerous precedent where someone can just generate valuable gift cards. To fix this, we need to patch the XSS vulnerability in the code - this is accomplished by removing the safe tag in Attack #1.
3. ***One* attack that allows you to obtain the salted password for a user given their username. The database should contain a user named ``admin.''**
    a. Bug: Views.py does not handle signature values well - we utilize a union attack to force the system to reveal the admin password (which should be securely stored, but more on that in part 3).
    b. Description: Analyzing "extras.py" within the LegacySite folder, we can identify the fields that are key to reading gift cards. We can identify the fields "merchant_id", "customer_id", "total_value", "records" as key to the operation of the gift card reader. Our target is the gift card reader ourselves, and we do this through an injection attack. With this, we can attempt to exploit views.py, as the program queries a signature. By including these SQL injection commands using a "giftcard" (our payload in reality), we force the views.use_card_view file to identify the corresponding information. Our goal is to, as "Salty Customer" (no pun intended), to attack the Tuition Card system by finding the administrator account and its paired password. The union operation included within our attack will pull the "admin" user password and place it within our table, thus revealing information regarding the LegacySite's database. To fix this issue, we replaced the problem code ("card_query = ard.objects.raw('select id from LegacySite_card where data = \'%s\'' % signature)") and changed object.raw to object filter, as well as the last bit regarding the signature. We used a encoded signature, which should allow safe retrieving of values and filtering (detection and blocking) of SQL attacks.
4. ***One* attack that exploits another attack not listed above on the server.**
    a. One word/abbreviation: HTTP
    b. Description The goal of this attack is to leverage the natural vulnerabilities present in HTTP. As said in many security classes and commonly known

in the security world, HTTP is a vulnerable protocol, as it does not utilize TLS (SSL) to encrypt HTTP(S) GET requests and responses (https://www.cloudflare.com/learning/ssl/why-is-http-not-secure/#:~:text=The%20only%20difference%20between%20the,uses%20HTTPS%20has%20https%3A%2F%2F). The main differences between HTTP and HTTPS are well documented, and any of a number of attacks can affect the HTTP Django server currently in place for the gift card website. Attacks include BGP and DNS hijacking, domain spoofing, and kinds of man-in-the-middle (MITM) attacks. The easy fix for this is to implement some form of server that utilizes HTTPS. Without an HTTPS fix, an attacker could utilize all forms of vulnerabilities present in HTTP. In our program we utilize an SSL server, run using runsslserver. We can check that HTTPS is implemented by examining the URL of the web app (https://127.0.0.1:8000).

5. A text file, bugs.txt explaining the bug triggered by each of your attacks and describing any other vulnerabilities or broken functionalities you came across.

#KZ: Still need to run Travis against these fixes up above. I conducted my program testing locally on my own Django distribution, but did not have enough time to validate with Travis (conduct Travis regression testing). *To make sure that these bugs don't come up again as the code evolves, write some test cases for django that test for these vulnerabilites. Then have Travis run these tests with each push.*

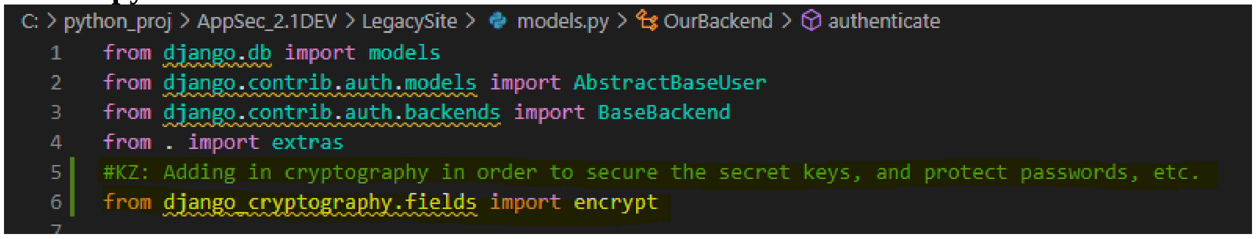
After finalizing writing and testing the above programs, I have marked this commit with the tag “Part_1_complete.”

Part 2: Encrypting the Database

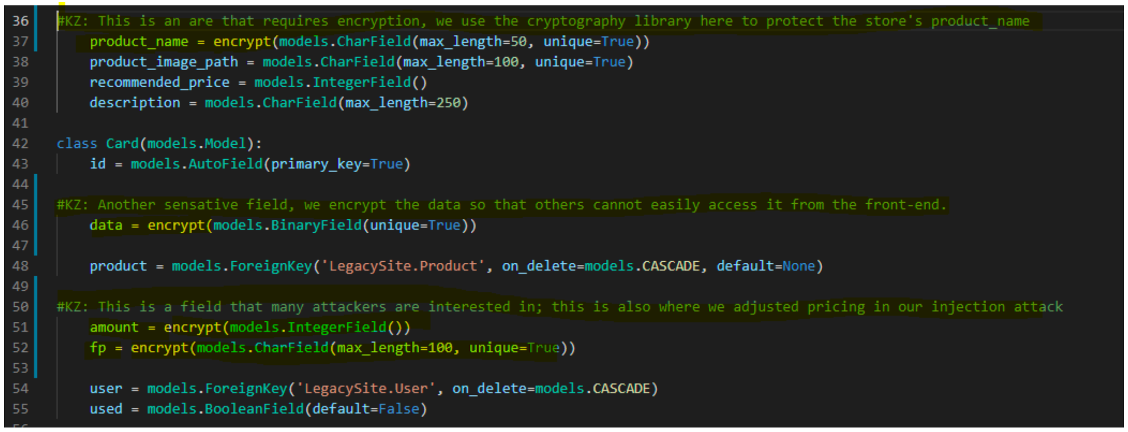
Setting up Encryption

- Requirements.txt file created to house Django libraries & plugins
- Researched and installed Django-cryptography
- Imported environ to contain Django-cryptographical keys
- Created Encryption_explanation.txt

Models.py:

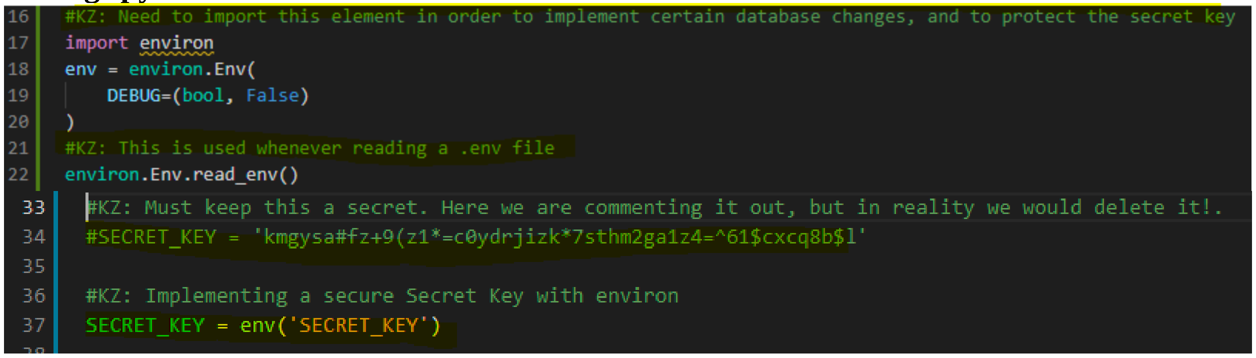


```
C: > python_proj > AppSec_2.1DEV > LegacySite > models.py > OurBackend > authenticate
from django.db import models
from django.contrib.auth.models import AbstractBaseUser
from django.contrib.auth.backends import BaseBackend
from . import extras
#KZ: Adding in cryptography in order to secure the secret keys, and protect passwords, etc.
from django_cryptography.fields import encrypt
```



```python
#KZ: This is an are that requires encryption, we use the cryptography library here to protect the store's product_name
    product_name = encrypt(models.CharField(max_length=50, unique=True))
    product_image_path = models.CharField(max_length=100, unique=True)
    recommended_price = models.IntegerField()
    description = models.CharField(max_length=250)

class Card(models.Model):
    id = models.AutoField(primary_key=True)

#KZ: Another sensative field, we encrypt the data so that others cannot easily access it from the front-end.
    data = encrypt(models.BinaryField(unique=True))

    product = models.ForeignKey('LegacySite.Product', on_delete=models.CASCADE, default=None)

#KZ: This is a field that many attackers are interested in; this is also where we adjusted pricing in our injection attack
    amount = encrypt(models.IntegerField())
    fp = encrypt(models.CharField(max_length=100, unique=True))

    user = models.ForeignKey('LegacySite.User', on_delete=models.CASCADE)
    used = models.BooleanField(default=False)
```

Settings.py



```python
#KZ: Need to import this element in order to implement certain database changes, and to protect the secret key
import environ
env = environ.Env(
    DEBUG=(bool, False)
)
#KZ: This is used whenever reading a .env file
environ.Env.read_env()
```

```python
#KZ: Must keep this a secret. Here we are commenting it out, but in reality we would delete it!.
#SECRET_KEY = 'kmgysa#fz+9(z1*=c0ydrjizk*7sthm2ga1z4=^61$cxcq8b$l'

#KZ: Implementing a secure Secret Key with environ
SECRET_KEY = env('SECRET_KEY')
```

Database encryption, Managed keys, and why you choose to manage keys that way:

Encryption of Database: Looking at the current LegacyGift and GiftcardSite, I will say there is something to be said about securing of confidential information. The fact that we were easily able to leverage a vulnerability in obtaining an administrative password speaks volumes to the type of security an application has without any form of encryption. Django itself has many options for encryption, but I decided to go with Django-cryptography. A small-scale and easy-to-implement (as easy as adding 'encrypt') library, we are given a form a bidirectionality when it comes to storing and retrieving data.

In implementing encryption, we have made changes to 'models.py' within the LegacySite and 'settings.py' within the GiftcardSite. Out of the two, 'settings.py' is far more important in that it is the program we use to implement the bulk of the encryption. Specifically, we targeted certain sensitive-data fields that I felt would be amiss if an attacker were able to gain access. One of which, for example, displays the pricing.

Secure Key Management: Securing of the secret keys is done through Django-cryptography, as the library allows us to set a default key as "SECRET_KEY." This library is coupled with the environ library, which together can store the secret key in an environment variable. The reason environ is key (no pun intended) in hiding is that without it, we are unable to store secret keys outside of key variables. In this case, we are given peace of mind if any attacker attempts to leverage settings.py for the key.

Additionally, Secure Key Management would usually implement multi-factor authentication while giving the least number of user access (least privilege). In fact, the status of the secret key is not sufficient, as we need to store the key within the same few files. If the key itself is stored separate from the code and secured by authentication (potentially nonce based), it would be much more obscure and harder to crack.

After finalizing the encryption, I have marked this commit with the tag “Part_2_complete.”