Lab 2: Overall Write-up

Part 1: Auditing and Test Cases

Attack 1: XSS (cross-site scripting vulnerability to call the javascript alert “hello”)

Attack Description

For this attack, I exploited the payload in the following files: buy.html file (that gets redirected from item-single.html file) and gift.html file. More specifically, I inspected the code in these files and I realized that the following lines contain the attribute “director” that does not get sanitized anywhere else:

Line 60 in gift.html: Endorsed by {{director|safe}}!

Line 62 in item-single.html: Endorsed by {{director|safe}}!

In other words, I used the “director” attribute since this attribute is being marked as safe. This is why I can give a script text after the director attribute in the URL. This attack is thus called a cross-site scripting attack: I am relying on the “director” and just entering my desired payload with the <script> </script> method for starting and ending the pop-up message, which is what the javascript alerts, and upon refreshing the site it reflects a pop-up box. So that is where I found a cross-scripting vulnerability. I just directly ran the alert method that is triggering the pop-up box with the message “hello” on my screen. In the URL part, I specifically wrote:

“http://127.0.0.1:8000/buy.html?director=<script>alert(“hello”)</script>” or

“http://127.0.0.1:8000/gift.html?director=<script>alert(“hello”)</script>”

So as you can observe, the payload we use is "?director=<script>alert("hello")</script>" in both buy.html and gift.html and I have written them in the xss.txt file in part1 folder of this assignment as well. After executing this payload, a pop-up alert box saying "hello" message gets generated successfully. To conclude, both item-single.html and gift.html files contain this XSS-related bug since the “director” attribute is marked as safe. Because of this, I was able to inject the desired XSS payload in the URL which pops up to the logged-in user.

Solution

In order to fix this vulnerability, I went through the code in both files: gift.html file and item-single.html file. In both these code files, I noted the lines where the “direct” attribute was used (line 60 and line 62 respectively as mentioned in the attack description above) and I removed the “safe” tag there since this is what is sanitizing the direct action group when entering the payload.

More specifically, this was the line before:

Endorsed by {{director|safe}}!

And this was the line after my modification:

Endorsed by {{director}}!

As stated before, I did the same modification in both: gift.html file and item-single.html file. To conclude, removing the safe tag prevents the site from a pop-up alert box in the “director” attribute.

Attack 2: Cross-site request forgery: allowing me to force another user to gift a gift card to your account without their knowledge

Attack Description

In order to execute this attack, we had to create a script or HTML page. The payload delivered by “xsrf.html” was employed to exploit this vulnerability. We first performed code analysis to reveal the absence of CSRF prevention measures in the code. Consequently, we were able to exploit the login and password fields to self-award a gift card with any desired value. In this example, the gift card was

successfully gifted to test2 (the attacker's designated name/account) upon executing my HTML file, confirming the effectiveness of the attack.

Solution

The current vulnerability lies in the fact that anyone can send gift credit to themselves or to any other user, and we want to prevent this from happening on our site. Since we detected the absence of CSRF prevention, we can use CSRF middleware to protect against this vulnerability. I implemented 5 changes in total using Django's built-in tag "% csrf token%". The following changes were done in their respective files as described below.

1. Line 80 and 81 in gift.html file: {% csrf_token %} //added CSRF token
2. Line 78 and 79 in item-single.html: {% csrf_token %} //added CSRF token
3. Line 52 in setting.py: 'django.middleware.csrf.CsrfViewMiddleware' //added in Middleware
4. Line 152 and 154 in view.py: commented out the GET requests actions for username and amount to disable the use of payload
5. Line 114 and 115 view.py: @csrf_protect //added decorator

Attack 3: One attack that allows you to obtain the salted password for a user given their username. The database contains a user named admin that you can use for testing the attack.

Here we performed an SQL Injection Attack. We have created a payload called sqli.gftcrd for this attack. We use this payload on the use.html page where we have an option to add a giftcard file with the name on your file. In this specific part, we uploaded the payload to get the admin password and it is successfully executed as it has a signature field with sql vulnerability because it is not being sanitized at the input time. So here I got the admin password in hash format (which is a common occurrence in the database) after running the sqli gift card. This is the salted password value that we get upon doing this attack:
0000000000000000000000000000078d2$18821d89de11ab18488fdc0a01f1ddf4d290e198b0f80cd4974fc031dc2615a3

Solution

The current vulnerability lies in the fact that we are able to inject raw SQL codelines directly to the database, and steal the administrator's password. To prevent this attack, I observed the view.py file and sanitized and filtered the signature value input. To achieve this, I performed the following modifications:

1. Line 222 in view.py: card_query = Card.objects.raw('select id from LegacySite_card where data = %s', [signature])
2. Commented out line 221

Attack 4: One attack that allows you to run arbitrary commands on the server.

For command injection, I can inject the payload in the text file and attack with an input text. So I could use any command injection gift card and I used the payload “& touch abc.txt ;”. I'm keeping the first place as a blank, and then using the parameter for the sql payload and just running that touch command so that it creates an abc.txt file and just entering the “;” at the end. I also got the Jason decoder in my terminal and on the web page as well and it would create the file “abc.txt” in the explorer. In other words, to go into more details, following the instructions.md guidance, I first investigated the giftcardreader binary and identified the parse card data() function responsible for processing the card file and path name from the upload. While progressing, I came across the message "# KG: Are you sure you want the user to control that input?" Later, attempting to inject a simple echo "hello" payload into the upload file name box on the use.html page at 127.0.0.1:8000 was unsuccessful. Upon revisiting views.py to analyze how the

entry was handled, I realized that a file with the correct extension (".gftcrd") was essential for initiating the server's operation, explaining my initial failure. Subsequent attempts also failed to give me results. Upon closer examination of commands, I noticed that every request included the term "present." To explore the possibility of creating a new file using the touch command, I reattempted command injection with " & touch abc.txt ; ". This action led me to the "JSONDecodeError at /use.html" page. Upon inspecting the directory's files, I confirmed that abc.txt had been successfully generated, underscoring the efficacy of command injection through this approach.

Solution

To prevent the above-mentioned attack from happening, I incorporated the isalnum() function and introduced an additional verification step. This new check ensures that the card fname attribute exclusively consists of letters and numbers. After implementing this fix, subsequent tests passed successfully, and the file was not generated, confirming the resolution of the injection attack. I have explicitly indicated the lines where these modifications were made to address the identified vulnerability as shown below:

in view.py file - if card_fname is None or card_fname == '' or not card_fname.isalnum(): (line 205)

in view.py file - signature = "" (starting from line 214)

```
try:
    signature = json.loads(card_data)['records'][0]['signature']
    raise "error"
except:
    pass
```

Part 2

Encryption Implementation

For the encryption implementation, after properly implementing django's djfernet library, I used the "EncryptedTextField()" function in the models.py file. More specifically, in line 45 of the models.py file, I added the following line: data = EncryptedTextField(). Using this, the card data in the database gets encrypted using this code line, and no users will be able to view or access it directly. I also just commented out the whole line of the binary field in order to give me the gift card with visibility. More specifically, this is the commented line: "data=models/BinaryField(unique.true)". So when I use this function EncryptedTextField(), this will just encrypt the gift card so no other user will be able to see the gift card. Please note that I also added the following lines at the top of the models.py file where we are importing the needed libraries:

```
from django.db import models
from fernet_fields import *
# --ADDED ADDITIONAL LIBRARY
```

Key management

Upon looking for the secret key, I spotted a line in settings.py file where it was shown in plain text. This is a very obvious vulnerability because literally anyone who can access this file can directly and clearly see the key in plain text. During our class lab, we saw how to store secret_key in an environment file. I created the environment file as .env then took the SECRET_KEY line from setting.py and pasted it to the .env file. I implemented the key's environmental configuration using Python's decoupled module because it is more secure for cross-accessing and will not get referred to.

In lines 27 in the settings.py file, I added the following code:

load_dotenv()
SECRET_KEY = config('SECRET_KEY')
So, in settings.py, I imported the “config” component of the “Decouple” module. This is used to read the secret key from the env file like: SECRET_KEY = config('SECRET_KEY')
I chose this method for two main reasons:

1. It decouples the code from the settings which is more secure
2. Storing it in a .env file ensures that this file does not show up on the Github repository for everyone to see if it is added to the .gitignore

To summarize this part, for the key management, I went to the settings.py file where there is a secret key written down directly which is not hardcoded. I just used the Django environment variable. And I created my environment, a virtual environment. And I use this key in a store file. So I just created an environment variable, and created the environment file, and entered the key there. I also made sure that whenever I am running this make legacy site, I am running into my environment. So I created my virtual environment using django and I enter the key inside of this file and I run it.

GitHub Actions & Testing

In regards to GitHub Actions, I installed the librariesin the virtual environment as shown in class, upgraded pip, installed django, requests and virtualenv, djfernet, cryptography: so basically all the libraries which we are using. And yes, I just then run the normal commands that we need to write in order to run the server. Note: I always made sure that I enter “test” instead of run server so that the GitHub Actions environment file is made in order to test the test cases.