Assignment 2 – Spring 2023

NYU - Raj Shah

GitHub Link: https://github.com/Raj-NYU/AppSec2

Part 1 – Bugs Writeup

XSS Summary:

- **Cross-site scripting (XSS)** is a type of cyber attack where an attacker injects malicious code into a website's content or script, which is then executed on the user's browser. This can lead to the theft of sensitive user data, such as login credentials or credit card information, and the distribution of malware. XSS attacks can be prevented by properly sanitizing user input and using secure coding practices.

- **Exploit**: In my investigation of the web site's security, I discovered an XSS exploit that utilized "safe tags" present in gift.html and item-single.html. These tags, designed to prevent code from being escaped, could be manipulated by attackers to perform an XSS attack. Specifically, an attacker could use the code: http://127.0.0.1:8000/gift.html?director=<script>alert("HACKED!!!")</script> to execute malicious code.

- **Fix**: To address this vulnerability, I determined that the safe tags were the root cause and could render the contents of item-single.html and gift.html files unsafe. To mitigate this issue, I removed the safe tags from the "director" tag and conducted thorough manual tests to ensure the website's normal functionality. In addition, I developed a comprehensive test that scans the server's response for the string "<script>alert("{alert_string}")</script>". This test effectively detects if the input was executed as JavaScript code and displays "XSS vulnerability detected" if a script is detected. Conversely, if the input was properly sanitized or escaped by the server, the test returns "SAFE!!!".

- **Conclusion**: Overall, these measures will help to enhance the website's security and protect it from potential XSS attacks.

Assignment 2 – Spring 2023

NYU - Raj Shah

CSRF Summary:

- **CSRF, or Cross-Site Request Forgery**, is a security vulnerability where attackers can utilize a user's credentials on one website to perform unintended actions on another website without the user's knowledge or consent. Developers can take various measures to prevent CSRF attacks, such as utilizing CSRF tokens and SameSite cookies.

- **Exploit**: During my analysis of the giftcard website's security, I discovered a potential CSRF vulnerability. To exploit this vulnerability, I created an HTML payload that could be placed within a malicious webpage. When the authenticated user visits the attacker's webpage, an HTTP request is triggered to the vulnerable site with the user's session token. The attacker can then perform actions as if they were the user, such as gifting a card to themselves. The malicious HTML code is included in the CSRF Payload folder for reference.

- **Fix**: To address this vulnerability, I implemented several measures. First, I added the "@csrf_protect" tag to the gift_card_view function in the views.py file where Professor Gallagher had commented. This tag enables CSRF protection for a specific view function and generates a token that is included in the HTML form and verified on submission, preventing CSRF attacks. Without this protection, attackers can submit malicious data to the website. Additionally, I inserted the {% csrf_token %} template tag in the gift.html file on line 78 to generate a unique CSRF token for each form submission. This token is checked against the submitted form data to ensure that they match during page loading, rejecting any requests that do not match. This measure helps to further protect the website against potential CSRF attacks.

- **Conclusion**: These measures help to enhance the security of the giftcard website and prevent potential CSRF attacks, providing greater protection to users and their sensitive information.

Assignment 2 – Spring 2023

NYU - Raj Shah

SQLi Summary:

- **SQL injection** is a type of cyber attack where an attacker injects malicious SQL commands into a web application's input field, exploiting vulnerabilities in the application's input validation process. This allows the attacker to access and manipulate sensitive information in the backend database. Proper input validation and the use of prepared statements or parameterized queries can prevent SQL injection attacks.

- **Exploit**: During my security analysis of the giftcard website, I came across a potential SQL injection vulnerability. To test this vulnerability, I created a payload called SQLi.gftcrd and submitted a giftcard through the use.html page. As a result, the website outputted the salted hash value for the admin in PNG format.

- **Fix**: To address this vulnerability, I modified the codebase by wrapping the variable signature in brackets on line 198. This simple but effective measure helps prevent the use of special characters in the query, also known as parameterization. By parameterizing the query, special characters cannot be executed, preventing potential SQL injection attacks.

- **Conclusion**: This fix helps to enhance the security of the giftcard website and prevent potential SQL injection attacks, providing greater protection to users and their sensitive information.

Assignment 2 – Spring 2023

NYU - Raj Shah

OS Command Injection Summary:

- **OS Command Injection** is a cyber-attack where an attacker exploits vulnerabilities in an application's input validation process to inject malicious operating system commands. The injected commands are executed with the same privileges as the application, allowing the attacker to perform unauthorized actions. Preventive measures include proper input validation, using parameterized queries, and limiting the application's privileges.

- During my analysis of the giftcard website, I discovered a potential vulnerability that involved the binary giftcardreader file. By thoroughly searching through the files, I found that the extras.py file contained the parse_card_data() function, which parsed the cards file and path after uploading. Additionally, a helpful hint from Kevin's comment led me to believe that there was a vulnerability present in this area. I then decided to review the views.py file to understand how entries are processed, which revealed the potential for an OS command injection attack.

- **Exploit**: To test this vulnerability, I created a "dummy" giftcard to input, which triggered an execution at the backend. I then performed an OS command injection attack by running "gift;ifconfig;ls; & echo HACKED!!! &". This would cause a segmentation fault, list (ls) the contents of a directory, and display a message (in this case, "HACKED!!!"), all of which would show up in the server's terminal.

- **Fix**: To remediate this vulnerability, I identified that the main input for injecting a command to the server terminal is a semi-colon. Therefore, I added an if statement that would run a check to ensure that a semicolon was not present within a file. After making this fix, I re-ran the same exploit and confirmed that the vulnerability was no longer present.

- **Conclusion**: Through my thorough analysis and remediation of this vulnerability helped to enhance the security of the giftcard website, preventing potential OS command injection attacks and providing greater protection to users and their sensitive information.

Assignment 2 – Spring 2023

NYU - Raj Shah

Part 2 – Encryption Explanation Writeup

Database Encryption:

- To implement encryption in the database, I utilized the Django Cryptography library. First, I installed the library and then added the following code to my models.py file: "from django_cryptography.fields import encrypt". Next, I applied the encrypt() function to all sensitive information in the models.py file, encrypting the information to enhance security such as the product_name, product_image_path, recommended_price, description, data, amount, and fp. The reason these are encrypted is to prevent unauthorized users from accessing it. After encrypting, the encryption keys are stored in the settings.py file such as 'SECRET_KEY'. However, since the secret key was stored in plaintext, I decided to make some key management actions (listed below) to prevent any unauthorized actions.

Key Management:

- Web applications rely on specific parameters to function correctly, such as the secret key and debug status (environment variables), which is typically located in the settings.py file. However, having these values exposed in the settings.py file can pose a vulnerability since they need to remain secret and hidden from attackers. To address this issue, I used the python decouple library to separate setting parameters from the source code.

- I created an environment file called .env in the repository's root directory to store secret keys and other environment-specific parameters. This approach allows for easy movement of the codebase between environments without any need to update the code.

- After installing the python-decouple library, I imported config from the decouple library and utilized the config() function to retrieve configuration parameters that were to be stored in the .env file instead of the codebase. By doing so, I removed plaintext secret keys from the codebase and implemented config('SECRET_KEY') to reference the keys from the .env file whenever needed, making it more secure.

- Managing keys in this way is recommended since storing secret keys directly in the settings.py file can limit flexibility, portability, and version control. Using a .env file, on the other hand, offers an additional layer of security and more flexibility for managing secrets. This approach also enables easy portability between environments and excludes secrets from version control.

    - Resources used:
        - https://pypi.org/project/python-decouple/
        - https://www.youtube.com/watch?v=LkyhTqDrSxA
        - https://pypi.org/project/django-cryptography/
        - https://django-cryptography.readthedocs.io/en/latest/examples.html