The Wall of Secrets

CSCE 4550 Fall 2017

Submitted To: Dr. Pradhumna Shrestha

Author: Joseph Vargas

Introduction:

This project is a website where users can post secret messages. The messages are each encrypted with a secret key. Only the secret key can decrypt the secret message back to its original format. This encryption is achieved using a specific kind of polyalphabetic substitution called the "Vigenere Cipher". The Vigenere Cipher is a method by which alphabetic text is encrypted by using a series of Caesar Ciphers (sometimes referred to as ROT-13) to rotate each character by a different amount based on the letters of the key. Its name comes from "Blaise de Vigenere" who has been misattributed as the creator of the cipher. The real creator was Giovan Battista Bellaso who first described the cipher in 1553.

I chose this project because I wanted to create a simple website to showcase the utility and limitations of the Vigenere Cipher and encryption in general. This project shows a basic encryption method which uses a key. As almost all encryption schemes use keys, I thought this showcase could help to solidify the understanding of the uses of keys in cryptography.

This project is also a good exercise for anyone who is interested in analyzing encrypted text. The system outputs encrypted text which anyone can do cryptographic analysis on to see if they can figure out a key or the plain-text message.

As was mentioned previously, the Vigenere cipher works by using a Key. The key is repeated as many times as necessary to become the same length as the message to be encrypted. After the lengths are the same, the plaintext is encrypted by rotating the characters of the plaintext according to the letters of the key. For example, if the letter "D" is in the key, the character in the plaintext corresponding to that letter would be rotated by 4 places. This transformation is repeated over the whole message for each letter in the key.

Because of the way the key is used, the vigenere cipher can also be thought of as an alphabetic version of the one-time pad. However, even though it took 3 centuries for a general method of breaking this cipher to be invented, the vigenere cipher is not theoretically unbreakable in the same way that the one-time pad is.

Methodologies:

Since I decided to make a web application, I knew I had several options for frameworks, software and languages to create this system. In order to start, I first had to make a design and some decisions. I started out by sketching a rough picture of what I wanted the front end to look like. Then I moved towards looking at languages and frameworks. I have the most experience creating viable front-end and back-end projects with Python, so I started thinking about which python-specific back-end framework I wanted to work with. I researched a few frameworks such as Flask and Bottle. However, I eventually chose the Django framework. I chose Django because I already have experience creating a full project with it, so I felt it would help speed the process and improve the overall quality in the end.

After I chose Django as a framework I made a list of what classes and functions I would need to implement. Among them were the database functions as well as the functions for taking and handling form input form the website. Central to the project are the functions which encrypt and decrypt user input.

In order to develop the encryption and decryption functions, I did some research on the algorithm that is used to encrypt alphabetic text using the Vigenere cipher. Using descriptions of the algorithm from my research, I was able to implement the Vigenere cipher encryption and decryption algorithms. I tested my implementation by attempting to reproduce a canonical example which is shown on the Wikipedia page for the Vigenere Cipher. The example I used was encrypting the phrase "attack at dawn" with the key "lemon". This phrase, when correctly encrypted using the Vigenere cipher produces the text "LXFOPVEFRNHR". The original cipher seems to require the user to use text without spaces. In my implementation, the user input can contain spaces. This is because in my algorithm, I remove any spaces from the input before processing the text.

After I verified the output with the canonical key and message, I was able to begin progress on the actual web-application. I created a github repository and started a new Django project in my IDE of choice, PyCharm by JetBrains. I created the Django page templates first, since writing HTML is the quickest part of the development process. I used CSS I was familiar with from a few other projects in order to create an attractive front-end. Then I moved forward with developing the back-end operations within the Django project. The first steps were to create Django "Apps", which correspond to each major portion of the website. I implemented the functionality of the front page first. In order to store the secret messages for the users I needed to create a model to store information in the database. I created a model called "Post" which holds the encrypted content of the post. Next, I created the structure of the form to which the user will input their message. When the homepage detects the HTTP "POST" function from the user entering a message, it will take the input from the form and store it in the database. Then the template will display the encrypted information back to the user on the homepage.

After I finished the functionality for the posting of encrypted messages I started developing the page in which the user can attempt to encrypt the messages other users have posted. I was able to recycle most of the HTML code from the homepage to create the decryption page. In order to complete this page of the web-app, I needed to create and develop a few more components. I started out by writing the form for the users to input a key to try to decrypt the messages in the database. However, this particular form does not need to interact with the database. So I implemented it in such a way that it only passes the key as input to the program. That input is then used to display the results of the decryption to the user. I use the key on each of the encrypted messages, and redisplay the results. If the end-user can see any readable messages, then their key was successful, and they decrypted a message. Ideally any key should only be used once, however it is entirely possible that a key is used twice. In that case, each message encrypted with that key will be decrypted and shown to the user. This could be a possible security issue. It calls into question the secrecy of messages. As well it shows a weakness of this kind of encryption.

Results:

This system allows users to encrypt a secret message with a custom key using the Vigenere Cipher. With a fresh database with no entries, the user will see the front page with no posts under the "Posts" section of the website as pictured in **Figure 1.** A prompt to "Write Your Message On The Wall" appears above the form where a user can enter their message and a key of their choosing. The form input supports spaces in the text of both the Key and the Message. However, the output will contain no spaces to obscure the message further. As shown in **Figure 2**, the user will first choose a key, and then enter a secret message. Then the user can press the "save" button and the post will be encrypted with the key and posted to the wall as shown in **Figure 3**.

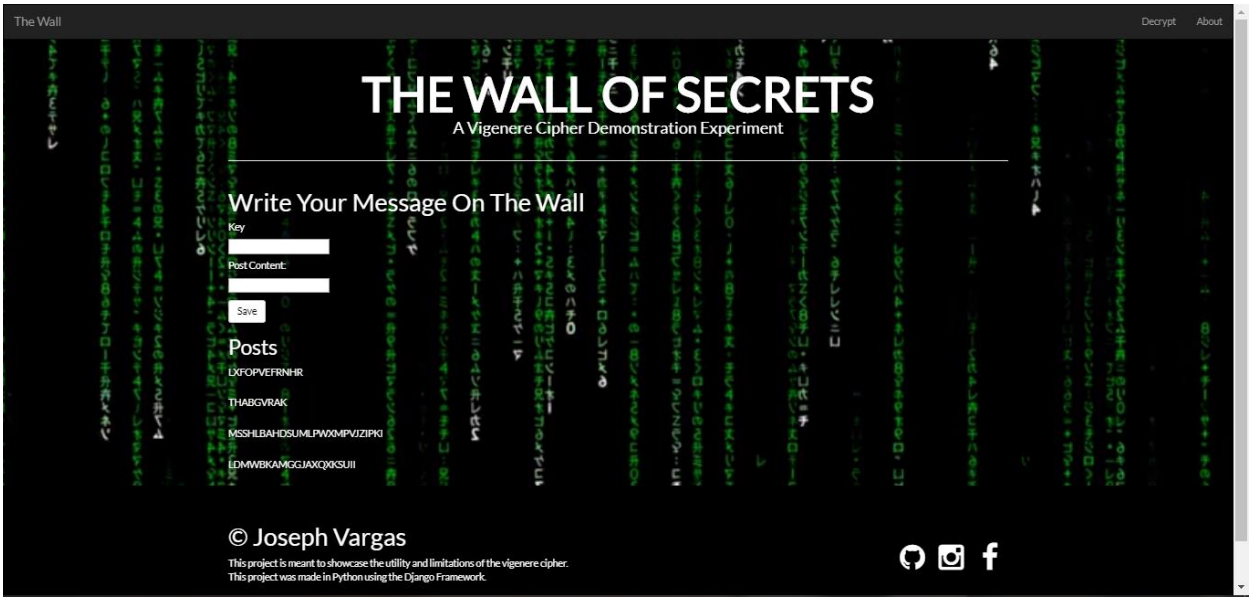

Figure 1: The front page of The Wall of Secrets which is displaying some previously input secret messages. For our examples and the final submission, I have left only one previously input message.

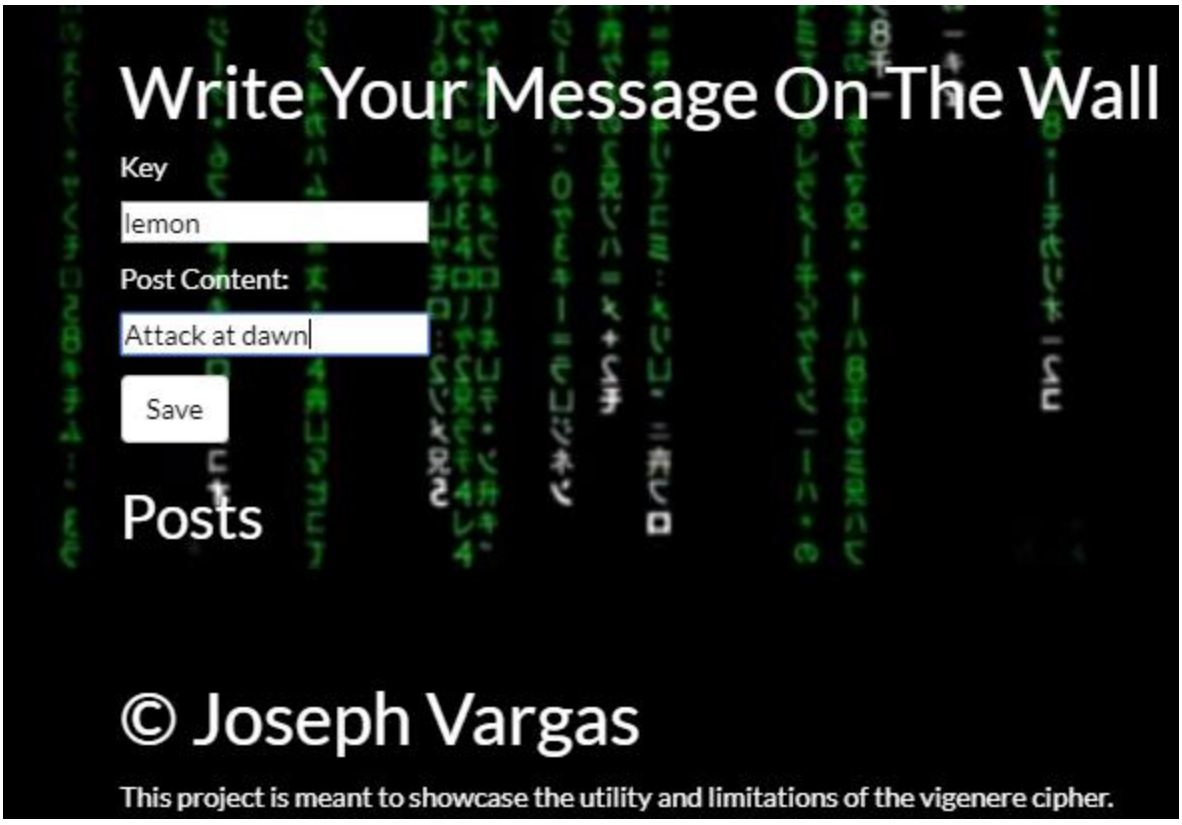

Figure 2: This screenshot shows me entering the canonical example as shown on the Wikipedia article for the Vigenere Cipher. This is the only post I will have included with the final submission.

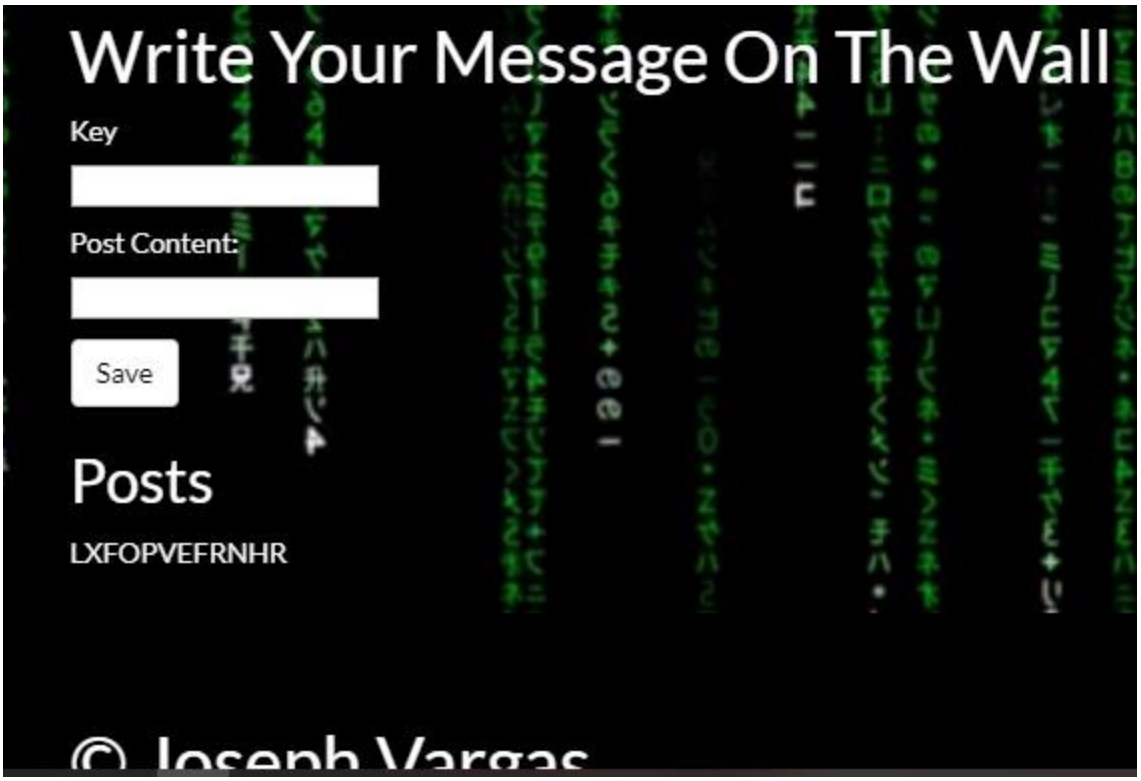

Figure 3: This screenshot shows the results of our input. The backend of Django has encrypted the message and saved it to the database and the webpage is now displaying the encrypted message to the user.

The system also supports users who wish to try to decrypt a message that has been posted on the wall. In the navigation bar, the user can click the "Decrypt" link and will be taken to another page which displays the posts on the wall and a form where they can enter a key with which the system will run the decryption algorithm (see **Figure 4**). To use this page, the user can enter a key (**Figure 5**) and then enter it to run the decryption with that key on all the posts on the wall as seen in **Figure 6**. The user should be able to scan the output quickly and tell if any readable output is produced. However, only the post that has been encrypted with a certain key will be readable. If the user inputs a key which does not successfully decrypt a post on the wall (**Figure 7, Figure 8**) the post will remain unreadable. When this happens, the user can either attempt to decrypt the posts with another key or, "reset the wall" with a link that refreshes the page and shows the originally encrypted messages.

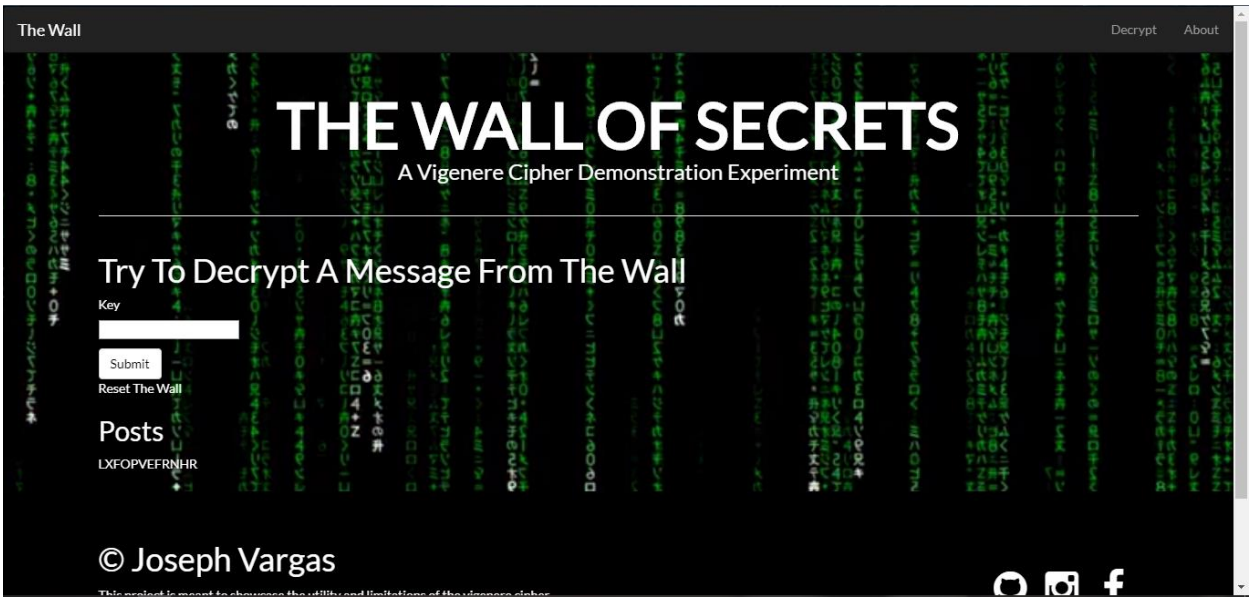

Figure 4: This screenshot shows the "Decrypt" page. Here users can enter a key and try to decrypt the messages on the wall

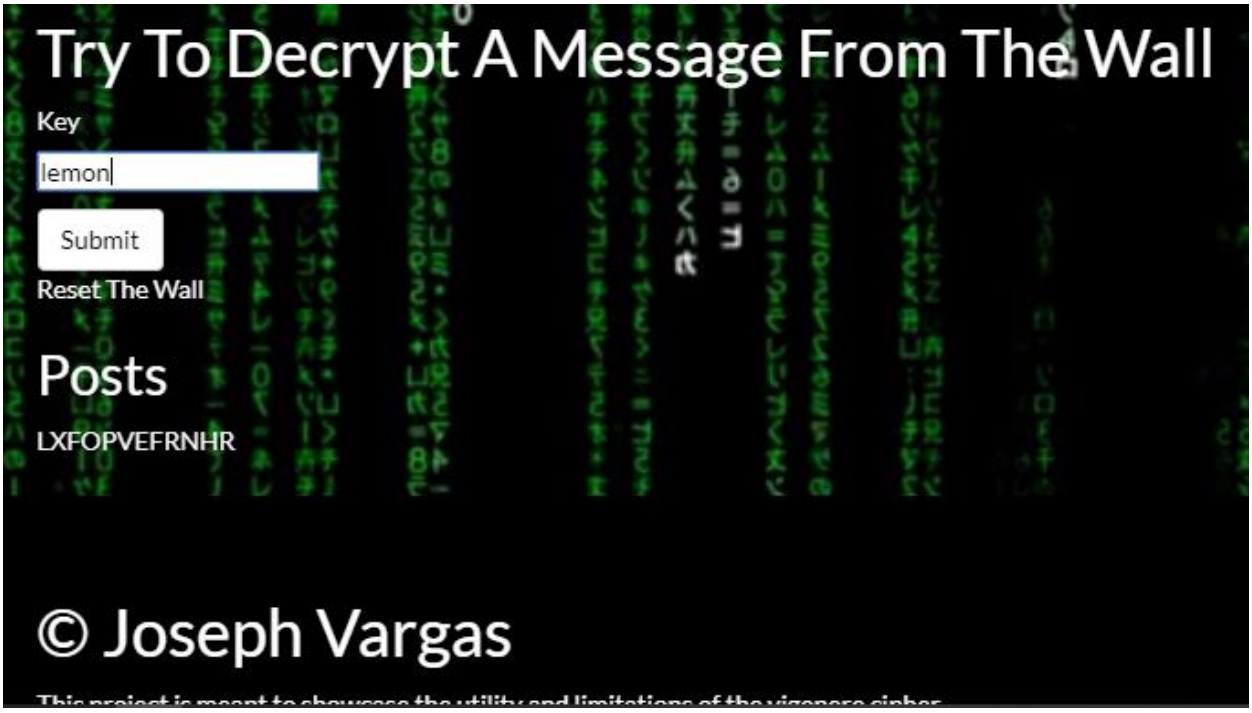

Figure 5: This screenshot shows the user entering the correct key. Before hitting submit, the post displayed is still encrypted and unreadable.

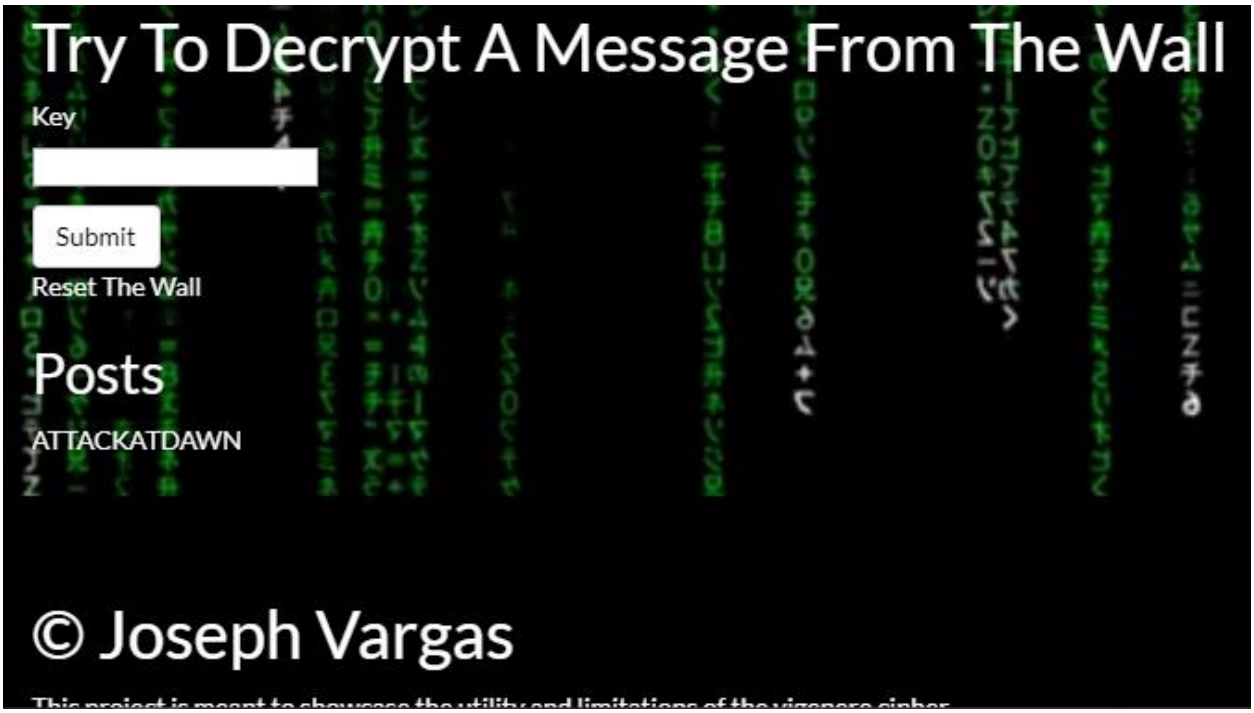

Figure 6: After the user submits their key, Django runs the decryption algorithm on all the posts using that key and redisplays them to the user. Here we can see that the user's key was correct, and the original message is readable as "Attack at dawn". It is important to note that this does not modify the database entry which contains the encrypted post. This simply displays the transformation.

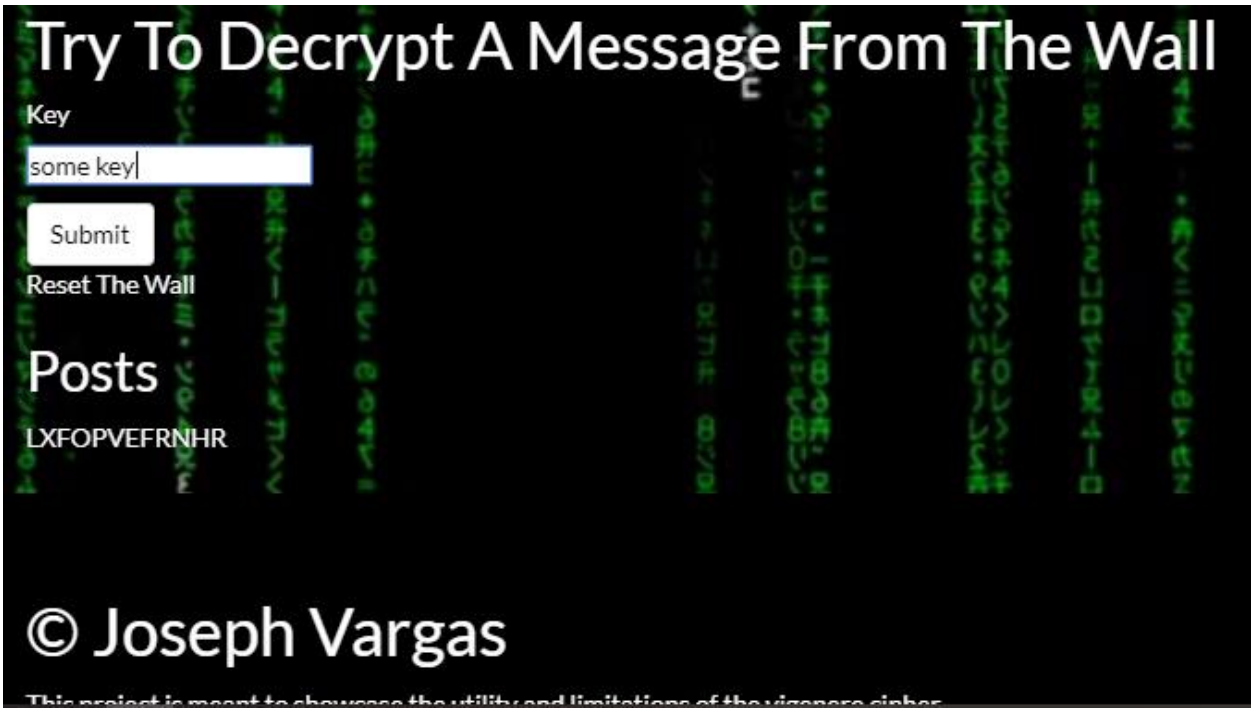

Figure 7: In this screen shot we can see the user is entering a wrong key. When they hit submit, the post will not be correctly decrypted.

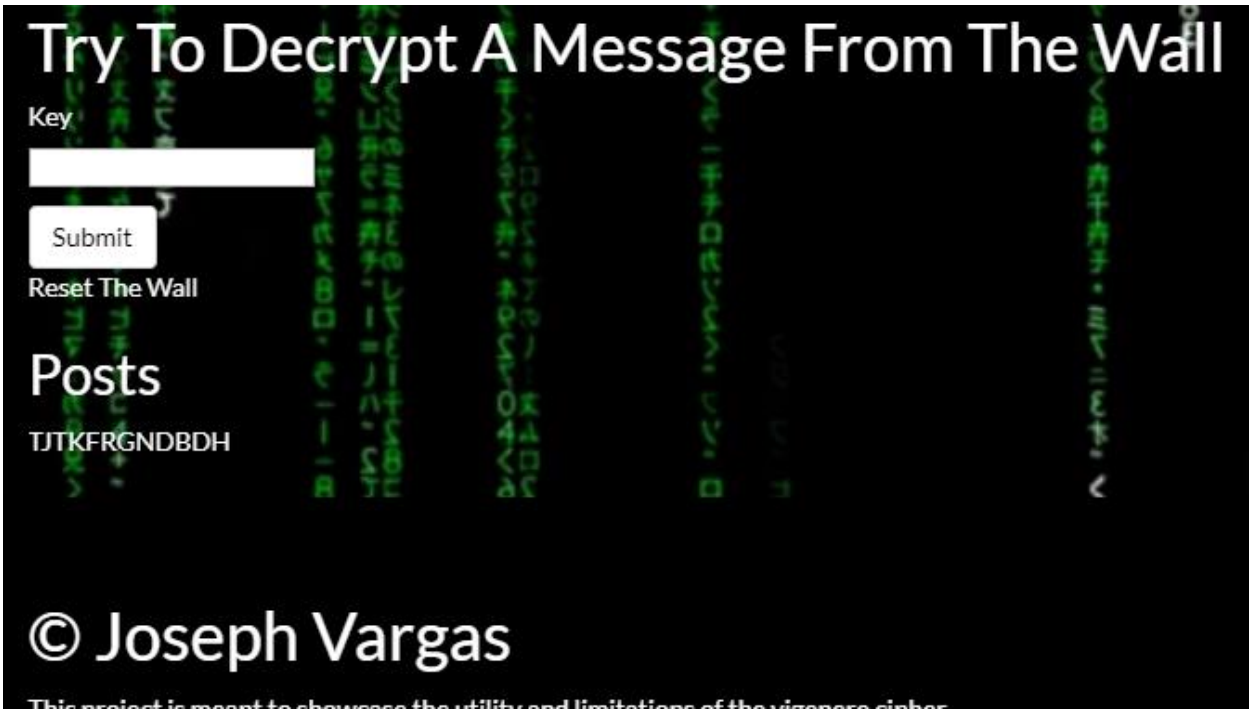

Figure 8: We can see that the post is different than the originally encrypted post (**Figure 7**), however it is still unreadable. The user has entered the wrong key. From here, the user can either enter a different key, or click the link to "reset the wall" which will redisplay the originally encrypted message.

Limitations:

I originally intended this web-application to contain only one page where the user can interact with all the features. However, due to the way I designed the page, I felt it would be cleaner to implement a separate page for the Decryption function. This makes the features more distinct and makes the user experience easier in my opinion.

If I was better at HTML and CSS I would probably have been able to create a cleaner look that would fit all the features on one page. However, the way I ended up implementing it has its own strengths.

Looking at the final output, I think there could be a few improvements. However, these "improvements" may impact the security. Namely, I think the output would look nicer if each post had a timestamp attached to it. However, this piece of information could make it easier to figure out who posted the secret message. Another possible area where security could be compromised is in the database. Currently, the database stores the key of each post. If the database were breached, an attacker could obtain the keys to each message, and decrypt each one.

As well, the message is first transmitted via an HTTP "POST" command. Since this project does not implement TLS, the contents of this POST command could easily be intercepted by anyone using a packet sniffing program. The message could hypothetically be read before it is even encrypted.

In order to fix this, I would have had to implement the encryption via JavaScript on the client

side. This would successfully encrypt the text before it could be sent to the database. Meaning, that not even the server or database would know the key or the original message.

Another possible but unlikely vector of attack could be a TOCTTOU attack inside the web server. Once the system receives the "POST" content, it briefly stores the unencrypted message in a variable. Nefarious code could hypothetically store or even modify this unencrypted message before it gets transformed using the Vigenere Cipher. In order to fix this kind of vulnerability, once again using client-side scripting with JavaScript would help defend against the content of the message being changed. Although, server-side mediation could also be implemented to make sure the message hasn't been altered even after encryption.

Conclusion:

This project taught me a lot about the techniques and pitfalls of basic cryptography. I solidified my understanding in the importance of keys to cryptography and learned how to implement the Vigenere Cipher correctly. As well I learned how to use the Django framework much better than I had before. Through the exercise of designing and implementing this web-application I have gained more knowledge of HTML, CSS, python, and security in general.

In hosting this project on Github, I hope to be able to show future employers that I have a good understanding of security, the python language, the Django framework, and basic HTML/CSS.