**Secret Keeper: Final Report**

03.13.2020

**─**

Teammates

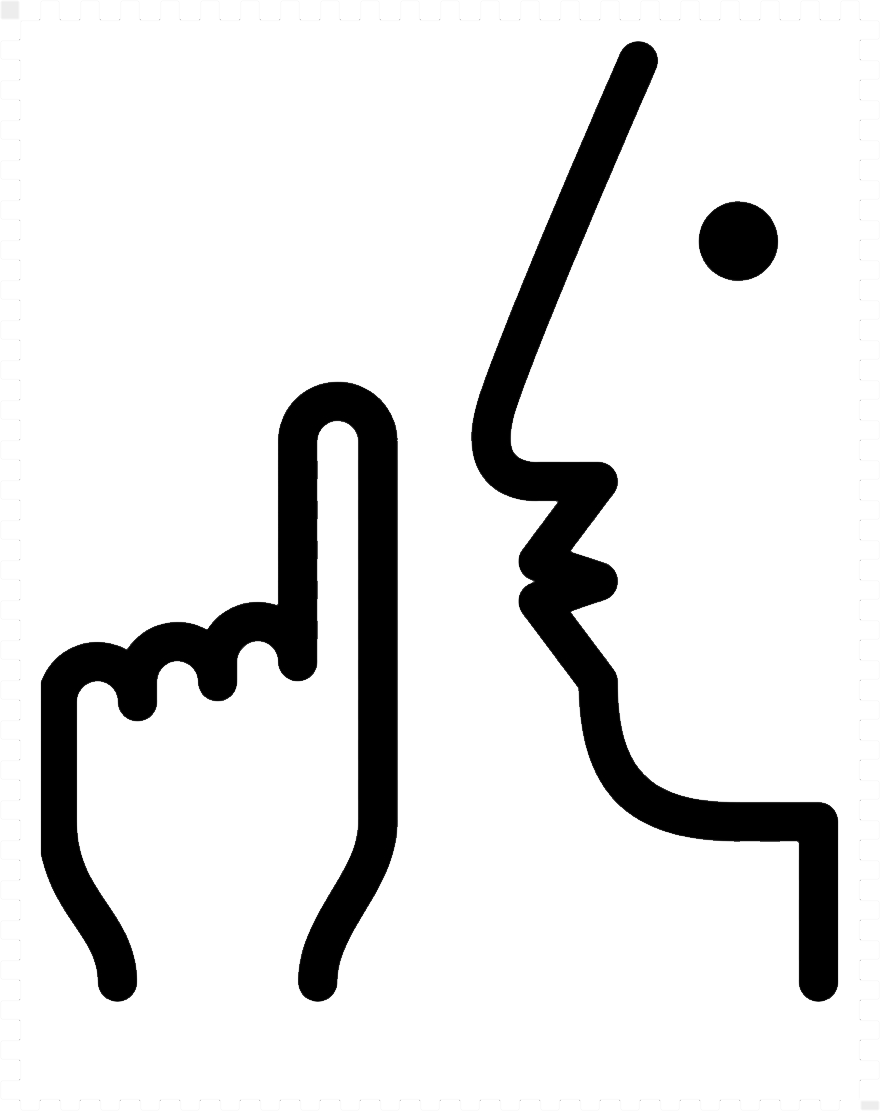
Kyle Dixon

Katie Young

# **Introduction**

The United States government estimates that malicious cyber actors cost the U.S. economy anywhere from $57-109 billion in the year 2016 alone (CEA, 2018). In response to the growing threat from these malicious actors, the Open Web Application Security Project (OWASP) Foundation created a list of the [top ten cyber security risks](https://owasp.org/www-project-top-ten/), as well as security implementations to address the risks.

The Secret Keeper project was designed and implemented at the request for an apparatus which illustrates the attacks outlined in the OWASP list. Three main components make up the project: a ‘weak security’ site, exploits designed to take advantage of the weak site’s security flaws, and a ‘strong security’ site designed to stop the exploits. Both weak and strong sites allow users to create profiles to save private data which can later be accessed and modified.



Throughout the implementation process, we learned that a simple change in a single line of code can enable injected-script execution, that hackers can evaluate thousands of passwords for correctness in the blink of an eye, and that no knowledge of the organization of an application’s database is required to gain access to a user’s account if the website doesn’t have appropriate security measures to prevent it.

Additionally, we became increasingly aware of the amount of creativity and trial-and-error required to perform these exploits. As such, it seems unlikely that all threats can be predicted, tested and prevented. It became clear to us that, in the future, additional measures for detecting and halting successful attacks would further improve the security of our site.

To date, widely-used and trusted applications release new software versions [containing exploitable vulnerabilities](https://www.kb.cert.org/vuls/). As hackers continue to learn new exploits, the need for strong security is more important than ever before. This experience has solidified our understanding that proactive and reactive measures are absolutely necessary to prevent attacks and minimize harm in the digital age.

# **Using the Site**

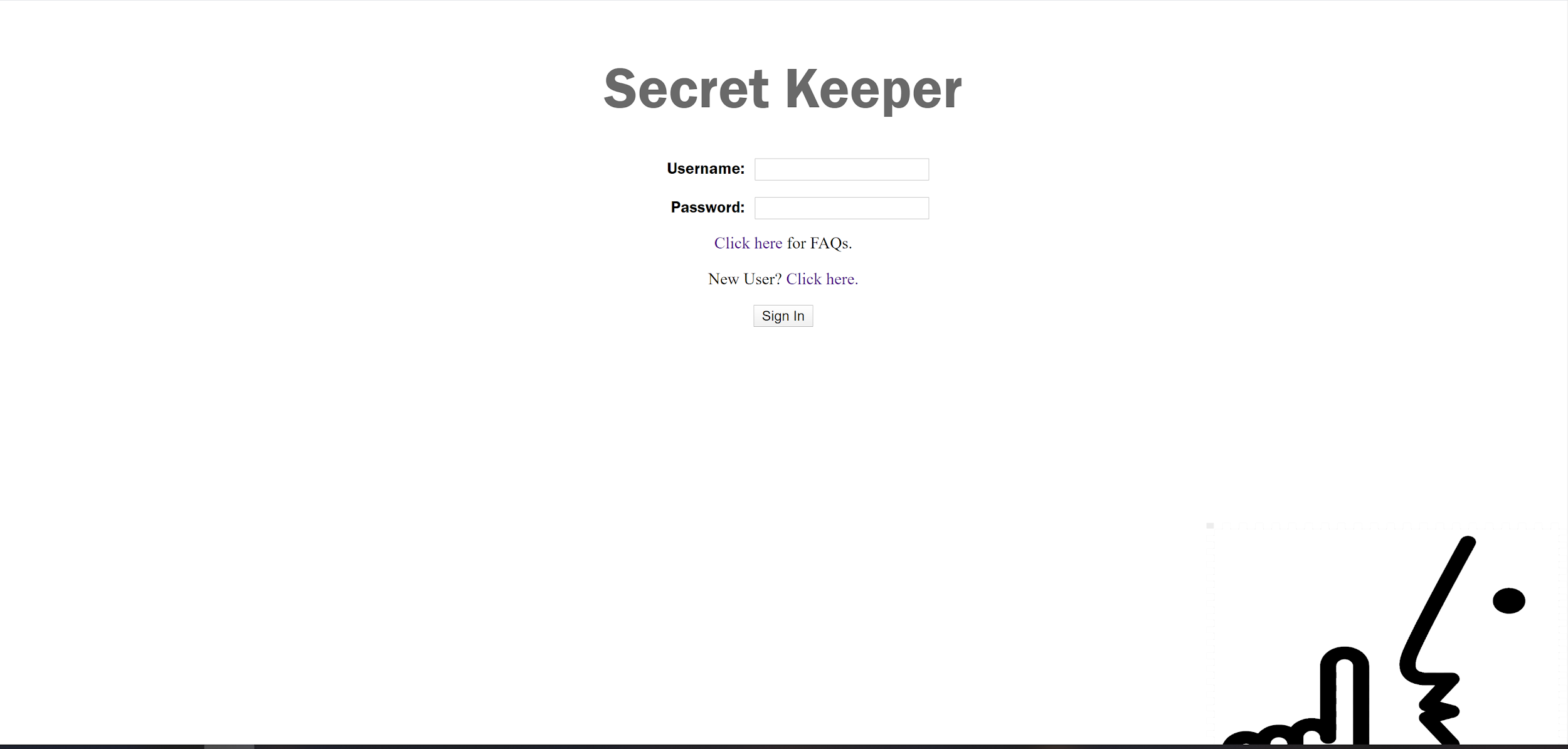
**To Access the Live Site**

The live site can be accessed using the following URLs:

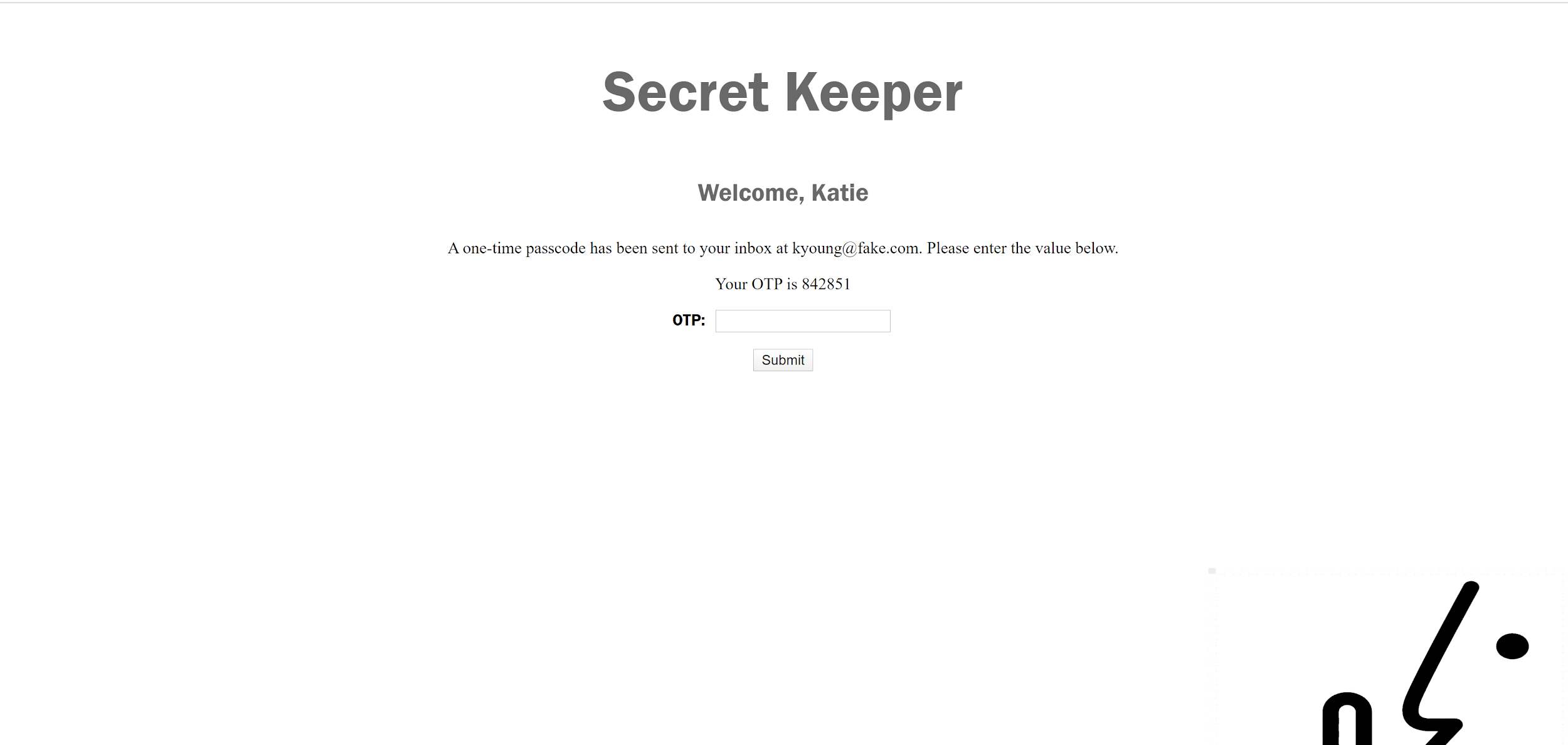
* Secure Site: <http://flip3.engr.oregonstate.edu:6060/>
* Not Secure Site: <http://flip3.engr.oregonstate.edu:6061/>

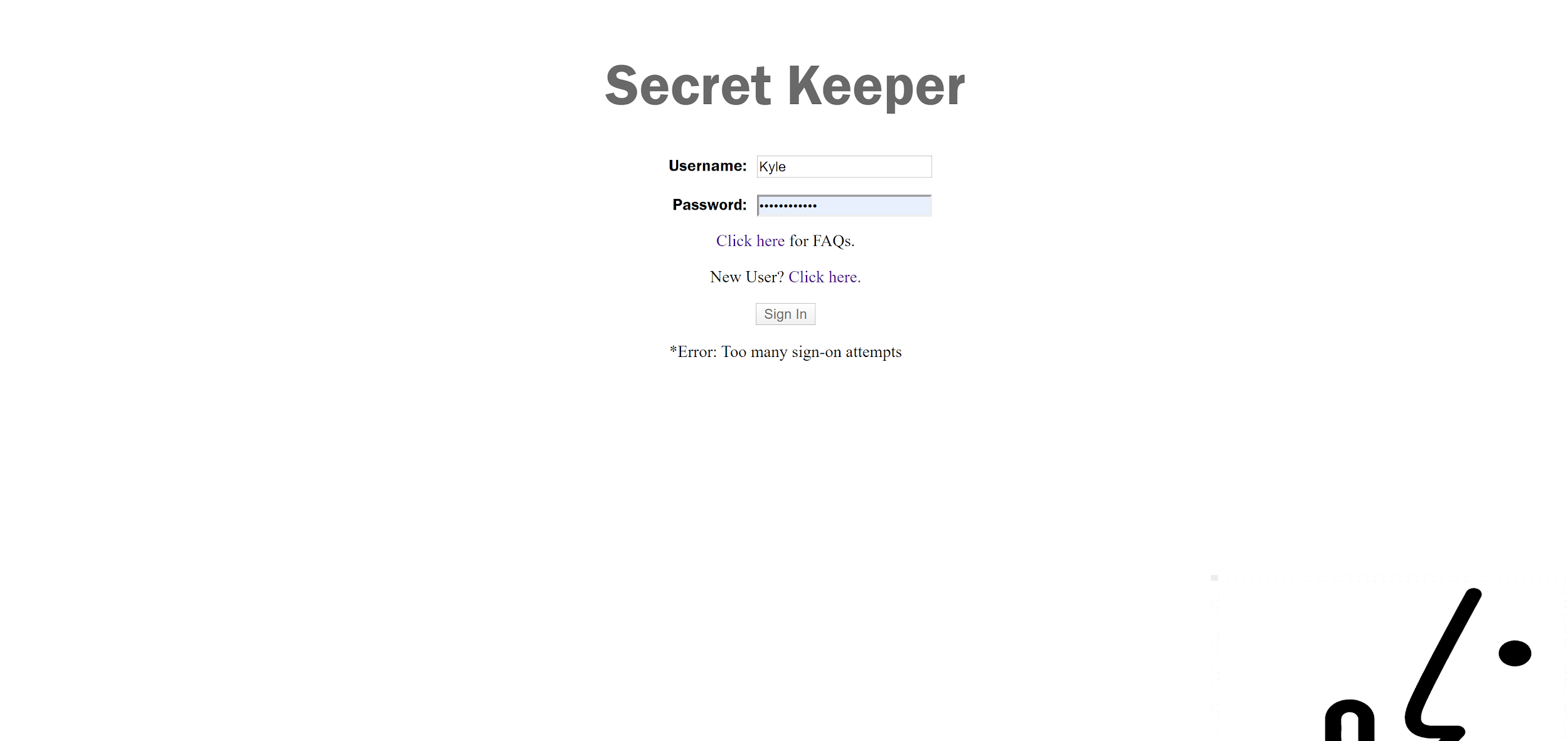
**The Sign-In Process**

The Secret Keeper home page appears identical to users of the weak and strong sites. Central on the page is a place to enter one’s username and password. If the user is new to the site, there is a link under the login which links to the user creation page. Additionally, there is a link to a frequently asked questions (FAQ) page.



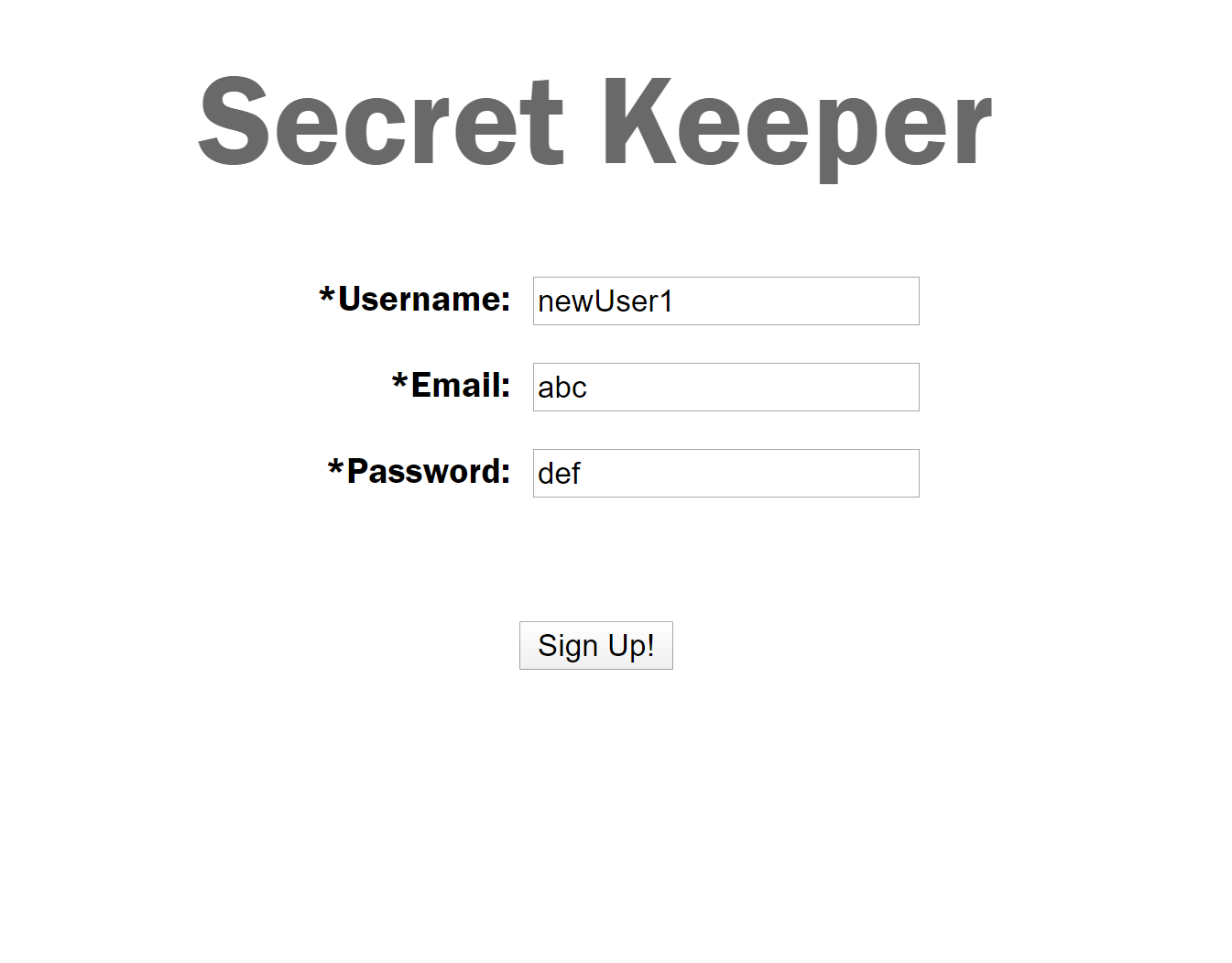
Successfully adding a correct username/password combination leads to different scenarios for each site. The weak site directly opens the user page, while the strong site includes a second two-factor authentication (2FA) step, which must be passed to open the user page. The 2FA feature includes a button that, upon being clicked, sends a randomly-generated, six-digit passcode to the corresponding user’s email. Correctly entering the passcode into the displayed text box opens the user page. *Note that, for ease of grading, we have chosen to display the emailed PIN on the 2FA page after the button is clicked*. In real-word scenarios, this line would certainly be removed. Users of the strong site may also notice that incorrect combinations of username and passwords will result in the login button being disabled depending on the rate of incorrect submissions and the username used.



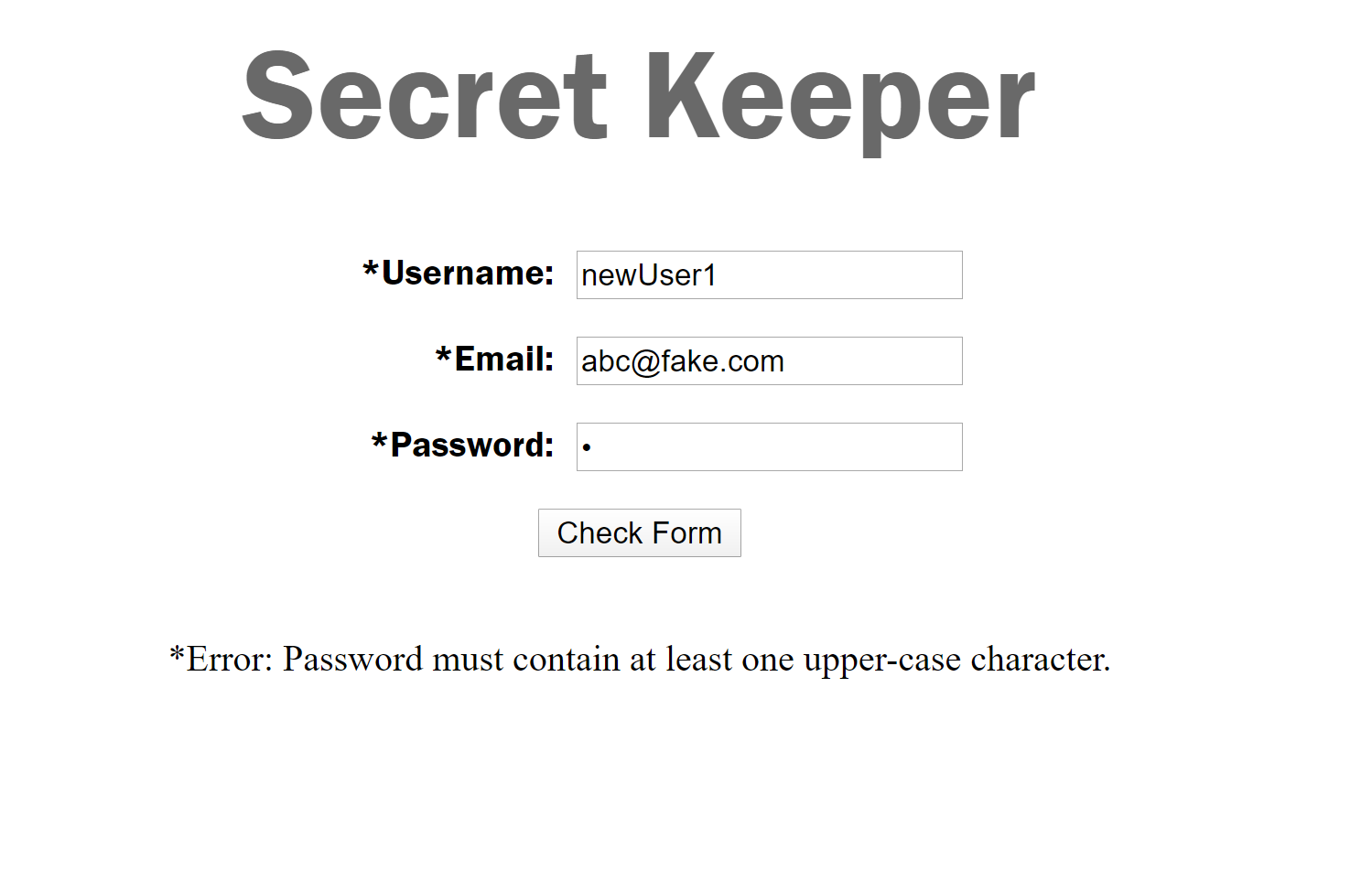
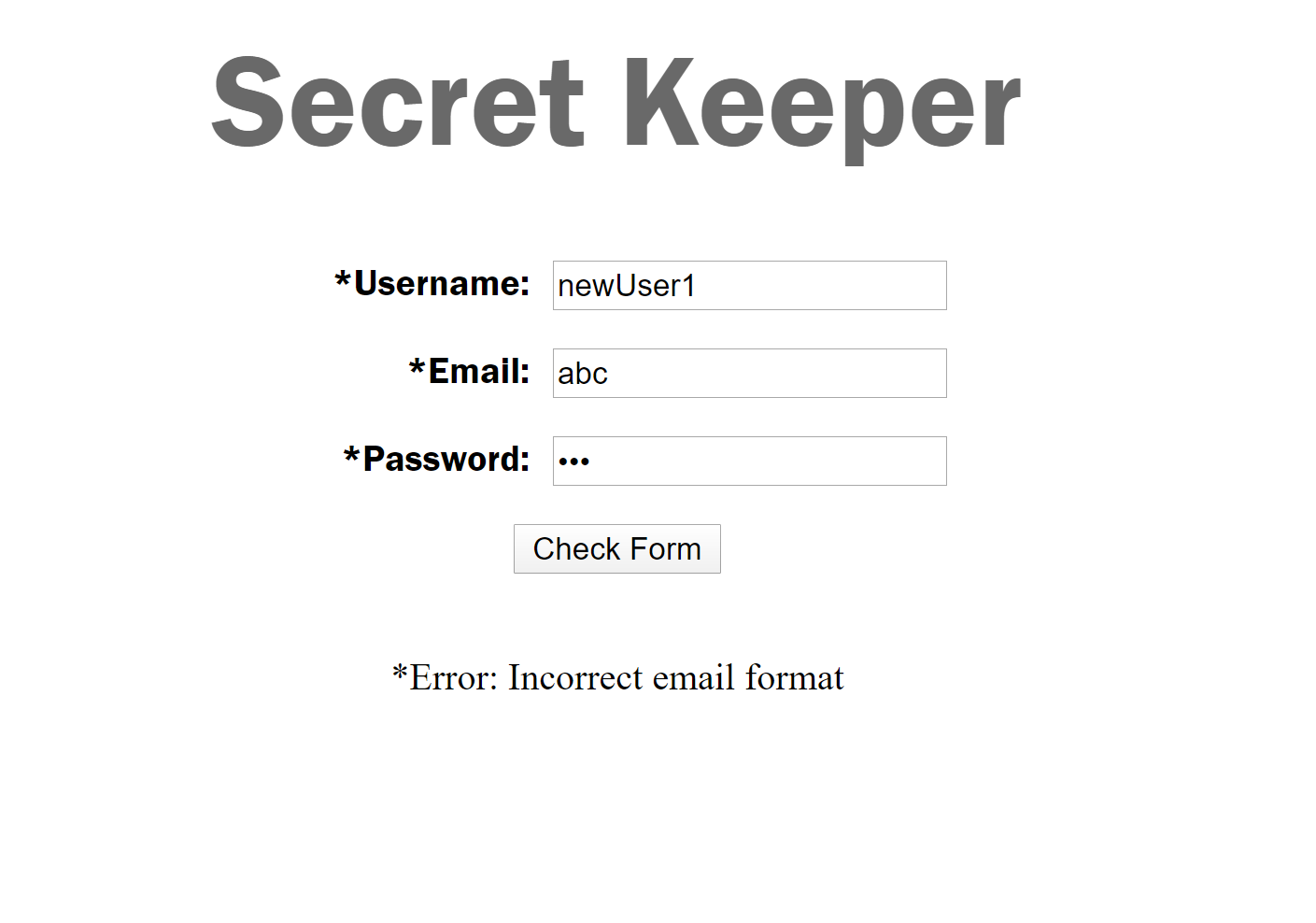


**Creating an Account**

The user creation page has a form for username, password, and email in both site versions. To successfully create a new account in the weak site, only the username must be unique to all saved usernames. To test if the username is unique, the user clicks on the “Check Form” button, which then searches profiles for the entered username and, if not found, displays a “Sign Up” button. If the user clicks this new button, a profile is created and the user is redirected to the home page.



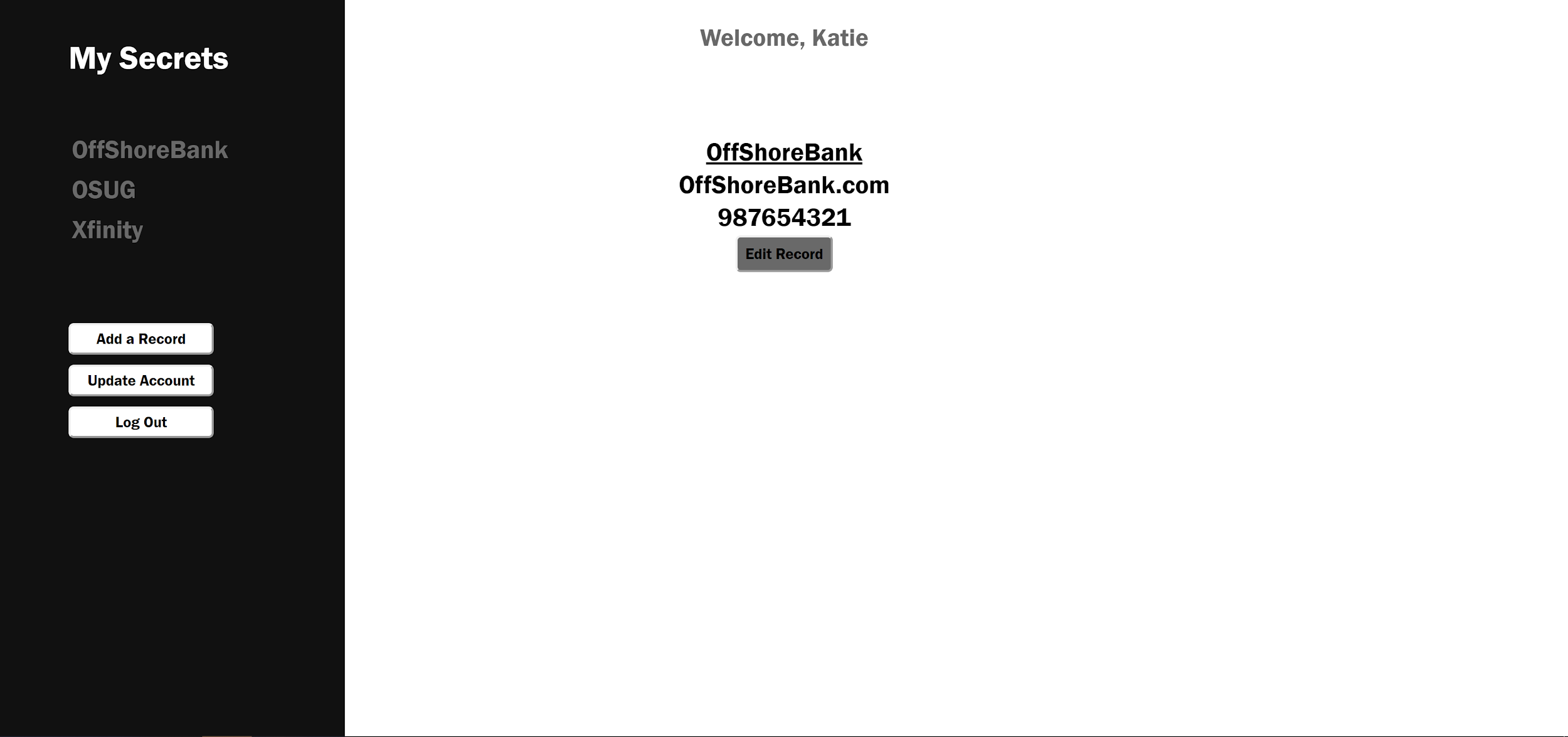
The strong site features more form checks than simply checking for a unique username. The entered password must include at least one upper-case letter and one lower-case letter. The password must also contain a number and be of at least 12 characters in length. Finally, no special characters are allowed in the password. As with the weak site, the user must click on the check form button and pass the checks before they are able to click on the signup button and create a profile.



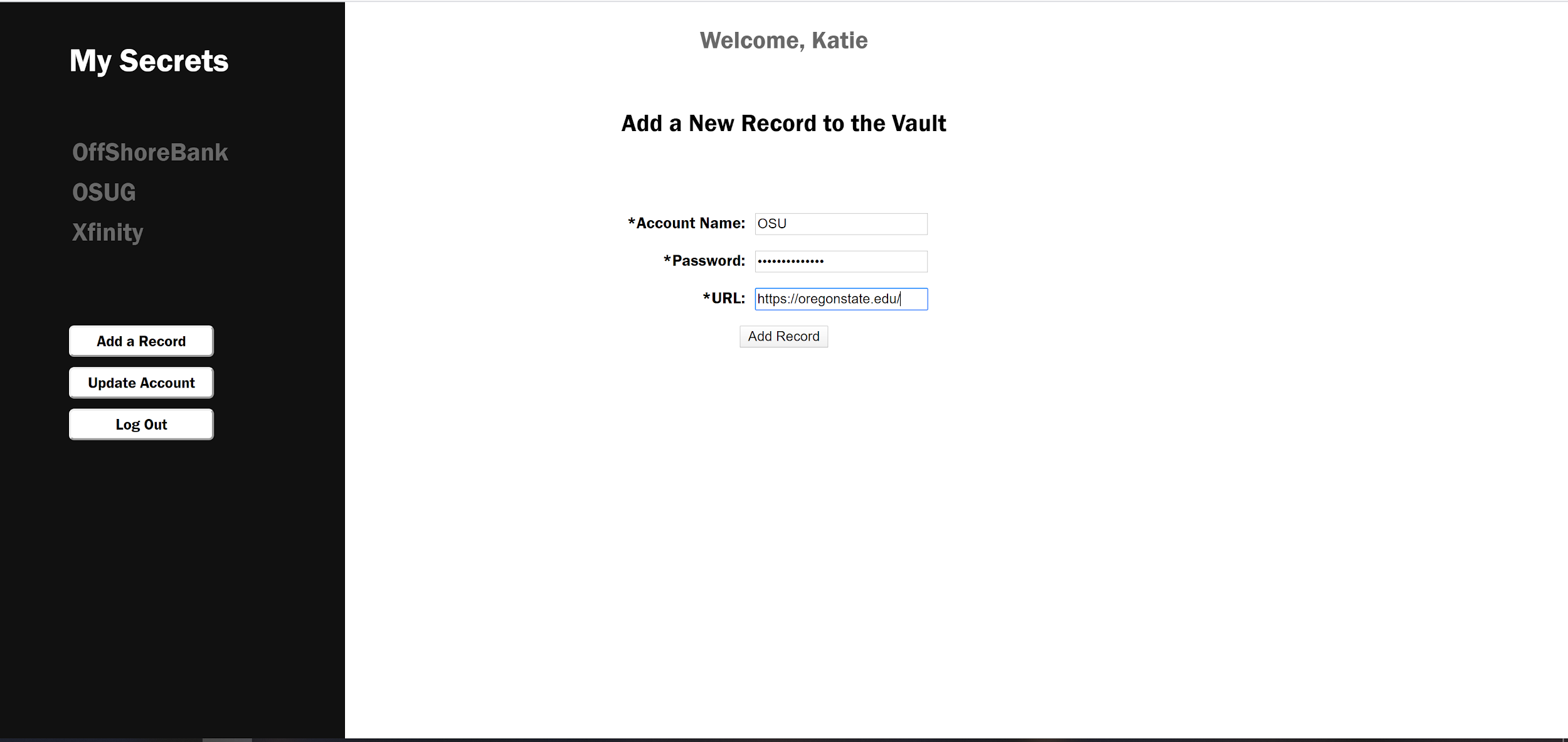
In the secure site, the password is also hidden from view on the user creation and home pages, as well as the update-account form on the user profile.

**Using the Site**

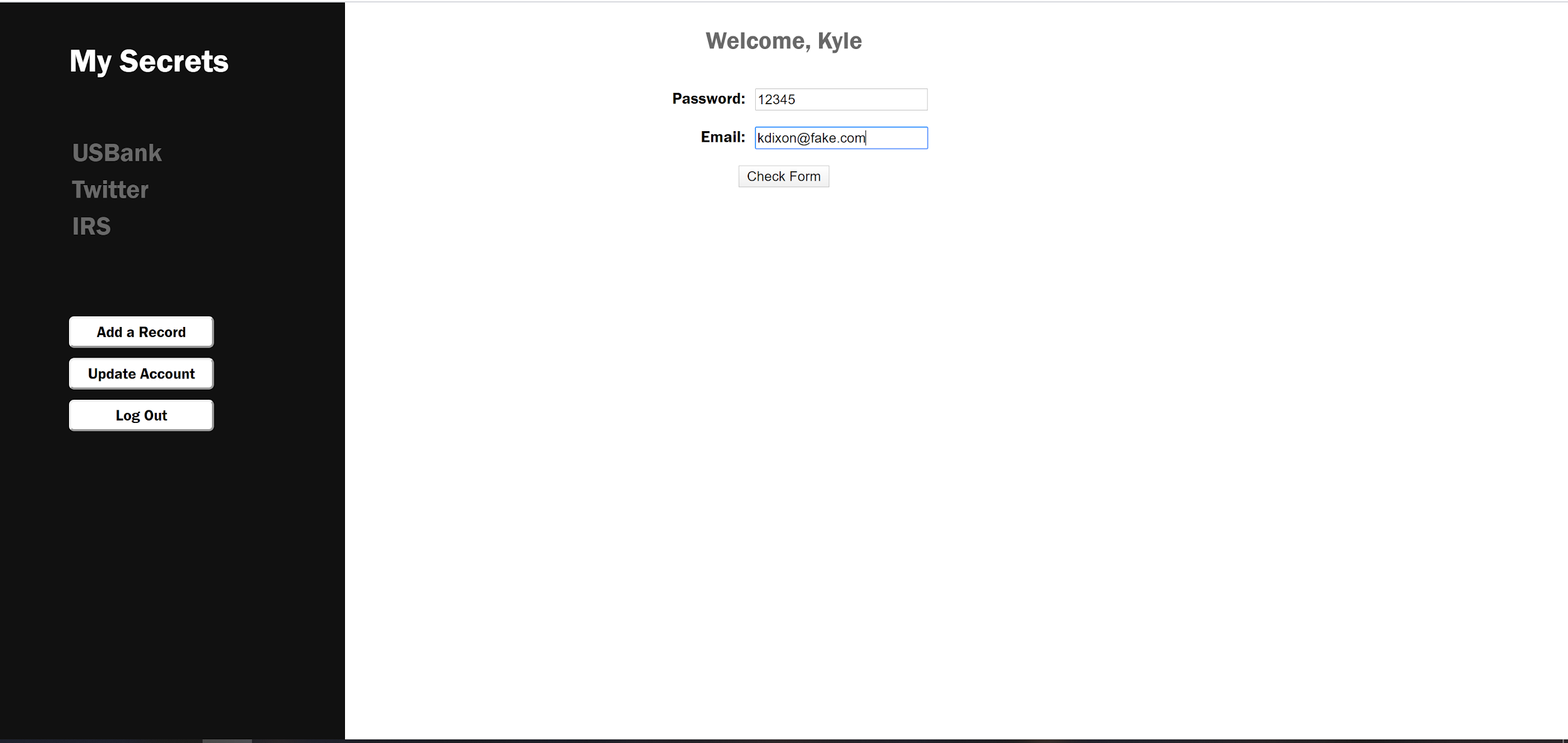
The user page is the location for users to add, edit, and delete their records. Records are composed of a name, password, and URL. Saved records are displayed by name in the left table of the page. Below the list of saved records are three buttons used to enter new records, edit the user profile, and return to the home page. Selecting to enter a new record opens a form on the center screen with text boxes for each record criteria. Saving the record reloads the user page with the new record now displayed by name in the left table.



Clicking upon a saved record name opens the record data on the center screen with a button at the bottom to edit the record. Choosing to edit the record turns the displayed record into a form which can be manipulated and saved. An option to delete the record outright is also displayed when editing the record. Editing or deleting a record reloads the user page with changes applied.

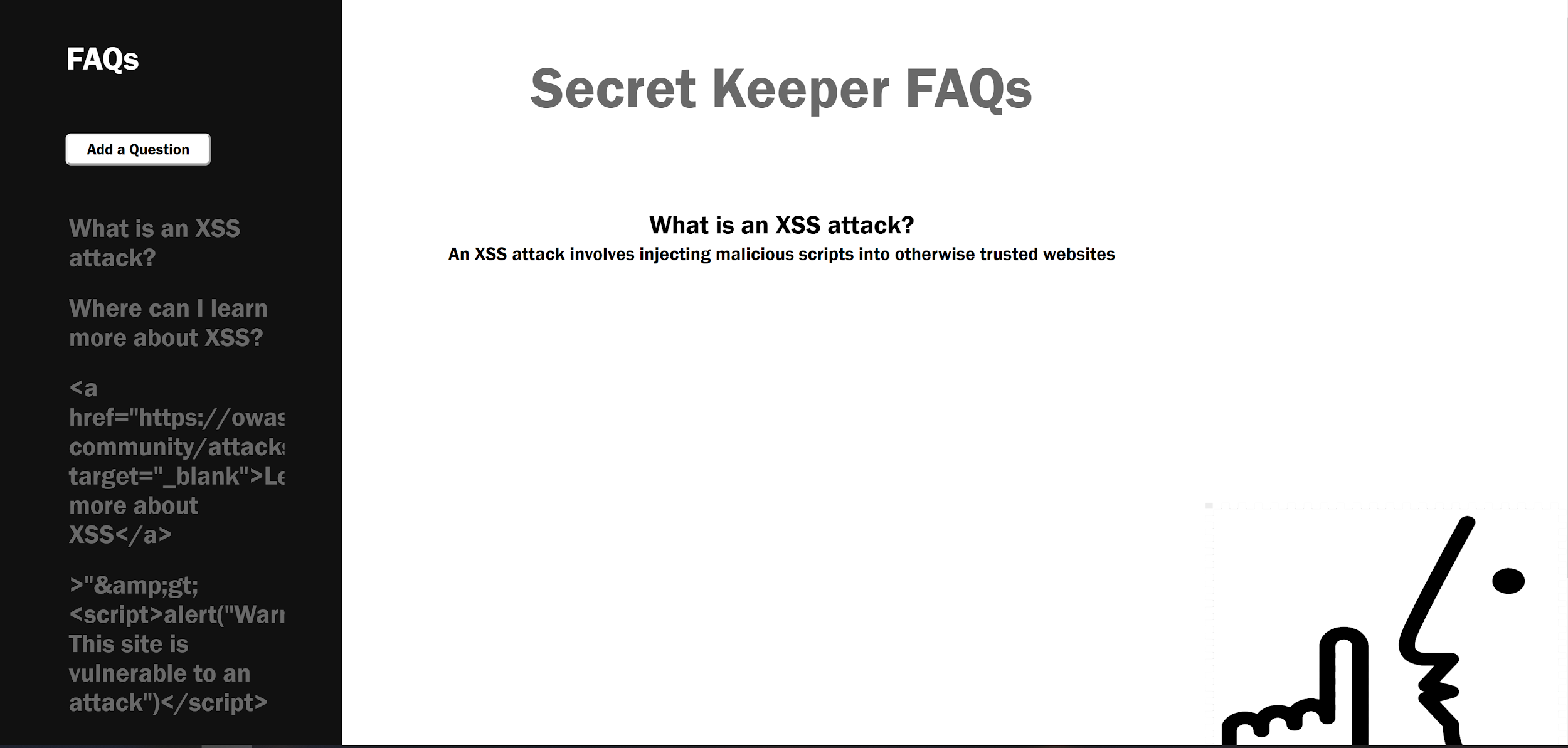
 

Selecting the edit user profile button opens a form on the center screen with options to change the user password or email. Users of the strong site have to pass profile credential checks similar to creating a profile inorder to save their changes. Successfully saving a change reloads the user page. Users of the weak site will be able to see their password change displayed directly in the URL.



**Frequently-Asked Questions**

The FAQ page serves as a way for users to get in touch with the administrators of Secret Keeper. Sample questions are listed on the page with sample responses. Users can enter a question of their own which is immediately displayed on the page. Responses to the question will be displayed when saved by the Admin account at a later time. This feature was a late addition intended to demonstrate a likely HTML injection scenario.



**Exploiting Site Vulnerabilities**

All exploits have been created for the weak site. When user credentials are needed, the Kyle profile (password: 12345) has been used as the example profile. This included some data hardcoded into the exploits. Many of the exploits listed can be used together as part of a broader attack against user privacy.

**Brute-Force Attack**

The brutePass script takes advantage of the PUT request on the home page to test username and password combinations. The request normally is used to test the count of the password and username combination in the user sql table. A malicious actor can use the poor implementation of leaving JavaScript on the bottom on the html file to see how the login function sends data and what the function is expecting to receive.

For the list-based exploit, usernames are provided by the script user and passwords are taken from passwords.txt which has the 10000 top passwords as listed by OWASP. The script looks for a JSON response with a count of 1, indicating that the password matches the username in the user table.

For the randomly-generated passwords exploit, usernames are provided by the script user and passwords are randomly generated using a list of allowed characters. The script looks for a JSON response with a count of 1, indicating that the password matches the username in the user table.

Associated Security Feature: login attempt rate-limiter

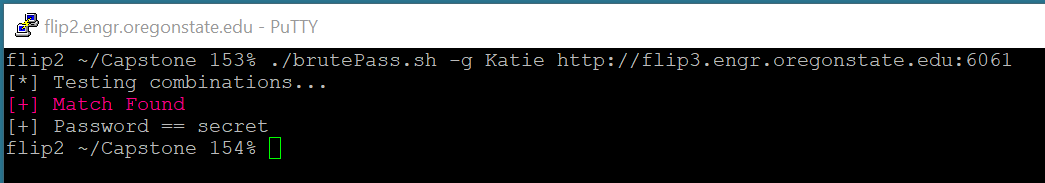
*Directions to use brutePass.sh:*

1. Place brutePass.sh and passwords.txt in the same directory
2. Open a connection to osu servers (do not use flip3, or the server hosting the site)
3. Open the directory with brutePass.sh
4. You may need to give yourself permission to run the script: chmod 777 brutePass.sh

To execute the script: ./brutePass.sh -g username URL

Examples:

* ./brutePass.sh -g Kyle flip3.engr.oregonstate.edu:6061
* ./brutePass.sh -g Katie flip3.engr.oregonstate.edu:6061
* ./brutePass.sh -g Admin flip3.engr.oregonstate.edu:6061



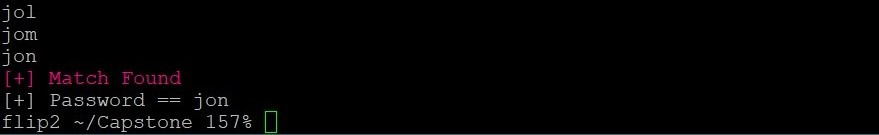
*Directions to use bruteForce.sh:*

1. Open a connection to osu servers (do not use flip3, or the server hosting the site)
2. Open the directory with bruteForce.sh
3. You may need to give yourself permission to run the script: chmod 777 bruteForce.sh

To execute the script: ./bruteForce.sh -g username URL

Example:

* ./bruteForce.sh -g John flip3.engr.oregonstate.edu:6061



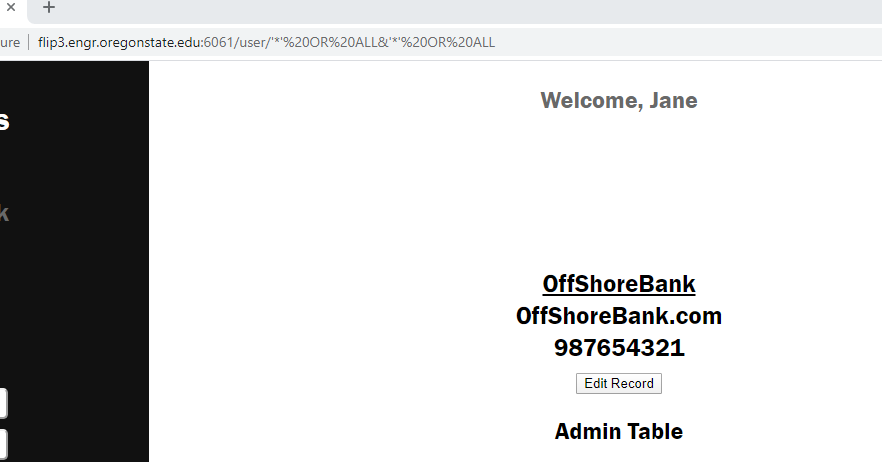
**SQL Injection Attack**

The username and password are passed in the URL for all user page requests. SQL code can be inserted into the URL, as placeholders are not used for MySQL credential checks in the weak site.

Associated Security Feature: SQL placeholders

*Directions to access users' information without their credentials:*

Navigate to the site using the following link [http://flip3.engr.oregonstate.edu:6061/user/'\*'%20OR%20ALL&'\*'%20OR%20ALL](http://flip3.engr.oregonstate.edu:6061/user/'*'%20OR%20ALL&'*'%20OR%20ALL)



**Persistent HTML Injection Attack**

Similar to the SQL injection attack, the lack of placeholders when passing data between component levels allows scripts to be executed by the client level of the weak site. This can become particularly nefarious if user credentials are stolen and malware, or a link to malware, is entered as a record for unaware users.

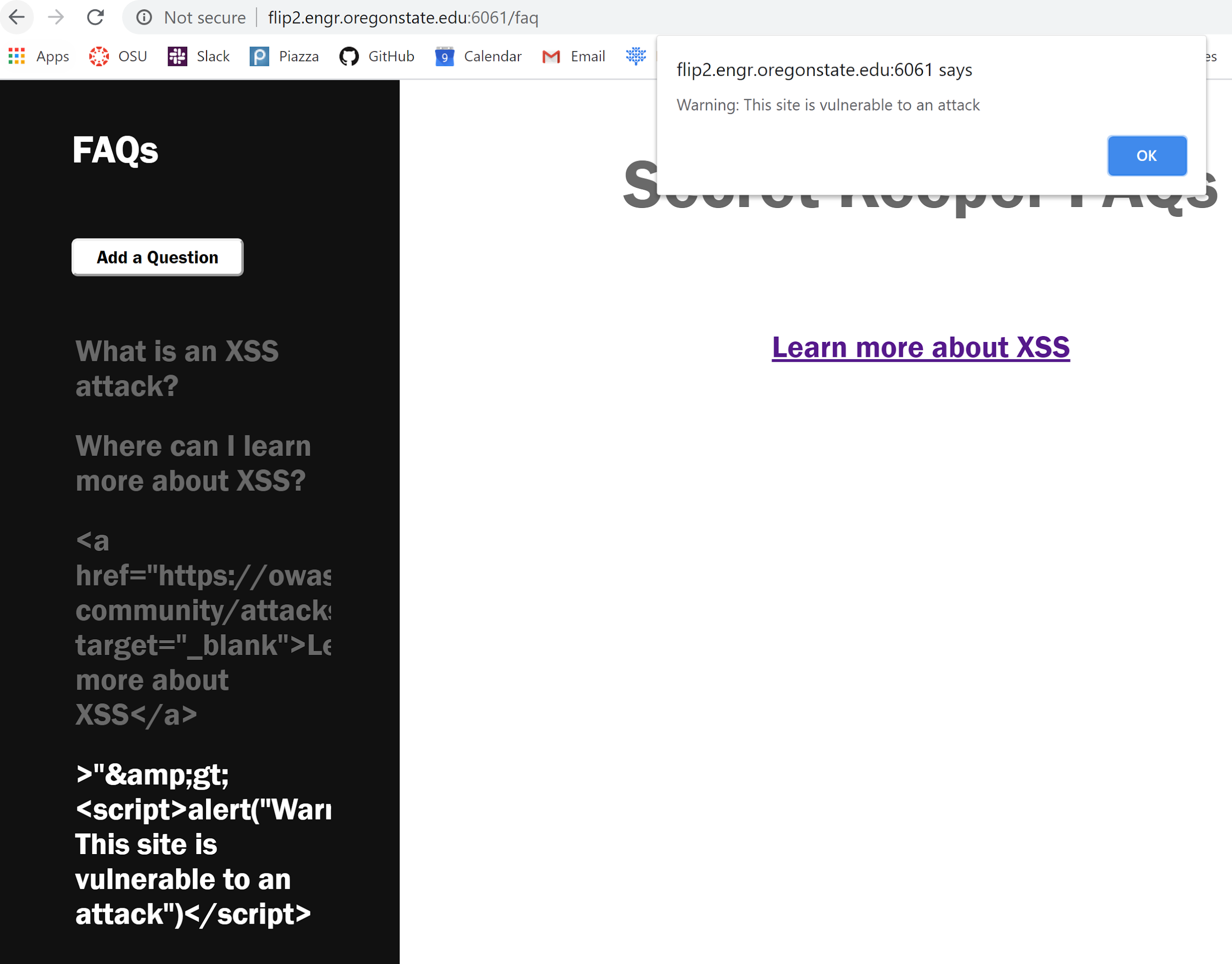
Associated Security Feature: non-html AJAX calls to prevent script execution via forms

*Directions to inject a script:*

1. Navigate to the FAQ page
2. Add a new question, using the following as the value:

>"&amp;gt;<script>alert("*Warning: This site is vulnerable to an attack*")</script>

1. Click on the question to execute the script



*Similarly, you may also inject a (potentially) malicious link:*

1. Navigate to the FAQ page
2. Add a new question, using the following as the value: <a href="https://owasp.org/www-community/attacks/xss//" target="\_blank">Learn more about XSS</a>
3. Click on the question and the corresponding link to leave the site

**Buffer Overflow Attack**

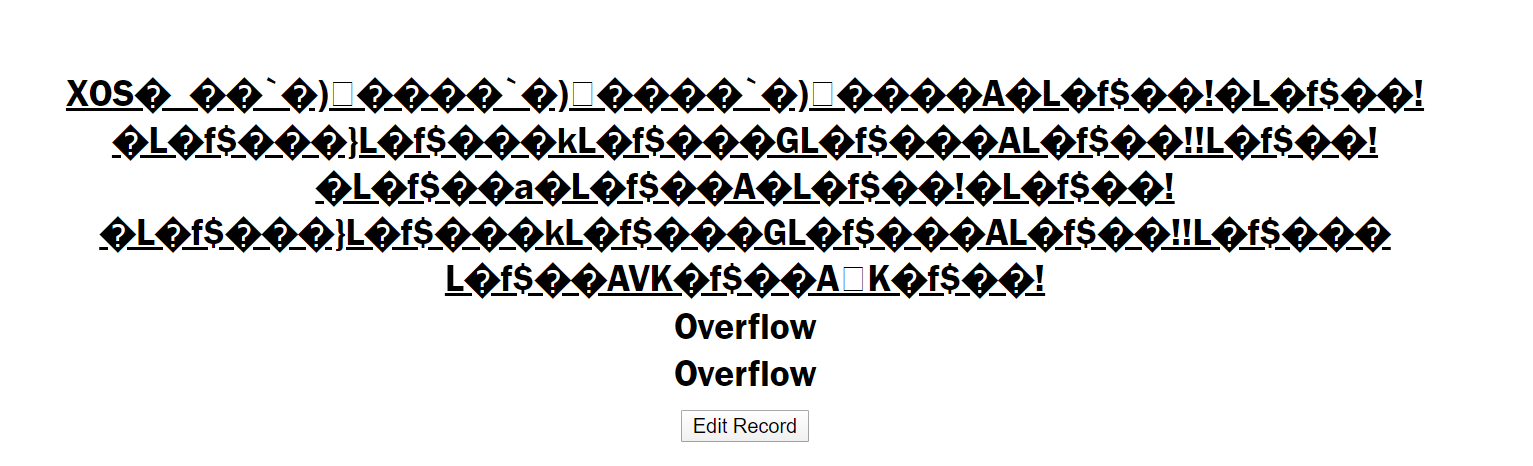
BufferExploit takes advantage of the buffer used when inserting a new record on the user page in the weak site. A Curl POST is sent with a number value, allocating empty space instead of saving a string value. Overflow memory is then accessed on the user page as a new record with the name being the data.

Associated Security Feature: Removal of the buffer and the use of placeholders

*Directions to use bufferExploit.sh:*

1. The data for the Kyle profile has been hardcoded into the attack code (the attacker needs a profile and their profile ID)
2. Run the script and access memory data at flip3.engr.oregonstate.edu:6061/user/Kyle&12345
3. Repeatedly run the script to get chunks of memory to later decode and decipher

To use the script: ./bufferExploit.sh



**Denial of Service (DoS) Attack**

Denial of service attacks involve consuming the resources of the server to a degree that it affects the server accomplishing other tasks. The following attacks illustrate how resources could be consumed maliciously to slow down a server.

Associated Security Feature: non-html AJAX calls to prevent script execution via forms

Directions to use Fork Bomb:

1. Make sure that the active internet browser will not block pop ups
2. Add a new record, using the following as the value for the record name (change the URL to link to the active user profile):

>"&amp;gt;<script>

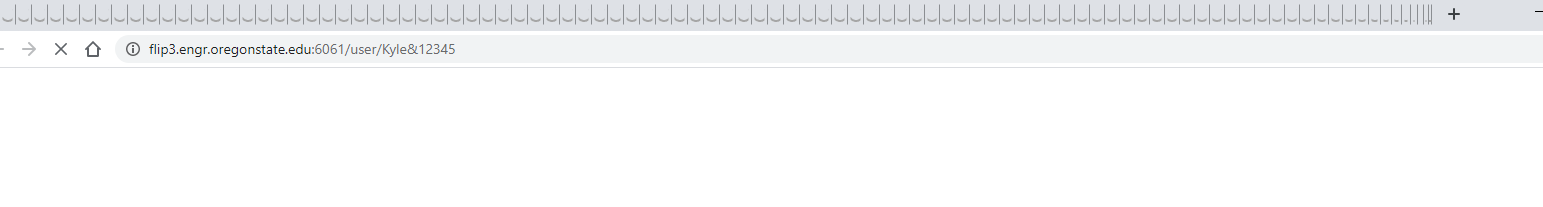
function fork() {

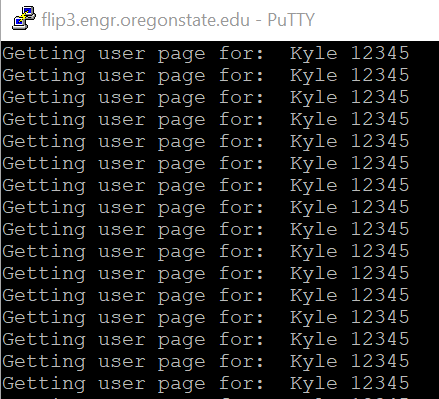
const win = window.open("http://flip3.engr.oregonstate.edu:6061/user/Kyle&12345");

setTimeout(fork(), 1); }

fork(); </script>

1. Select the name of the record from the record table to execute the fork bomb





WARNING: This will slow down the user computer. This does not bring down the school server (thankfully) rather it illustrates how to slow down a weak server. The Kyle account has this record already saved and ready to execute.

*Proof of Concept: dosExploit.sh*

The script dosExploit.sh opens multiple concurrent POST requests to save new records. This has been capped at 10, although many more could be opened at once. Curl allows the limit rate for sending the record to be set to a very slow rate and kept alive once complete, both of which consume server resources. A weak server will be slowed down while waiting for the requests to finish and close.

WARNING: This will slow down the user computer, if not crash it outright! Again, this does not bring down the school server, rather it illustrates how to slow down a weak server.

To use the script: ./dosExploit.sh

**Additional Exploits**

In addition to the attacks outlined above, additional attacks we have addressed include:

* Use of stolen credentials (2FA)
* Over-the-shoulder password theft (Hidden password characters)

**Software and System Functionality**

The Secret Keeper project makes use of the client-server model to provide functionality to the user. The client component interacts with the user, requesting data (user and record) when appropriate. The server component (Node.js) connects to the MySQL database and provides communication between the database and the client component. The Handlebars templating language is used to create HTML pages with data provided by the server.

An example of how the three levels (client, server, and database) interact can be illustrated using the login service. A user is asked for their username and password by the client component. Once provided, the data is sent to the server via a PUT request. The server uncouples the important information from the request and sends a MySQL query to the secret-keeper-project database. The database responds back to the server and, in the case of login, sends the total count of the username and password combination in the user table. The server then sends the count to the client component in a JSON-formatted message. The client component uses JavaScript to read the message and responds accordingly, advancing the user to the user page (or the 2FA page in the secure version) if there is a count of 1 or, if the user/password combination is incorrect, displaying an error message in the HTML.

Data is encoded into the URL in both the weak and strong sites - specifically the username and password. The server component uses this, as well as additional data included in JSON-formatted messages, to fulfil the appropriate GET, POST, PUT, and DELETE requests. The weak side uses neither encoding nor placeholders, so data can be entered directly into the URL and it will be inserted by the server into MySQL queries.

Unique encryption functions were created and used in the secure site, which allowed data stored on the database to be encrypted in the event that it is maliciously accessed. Both the client and server have access to variations of the encryption function, with the server also having access to a salt variation of the encryption. The non-salt variation creates a unique salt in the encoding process and attaches the salt to the front of the encrypted string. This is later taken off and used in the decoding process. The added salt is encoded so as to not be obvious. The salt variation takes a salt from the MySQL salts table via the user-provided username. The non-salt variation results in unique strings for each encoding, whereas the salt variation results in the same string for each encoding.

To combat brute-force attacks, a rate-limiter was created and used on the PUT request to log in to the user page. The rate-limiter uses a map which stores usernames as a key and an attempt number with timestamp as the value. When a user sends too many requests with the same username in a short time period, the MySQL query is delayed from being completed. Additionally, the login button on the home page is disabled while the delay completes. The delay increases with each immediate failure, quickly resulting in unfeasible times for brute forcers.

An Admin account was created for both secure and weak sites. Upon logging in at the home page, the Admin is taken to the user page where they have access to a unique admin table. The admin table is populated with all user accounts and the ability to delete any account. The questions and comments submitted by users at the FAQ page are displayed below the users with the option to respond to or delete the question.

Two-Factor Authentication (2FA) makes use of the Nodemailer module to send users an email with a randomly-generated pin needed to proceed to the user page. For testing purposes, the pin has been set to print on the 2FA site page. As with the rate-limiter, the generated pin is mapped to the user’s credentials. 2FA protects user accounts which have had their credentials stolen, adding an additional security layer which must be compromised to gain access to the user’s account.

The exploit scripts were written in Bash and use Curl as the means to communicate with the server. Curl can send requests directly to the server component, bypassing any client components which may have security features. For example, a buffer was added to the server when creating a new record name on the weak site. The buffer can be exploited by Curl to allocate large chunks of memory, which can be decoded. This potentially reveals private data or code meant to be hidden. The buffer exploit is not possible via malicious interaction with the client components.

**Software Languages and Tools Used**

The Secret Keeper web application was created using the following languages, libraries and tools:

*Client Components: HTML, CSS, JavaScript, AJAX calls*

On the client side, HTML and CSS were used to form and style the visual implementation of Secret Keeper, and JavaScript and AJAX were used for client-side interactivity.

*Server Components: Node.js with Handlebars, JavaScript*

Node.js/Handlebars and JavaScript were used for server-side logic implementation.

*Database Components: SQL via MySQL*

Database CRUD (create, read, update, delete) actions were implemented using SQL.

*Exploit Components: Bash with Curl, JavaScript, HTML & SQL code injections*

These were used for the various exploits outlined in the “Exploiting Site Vulnerabilities” section.

*Modules: NodeMailer*

NodeMailer was used to implement the email-sending functionality in support of 2FA.

Note: When possible, the Secret Keeper project team chose to write their own code instead of using modules. Increasing understanding of functionality was deemed more important than adding complexity.

*Images and Graphics: Microsoft Paint 3D*

The background of the Secret Keeper “shhh” logo was edited using Microsoft Paint 3D to match the white background of the site.

*Version Control: GitHub*

Secret Keeper was created with heavy use of GitHub for version control. After creating the barebones version of Secret Keeper, we created two branches off of master: secret-keeper-secure and secret-keeper-not-secure. Using these branches, we were able to implement exploits and security features separately and pull in common code via master when necessary. [Click to navigate to the Secret Keeper repository](https://github.com/dixonky/Capstone).

*Collaboration: Slack, Google Drive & Google Docs*

Communication is crucial in any group project. Throughout the term, we communicated nearly daily via Slack regarding new implementations, project ideas and questions. A project-specific shared folder in Google Drive containing relevant Google Docs allowed us to create the initial project outline, midpoint check-in and final outline collaboratively.

**Team Member Contributions**

Team members worked collaboratively on all parts of the project. The distinction below represents the component lead, not the sole contributor.

**Katie Young**

Katie brought the project to life with interactivity and a pleasing - and consistent - user experience. This included implementing the styling and interactivity of the various forms found throughout the Secret Keeper website, editing the site’s “shhh” logo and eliminating clutter using hidden divs for items such as password-validation notices. On the secure site, she implemented password validation criteria for the user creation page and the update-account functionality on the user page. She also hid the values of passwords as they were entered by presenting them as dots on the screen and tackled two-factor authentication. On the not-secure site, Katie crafted an AJAX vulnerability, allowing her to inject an alert and an HTML link into a text field. This prompted the creation of the FAQ page for a more public (and, thus, more real-world) demonstration on how an injection might be used to impact everyday users. Although her initial alert message, “I win!”, was replaced with a more professional-sounding, “Warning: This site is vulnerable to an attack”, the former still holds true.

*TL;DR: Site design, User credential criteria, HTML & CSS security, HTML injection, FAQ, 2FA, hidden passwords*

**Kyle Dixon**

Kyle paved the way into the great unknown for the duo. This included proposing the group’s timeline, initial website and GitHub setup, database setup and implementing the site’s C-R-U-D functionality using MySQL and AJAX calls. Additionally, Kyle spearheaded encryption and decryption, creating a new database table for salted data and effectively moving information throughout the site in a more secure way. After identifying the SQL injection attack outlined above, Kyle moved on to successfully create brute-force (using both list-based and randomly generated passwords) and buffer-overflow attack scripts. He also implemented the rate-limiter used to counter brute force attack attempts, adding increasingly-long breaks between allowed log-on attempts when incorrect credentials are entered. Finally, Kyle crashed his own computer in the process of implementing the Denial-of-Service attack. Talk about taking one for the team!

.

*TL;DR: MySQL calls, AJAX calls, Encryption, SQL injection, Buffer Overflow, Brute Force, DoS attacks, Rate-limiter*

**Deviations From Original Plan and Future Feature Ideas**

The Secret Keeper group lost a team member early in the process, which reduced the number of security implementations and exploits that could be completed. For example, special characters were excluded for passwords as there was problematic interaction with the MySQL queries. Given more time, special characters would be introduced to passwords as a criteria to increase password security.

One site feature that was not a part of the original plan is the implementation of the frequently-asked questions page. After identifying a way to inject HTML into a field, we realized that it didn’t make sense to have this vulnerability in a non-public space. We implemented the FAQ page to address this, allowing us to inject scripts and links in a publically-available forum.

Additional future features might include the following:

* All data at rest should be encrypted, but the usernames in the salt table are not encrypted in the strong site. Future additions should involve encrypting these names with traditional encryption methods.
* The database table which stores the FAQ and responses has a spot for the user email of whomever sent the question. Nodemailer could be used to let users know when their question has been addressed.
* Defensive features that reduce the impact of a successful attack would be beneficial. For example, interaction with the user via email regarding changes to a user’s account credentials might allow the user to catch and address an attack early.

**Conclusion**

There are many security features addressed in OWASP’s top ten security risks list which can be implemented to mitigate malicious actors. The Secret Keeper project sought to illustrate these features using two versions of a record storage site with different security protections. Exploits, created based on the absence of security features in the weak site, were built to gain access to private data and wreak havoc on software functionality. Simple features, such as use of placeholders when passing data between component levels and encryption of data at rest, stop the majority of examined exploits. While Secret Keeper is not 100% secure, it shows how effective taking simple precautions can be towards protecting users and curbing malicious actors.

**Citations**

* Rate Limiter: <https://levelup.gitconnected.com/rate-limiting-a0783293026a>
* Encryption: <https://www.sitepoint.com/how-to-build-a-cipher-machine-with-javascript/>
* XSS: <https://codepen.io/njmcode/pen/qddaeB>
* Modulo Bug: <https://web.archive.org/web/20090717035140if_/javascript.about.com/od/problemsolving/a/modulobug.htm>
* Curl:
  + <https://curl.haxx.se/docs/manpage.html>
  + <https://linuxhint.com/curl_bash_examples/>
  + <https://linuxize.com/post/curl-command-examples/>
* Buffer: <https://snyk.io/blog/exploiting-buffer/>
* Brute Force: O’Connor, TJ. *Violent Python.* Syngress, 2013
* Bash: Cannon, Jason. *Shell Scripting.* 2015
* OWASP: <https://cheatsheetseries.owasp.org/>
* Council of Economic Advisors (CEA, 2018): <https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf>
* 2FA: <https://www.w3schools.com/nodejs/nodejs_email.asp>
* Recent vulnerabilities: <https://www.kb.cert.org/vuls/>
* “Shhh” Logo: <https://www.pinclipart.com/pindetail/ioRhxh_noise-clipart-quiet-place-quiet-png-transparent-png/>