Team Kappa
Ryan Webber
Eniola Abdul

Our project is a web crawling interface written in python that can be directed at an HTML website. Once this is done it will crawl that page, autonomously search for links and crawl those until either runs out of content to crawl or a constraint defined in its initialization has been met. Once done the crawler presents the user with the ability to bruteforce the login page of that website, if it was able to find it, based off of the credentials it retrieved from crawling the page, and report back if any of the credentials that it found were successful in bruteforcing the page.

When we first began, we decided to divide up the work into two main components, the first would handle the crawling and parsing of a page given the HTML content, and the later would handle sending valid HTTP requests to the pages that we wanted to crawl and the login page we want to attack. These became the two primary objects of our project and hold most of the functionality. This was paired with a GUI class that utilizes them both and provides an easy interface for the user to interact with and receive the information that they are looking for.

Though it was not done first, it is probably best to begin with the requester that we created. Like many other requesting services, the goal of our requester object was to properly send GET and POST requests to various web servers. It would also be able to react to the responses that it receives and make decisions on how to continue the transfer of information. The first design decision we made was to send our request headers over HTTP 1.1. This was because

HTTP versions that succeed this one assumes that the entity reading its data can properly parse a response that uses chunked encoding. Due to some of the difficulties we had parsing this information, we agreed that it would be better if we used a version of the protocol that did not support this encoding so that we could more easily move forward. This leads to our first assumption being that whatever web page we're requesting can serve content over more dated versions of HTTP. Continuing from here we were able to utilize sockets to properly send and receive information to any HTTP domain of choice. To build upon this, we had to add the ability for our requester to maintain some general forms of state when communicating with a web. We learned that many modern websites will not serve any types of post requests that don't have the relevant cookies attached. Without this it would be impossible for our requester to be properly served a page representing a successful login, or even recognize that the page was one of a successful login. We were able to do this simplistically by constructing a simple cookie-jar for the requester that it can append to its post requests to mimic a session. Since the purpose of the requester's post methods was for the eventual credential brute forcing, we assumed only the initial set of cookies given would have any value for this goal, so we generally only worked with the cookies from the initial get request when attempting to brute force a page. Our crawler also assumes that any response content will be HTML that is formatted correctly and can be parsed with Beautifulsoup.

When implementing our requester, we also decided to make a new socket for every page crawled because there were issues with the sockets timing out and pseudorandom intervals. We came to the conclusion that the complexity of handling these timeouts was not worth the speed lost by creating multiple sockets. Another important decision we made was to to take the entire

HTTP response as properly formatted HTML/plaintext pages. This assumption made it possible to search HTML content and robots.txt indiscriminately. The disadvantage of the approach is that if the response was improperly formatted or garbage was sent after the response, we would read this as userproperly formatted pages, introducing keywords that were not present on web pages.

In addition to the state generated from session cookies, other forms of understanding were necessary for our requester to be a reliable tool to crawl and brute force with. In the event that the response was a redirection, the GET and POST methods had to be able to handle some of the various error codes that these pages can return. Errors in the 4XX and 5XX range were mostly ignored by simply returning to the caller with an error value. 3XX redirects however needed were directly handled using helper parsing functions. We of course assumed that all of our responses would have valid HTTP response headers and were able to use the format of the header along with some regular expressions to easily classify the type of response we received and react accordingly. In the case of a redirect we tried to establish a new connection with the updated location and send another GET request to that page and return its contents instead. Some of these were simple but others ended up needing their connections to be recreated at the socket level because they were on subdomains or were redirects to HTTPS versions of the page. First, when we decided to also use the open_ssl libraries to implement support HTTPS, it allowed us to easily handle when a page asks the user to redirect to the HTTPS version of their website which allows the user the convenience of not having to explicitly indicate that they are visiting an HTTPS site. In addition, this also proved to be extremely important when trying to brute force post requests to a page. This was because almost all login form submissions were met with

redirections, so handling this functionality allowed us to better determine whether a brute force attempt was successful.

For the crawler/bruteforcing section, we decided to make several functions as opposed to a single, all controlling one to make the project more modular and easier to debug. In addition, splitting up the functionality also made it substantially easier to convert or command line program into a GUI. In our crawler class we first have flags that are user set to determine how the crawler will run. From the GUI, the user selects breadth first/depth first searching, whether to search subdomains, robots.txt, and the max depth and amount pages to search. Once these are selected the a seachInit function is called, which selects which searches to do and for how long. For breadth first search we used a queue to hold elements due to its first in, first out nature and for depth first search we chose a stack since it is first in, last out. Since subdomain and robots.txt searching are treated as auxiliary searches, it will use whichever data structure chosen for the main search. Since the only thing differing between the main, subdomain, and robots.txt search is what page is being checked, we decided to have all three parse pages the same way. Our parsing is done by using BeautifulSoup to parse the response from our requester as an HTML file. From there we create a list comprised of links found on the page by finding all lines that contain ‘href’ and ‘http’. By using this filter, we are also able to find links that may been hidden from the user but still on the page. Lastly, we check if the page has a login form by searching for form fields that contain a ‘password’ input. If one is found, we save the the names of the login fields in global variables to later be called by the bruteforcer. When a search is completed we return the data wrapped in a custom object that is stored by the respective data structure. We heavily

debated making the crawling multi-threaded but went against is because process of crawling was very short and there would not be a significant increase in adding more threads.

The crawling will end when any of the three cases occur: 1) When the maximum amount of pages increases, 2) When the only remaining links are past this maximum depth, or 3) If there are no more pages left to crawl. When this finishes we begin take all the text received, split them by newlines, and strip every word of punctuation. We then generated the leet speak and reverse of each word and stored them along with the original and a final list of words to be used as passwords. We debated making a blacklist of characters to strip such as "!" or "?" but since these are valid characters when making a password we only filtered characters that we both thought would never be seen in a password. When making this we also had a min word length to be used as a password to reduce the amount of unnecessary tries our bruteforcer would make but since many websites still don't heavily enforce a minimum password length we decided not to filter too aggressively by length.

Following the crawling and parsing of web pages, the list of passwords are compiled and sent to our requester object as parameters to it's bruteforce function. In our implementation of bruteforcing a login page we first request the login page from the host. We then separate the list of passwords into five sections and create a thread of the function ATTACK to handle each partition. The ATTACK function exhausts its list of passwords on a login form until it either runs of passwords or succeeds in logging in. Within the requester is a default username(root) that ATTACK will pair with the passwords but the user can also add any additional usernames through the GUI. We detect a successful login by first detecting if the POST request causes a redirection. If it does, we handle the redirection as stated earlier, retrieve the page we were

redirected to and check if it is the different from the login page. In this case, we assume that on an unsuccessful login, we will be remain on the original login page. The threads communicate with each other through a global flag that is turned on when any of the thread succeed. It was decided that we would stop on finding the first successful credentials instead of trying all keywords because as more pages are crawled, it would take an extremely long time to attempt the entire list. We also assume that there is no password field on the page resulting from a successful login. In addition, we accounted for the need of nonces in some websites by including nonces in the POST request while bruteforcing. Because this process, even when multithreaded, can take a long time due to delays from servers and socket connections, we decided to print some intermediate information in the console while the GUI hangs to let the user know that code is still being executed and requests are still being made. This output shows in real time passwords that are being attempted. When all of the passwords are exhausted or the bruteforcer thinks it found a successful set of credentials, the bruteforcer will terminate, display how many credentials it tried, whether or not it was successful, and if successful, the credentials that were able to give the crawler access to the system.

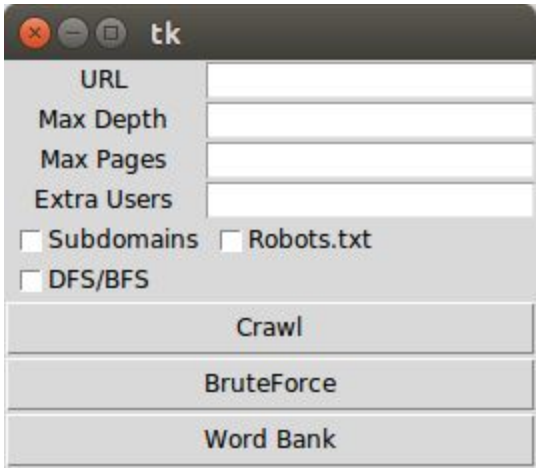
Our group, Team Kappa has had the luxury of operating in very close proximity of one another. It is for this reason that the delegation of responsibility for individual tasks within the project weren't rigid as we were generally able to frequently discuss and make contributions on particular tasks in the project. That being said, for certain parts one particular member generally tackled a large amount of the work because they decided to do it first or they felt a bit more comfortable. Within the crawler, Eniola handled most of the search functionality and the building of the set of words that would eventually be passed to the inbuilt parser to structure for

use by the bruteforcer. The inbuilt parser was mostly handled by Ryan. Moving to the requester, Ryan handled the majority of the HTTP GET request while Eniola did the majority of POST. Eniola created the the logic for the bruteforcer which Ryan then decided to make multithreaded to help improve performance. Ryan also helped build by building a basic vulnerable php web page that we could test our crawler with.

In conclusion, our team was able to learn in depth the format of how web servers feed their content to clients. This also proved to be what made this project difficult because we learned that many of the ways that hosts choose to interact with and implement their servers can vary. Many of the assumptions we made in the beginning later on turned out to be completely incorrect and we continuously found more and more obscure cases that we tried our best to handle. The web as a whole was much less standardized than we thought it would be. This puts the burden on clients like web browsers and our crawler to be able to react to different types of responses from different types of web servers and be able to interpret and handle them correctly. Trying to write modular and efficient code that has to be able handle all of the peculiarities and inconsistencies of the web proved to be a difficult task but we were able to put together something that was functional and capable.

How to Run:

1) From terminal run: python2.7 GUI.py. The following screen should appear:



2) Enter the URL, max depth, and max pages in the respective text fields.

3) Enter additional users that are comma separated.

4) If crawling subdomains and Robots.txt is desired check their respective boxes.

5) By default the program runs using DFS search, to switch check the “DFS/BFS” box.

6) When configuration is finished select “Crawl”.

7) After the crawling is complete, selecting “Word Bank” shows all the words that will be used as possible passwords for bruteforcing.

8) To begin bruteforcing, click the “Brute Force” button. If a login page was found during crawling, it will proceed. If not, a message will appear saying the attempt failed due to a login page not found.