CSE 361 Web Security Final Project Web Crawler + Form Bruteforcer Report

Team: RSA

Jian Chen 110296913

Zhi Zou 109825816

11/3/2017

1. Introduction

Web crawler, also known as a web spider. It is a robot that automatically browses around the world wide web and collects data. Each web crawler has a unique task and a unique way of processing the data collected. We can do many things with the data collected. However, many companies nowadays have already come up with various defenses to protect their data from being collected by the web crawlers. This will indeed increase the difficulty of collecting data from the web, but there are always ways to bypass these defense mechanisms. The task of our web crawler is to collect visible text on the website and extract individual word to build a database that contains passwords. Whenever we encounter a login page while crawling the target domain, we use the password database to crack the real password by trying each password in the database. We also set up our own websites to test the functionality of our web crawler.

2. Design Decisions

- Programming Language Used: Python
- We used Breath-first search when crawling a website.
- Maximum page to crawl: 100 (11 pages for our testing website)
- Maximum depth of pages to crawl: 10 (2 levels for our testing website)
- Libraries that we used: re, urllib, tldextract, bs4, ssl, socket
- Programming paradigm: object-oriented programming

The web crawler implementation is separated into different modules. The structure overview is shown in Figure 1.1. The crawler_main will be the core of the web crawler. The dataset class will be responsible for updating and retrieving data collected. The html_processer will handle webpage downloading and data uploading. The html_parser will process the data by extracting words and links retrieved from the web page downloaded. The self_get_post will include the implementation of custom GET/POST request. The subdomain class will extract url paths from common subdomains and robots.txt. The url_list will serve as an url manager to record the processed url and the urls to be processed.

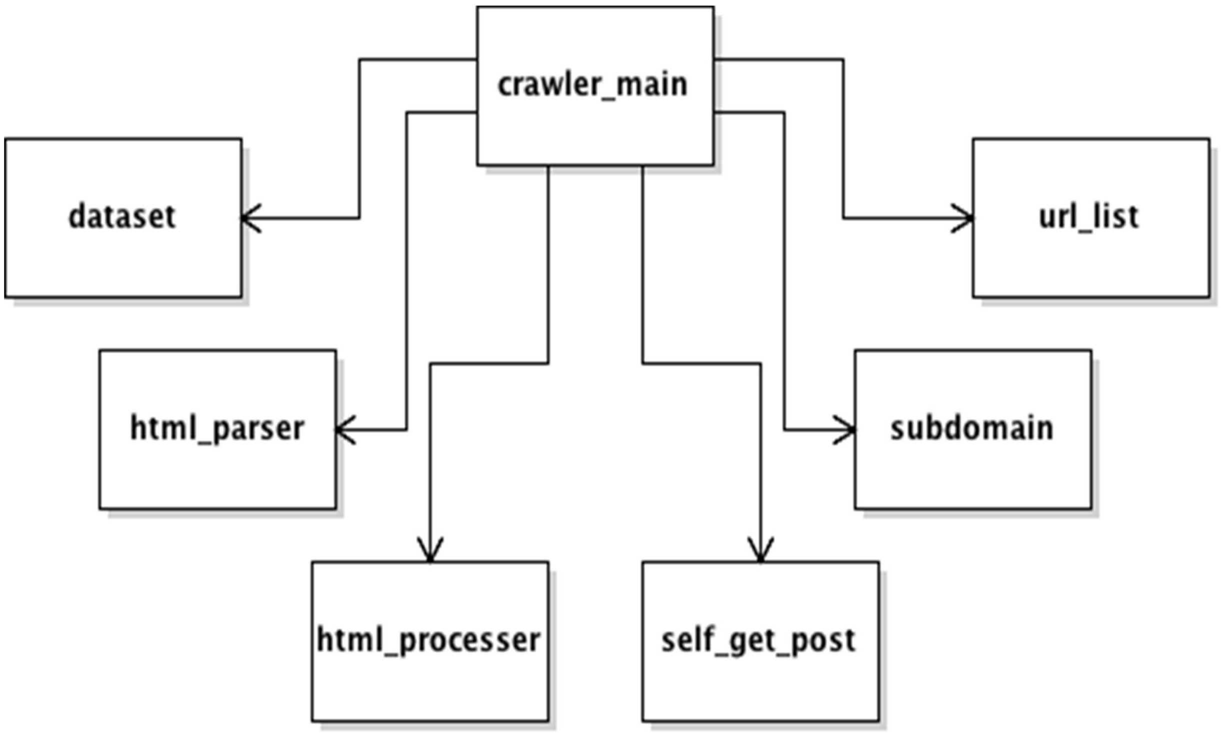


Figure 1.1 Web Crawler Structure Overview.

3. Implementation Detail

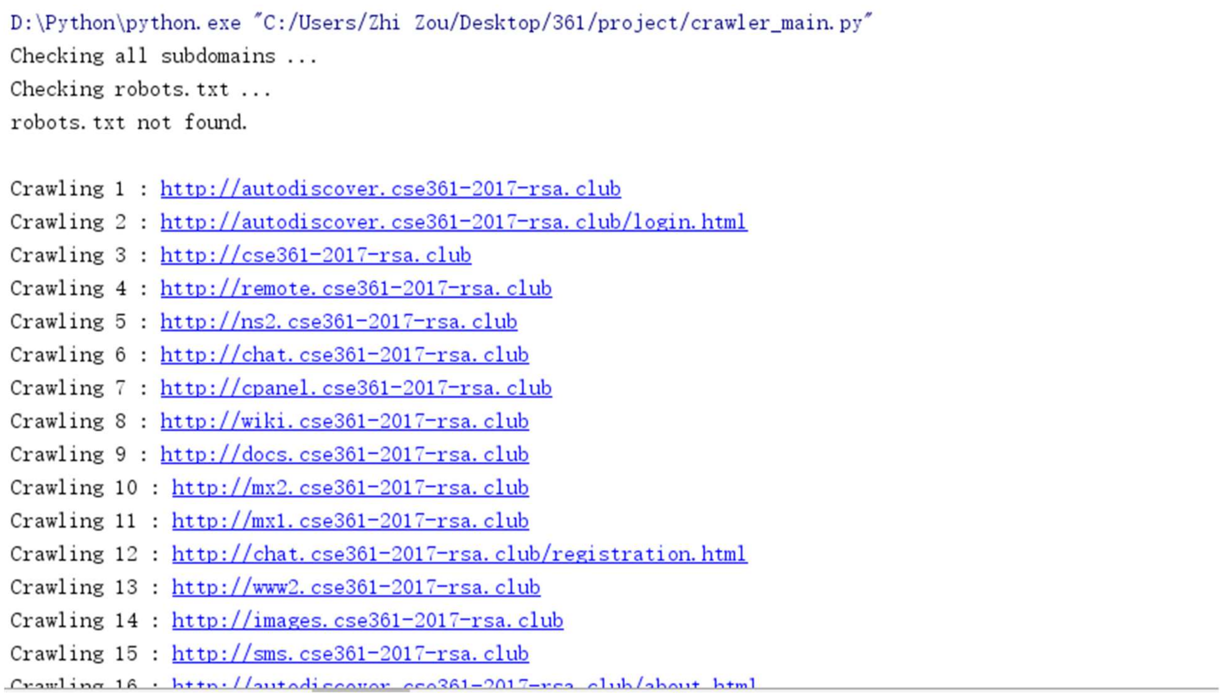
a. Jian's Part:

- Jian set up the testing website.
- Jian coded the overall structure of the web crawler.
- Jian implemented the data mining part including collecting new links and html parser for the possible passwords.
- Jian implemented collecting the subdomain of the target domain.

In order to test the functionality of our web crawler, I set up the testing website. The website includes a total eleven pages such as, about.html, contact.html, login.html, mail.html, news.html, random_article.html, random_article1-3.html, and registration.html. All of above web pages except the mail.html belong to the same domain. The mail.html will be a subdomain of our main domain and the content of it will be shown when the user types “http://mail.cse361-2017-rsa.club”. The purpose of creating mail.html is to test whether our web crawler will correctly identify possible subdomains of a domain and collect data from that page. Besides the case of checking subdomain, I also covered other test cases. For example, I include external links and links that are within the same domain in the about.html. In the contact.html, only external links are included in the web page. Other web pages will either have no links or some external links in them. Figure 1.2 shows the process of determining subdomains and links that we have not crawled before in the robots.txt. However, there is one problem in the testing website which is that all the subdomains will point to the same website except the mail subdomain, which I explicitly provided different web content. Another important test case for the web crawler will be differentiating login form from registration form. I set up the login page, registration page and corresponding php script when a user clicks submit button. In this project, we do not have a database for storing usernames and passwords. We only include one user with username “admin” in the php script and check if the password entered is correct or not. Nonetheless, there is a limitation of differentiating log in form and registration form, which will be discussed in the later section.

The subdomain url is implemented by appending common subdomains in front of the target domain and reformatting the url to a valid url. After the new subdomain url is constructed, we then called custom get request to identify whether such subdomain exists or not. If it exists, then we crawl to that web page and collect the data.

In the html_parser module, the first thing we do is to analyze the data collected from the web page. After we use the beautiful soup library to extract texts from the web page, we then trim off the extra white spaces. The texts are then split into words, and we add the variations of each word such as lower case, upper case, reverse and leet-speak version of the word into a word list, which will later be used as a database to crack passwords.



```
D:\Python\python.exe "C:/Users/Zhi Zou/Desktop/361/project/crawler_main.py"
Checking all subdomains ...
Checking robots.txt ...
robots.txt not found.

Crawling 1 : http://autodiscover.cse361-2017-rsa.club
Crawling 2 : http://autodiscover.cse361-2017-rsa.club/login.html
Crawling 3 : http://cse361-2017-rsa.club
Crawling 4 : http://remote.cse361-2017-rsa.club
Crawling 5 : http://ns2.cse361-2017-rsa.club
Crawling 6 : http://chat.cse361-2017-rsa.club
Crawling 7 : http://cpanel.cse361-2017-rsa.club
Crawling 8 : http://wiki.cse361-2017-rsa.club
Crawling 9 : http://docs.cse361-2017-rsa.club
Crawling 10 : http://mx2.cse361-2017-rsa.club
Crawling 11 : http://mx1.cse361-2017-rsa.club
Crawling 12 : http://chat.cse361-2017-rsa.club/registration.html
Crawling 13 : http://www2.cse361-2017-rsa.club
Crawling 14 : http://images.cse361-2017-rsa.club
Crawling 15 : http://sms.cse361-2017-rsa.club
Crawling 16 : http://autodiscover.cse361-2017-rsa.club/about.html
```

Figure 1.2 The Web Crawler is Determining Links for Subdomains and Links from the Robots.txt.

b. Zhi's Part:

- Zhi implemented our own GET and POST method (using socket).

- Zhi implemented checking login form and set up the proper login information.
- Zhi implemented checking the response of the GET and POST requests.
- Zhi implemented collecting all urls from robots.txt.
- Zhi combined the codes and finished final testing.

Before Jian finished the basic structure of the crawler, I started to implement self_get_post.py file which is mainly about the self-get and post request methods. The first method I implemented is the “get_method(URL, cookie_list).” My goal was to create the method so that we can create a socket using socket lib and send the socket with the GET request header to the URL server which we want to crawl. There will be two different protocol types. One is HTTP and another is HTTPS. Because the port number I need to connect to the domain is different, I have to cover the protocols individually. For HTTPS protocol, we need to wrap the socket with SSL or TSL. I used the wrap_socket method in the SSL lib when the URL entered using HTTPS protocol. I extract the host from the URL first, then send the header “GET + url_path + HTTP/1.1\r\nHost: url_host\r\n……” to the URL host side. The return value of this function is a responses list which contains all the responses from the server. If the return code is 200, I return the last index which should be the page contents. If the return code is 3XX, I scan the list and find response header “Location: ”, and I repeatedly resend a request to the URL follows by that header until the return code is not 3XX. If the return code is 4XX or 5XX, the method returns None so that whoever uses the method know whether the URL input is correct or not. I didn’t throw an exception when the return code is 4XX or 5XX because whoever uses the method may not catch exception so that the program would be terminated. For the second parameter, cookie_list is a cookie set need to send to the server if needed, and I would explain it further later.

The second one is the post_method(URL,data_dict,referrer_page). The goal was mostly the same as the get_method except it's a post request. The data_dict is a dictionary of data which need to send to the server in the post request. The referrer_page corresponds to the Referer header in the post request. The structure is also like the get_method. One different case is the page redirection. Frequently, a lot of websites will redirect you to a new web page when you log in and set your browser cookie with the proper section id. In addition, for the cookie_list in the get_method, it's a cookie list generated from the post request. If we don't include it, we would never be able to log on the page. The redirection of post request could only be using get request with the correct section id in the cookie. The post header I sent is like "POST + url_path + HTTP/1.1\r\nHost: url_host\r\n……\r\n\r\n input1=a&input2=b".

The next thing is to combine my method into the total project when Jian finished the crawler structure and finish mining data. Before I test the collected password, I have to get all URLs from the robots.txt file. Because there are Allow ones and Disallow ones, I searched it on the Internet and found out we are not allowed to crawl the Disallow ones. I use the self_get_post.get_method(URL + "/robots.txt",None) method to get the data from the file, and then use re lib to compile the Allow ones only. Use the new path and the original URL to generate new URL and add it to unprocessed URL list. Because we parsed all inner page links first because of mining data, I included this method before parsing the data. When I finished the robots.txt part, the next thing needed to be covered, which is also the most important one, generates a function to check login form in the web page and process the login request.

For testing a web page contains a login form or not, we discussed several different approaches and finally choose one most reasonable. First, we simply thought the easiest way is to check whether there is a form data which contains an action, but we realize there is another action variable usage for the form tag like selection function. The second thought is that to check whether there is any input tag in the form tag and whether there is a submit option. However, it's not correct either. If we inspect the HTML page of a register form, it would match that requirement. It's not the case we want. Then, we inspect most login pages. The one common thing we found is that almost every login form contains an action variable, have one or two text input tag inside of the form tag and one and only password input tag. Indeed, our final decision is to check whether a web contains a form tag which has action variable, one or two text input tag and exactly one password input tag.

For using the data, that we mined from the URL, to test with a specific user_agent name, we need to check the response of the post method for each account password set. There are also several considerations about whether it successfully log in or not. We first considered testing whether the response return code is 3XX or not, which means whether the website redirects us to a new website. However, it's incorrect. We cannot assume the correct password would take us to another website nor the incorrect password wouldn't take us to another one. We came up with a much more reasonable way to solve this problem. We found that when you are successfully login to a website, the identical login form wouldn't be visible or might not even exist. We think that we could benefit from this property, to check whether there is an identical login form tag before and after we processed a post requests with each password set. If there is one, the password is incorrect, otherwise, we find the correct one.

There is another thing I found when I implemented the login request. The URL we posted to the server is not the original website we are at. The URL contains login form should be the referrer

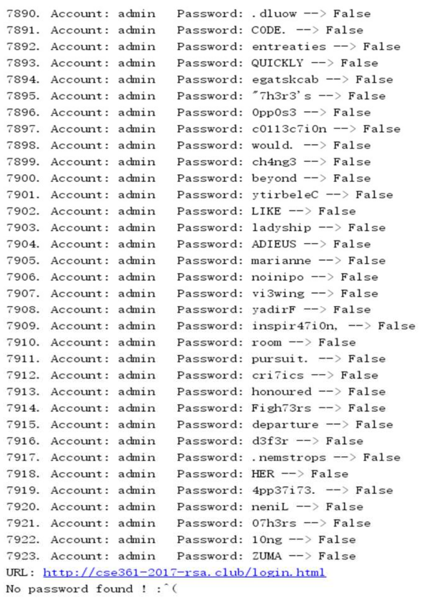
URL for the post method. The correct URL we want to post should be the combination of the domain and the new path created by the "action" variable.

When our crawler processes the login, if we found one correct password, I directly return that one because in most cases there is only one password corresponding to one account, and we don't need more than one correct password. It would save much more time by not processing the rest of the password. It would make our crawler more efficient.

4. Evaluation

    a. Performance

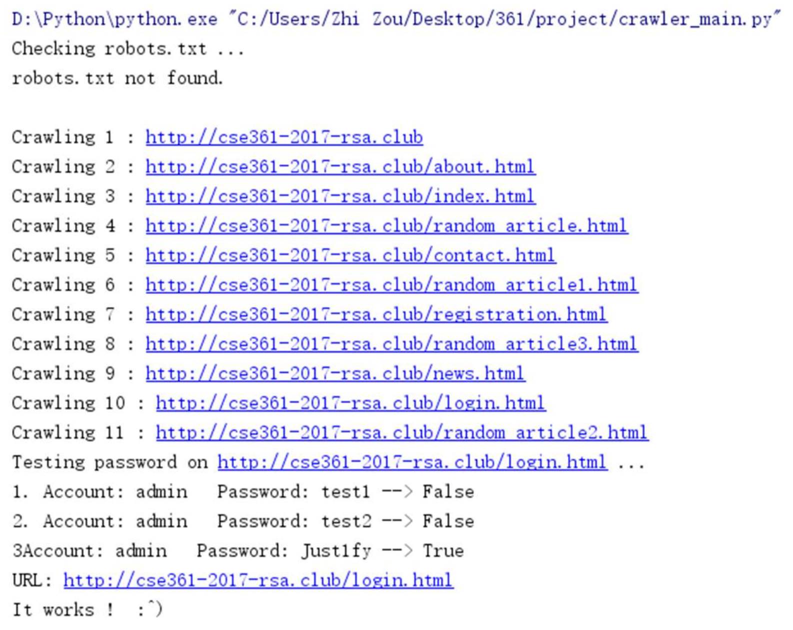
    Since the testing website that we set up has no defense mechanism against login form brute forcer, we can try to login with different passwords continuously. The time it takes to process 8000 passwords will take approximately 10 minutes. Figure 1.3 shows that the crawler cannot login as a user because there is no correct password in the current password database.



```
7890. Account: admin   Password: .dluow --> False
7891. Account: admin   Password: CODE. --> False
7892. Account: admin   Password: entreaties --> False
7893. Account: admin   Password: QUICKLY --> False
7894. Account: admin   Password: egatskcab --> False
7895. Account: admin   Password: "7h3r3's --> False
7896. Account: admin   Password: 0pp0s3 --> False
7897. Account: admin   Password: c0113c7i0n --> False
7898. Account: admin   Password: would. --> False
7899. Account: admin   Password: ch4ng3 --> False
7900. Account: admin   Password: beyond --> False
7901. Account: admin   Password: ytirbeleC --> False
7902. Account: admin   Password: LIKE --> False
7903. Account: admin   Password: ladyship --> False
7904. Account: admin   Password: ADIEUS --> False
7905. Account: admin   Password: marianne --> False
7906. Account: admin   Password: noinipo --> False
7907. Account: admin   Password: vi3wing --> False
7908. Account: admin   Password: yadirF --> False
7909. Account: admin   Password: inspir47i0n, --> False
7910. Account: admin   Password: room --> False
7911. Account: admin   Password: pursuit. --> False
7912. Account: admin   Password: cri7ics --> False
7913. Account: admin   Password: honoured --> False
7914. Account: admin   Password: Figh73rs --> False
7915. Account: admin   Password: departure --> False
7916. Account: admin   Password: d3f3r --> False
7917. Account: admin   Password: .nemstrops --> False
7918. Account: admin   Password: HER --> False
7919. Account: admin   Password: 4pp37i73. --> False
7920. Account: admin   Password: neniL --> False
7921. Account: admin   Password: 07h3rs --> False
7922. Account: admin   Password: 10ng --> False
7923. Account: admin   Password: ZUMA --> False
URL: http://cse361-2017-rsa.club/login.html
No password found ! :^(
```

Figure 1.3 Result of Login Attempt Failed.

We also tested the situation where we have the correct password in our database. Figure 1.4 shows the result of successfully log in as a legitimate user.



```
D:\Python\python.exe "C:/Users/Zhi Zou/Desktop/361/project/crawler_main.py"
Checking robots.txt ...
robots.txt not found.

Crawling 1 : http://cse361-2017-rsa.club
Crawling 2 : http://cse361-2017-rsa.club/about.html
Crawling 3 : http://cse361-2017-rsa.club/index.html
Crawling 4 : http://cse361-2017-rsa.club/random_article.html
Crawling 5 : http://cse361-2017-rsa.club/contact.html
Crawling 6 : http://cse361-2017-rsa.club/random_article1.html
Crawling 7 : http://cse361-2017-rsa.club/registration.html
Crawling 8 : http://cse361-2017-rsa.club/random_article3.html
Crawling 9 : http://cse361-2017-rsa.club/news.html
Crawling 10 : http://cse361-2017-rsa.club/login.html
Crawling 11 : http://cse361-2017-rsa.club/random_article2.html
Testing password on http://cse361-2017-rsa.club/login.html ...
1. Account: admin   Password: test1 --> False
2. Account: admin   Password: test2 --> False
3Account: admin   Password: Just1fy --> True
URL: http://cse361-2017-rsa.club/login.html
It works !   :^)
```

Figure 1.4 Result of Successful Logging in as A User.

b. Limitation

One limitation of our web crawler is that our way of determining login form and registration from. The current implement heavily depended on the number of input tag of the form. If the number of input tags are the same between these two forms, then we won't be able to tell which one is the login form. We are also making the assumption that the two input tags in the log in form will be username and password.

Another limitation of our web crawler is that it may not work on some other websites. The main reason is that the account with that username will most likely be banned after a few times of trying different passwords using login form bruteborcer.

5. Conclusion

The web crawler that we implemented will successful execute the following tasks:

- Parsing web content to various passwords.
- Remaining within the same target domain.
- Recovering from dead-end pages.
- Handling redirection correctly.
- Ignoring 4xx/5xx HTTP error codes to prevent crawler crashing.
- Identifying subdomains and hidden paths from robots.txt.

There are many improvements that we can add to the web crawler. The most important one is accurately identify the login form from other similar forms. One important aspect of making the web crawler will be carefully analyzing the target website. The main structure of the web crawler will be similar, but there are some variations that we can add to it to fulfill our needs. In conclusion, the process of making the web crawler enhances our understanding of how web sends and receives requests and how to parse data from the web.