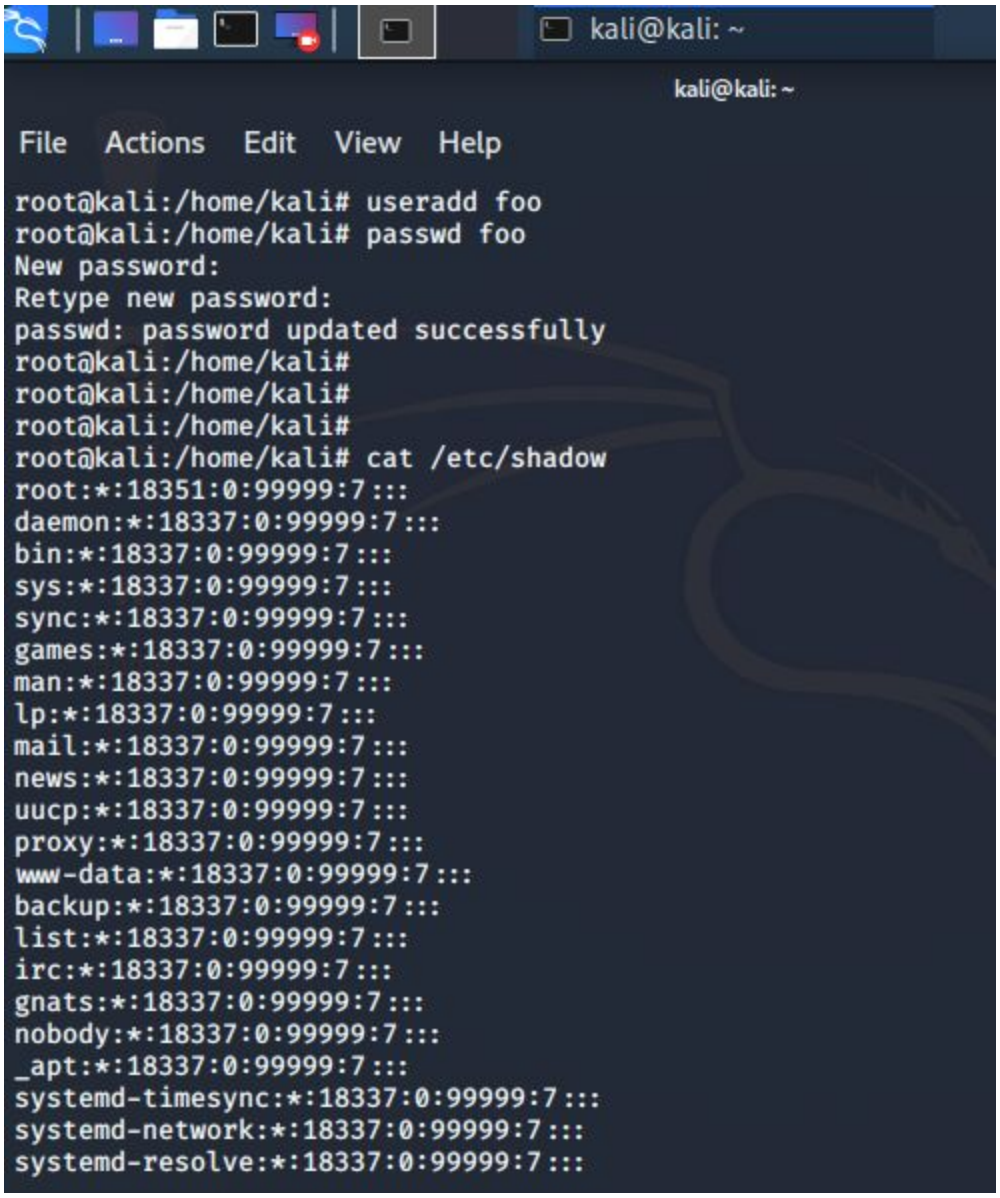
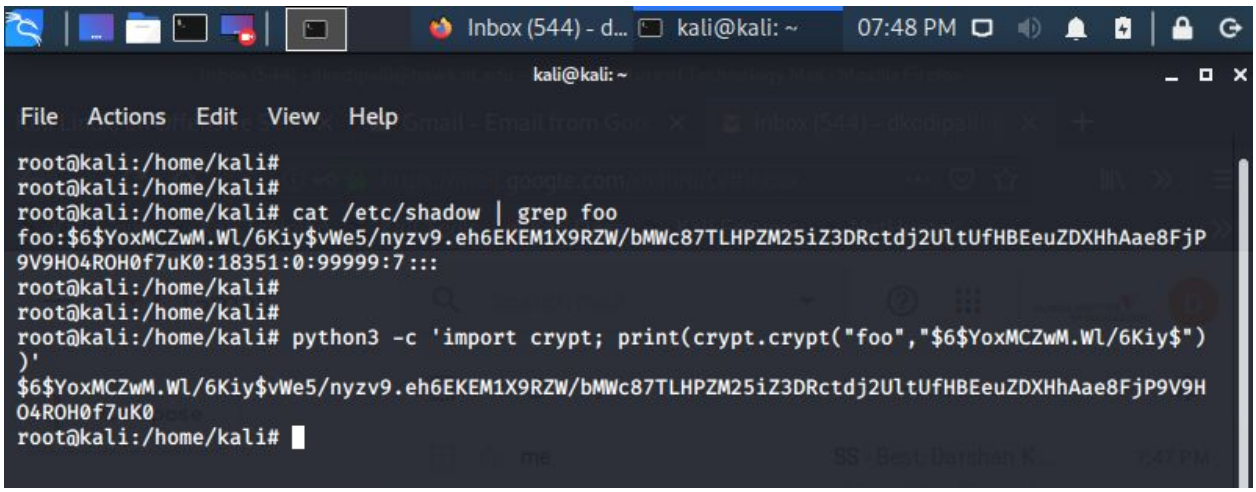
Task 1 : Playing the Fool

In this task, we login into the live kali linux vm and create a new user called foo and set his password to foo. Once that the user is added, we can verify its password by checking the contents of the file ***/etc/shadow.*** Below are the screenshots of how the user is added and how the details of the new user added is extracted from the cat command with the path to shadow file as its argument.



In the above screenshot, the first 2 commands are used to add a new user **foo** to the system. And the output of command cat /etc/shadow shows the details of all the users added in the system.

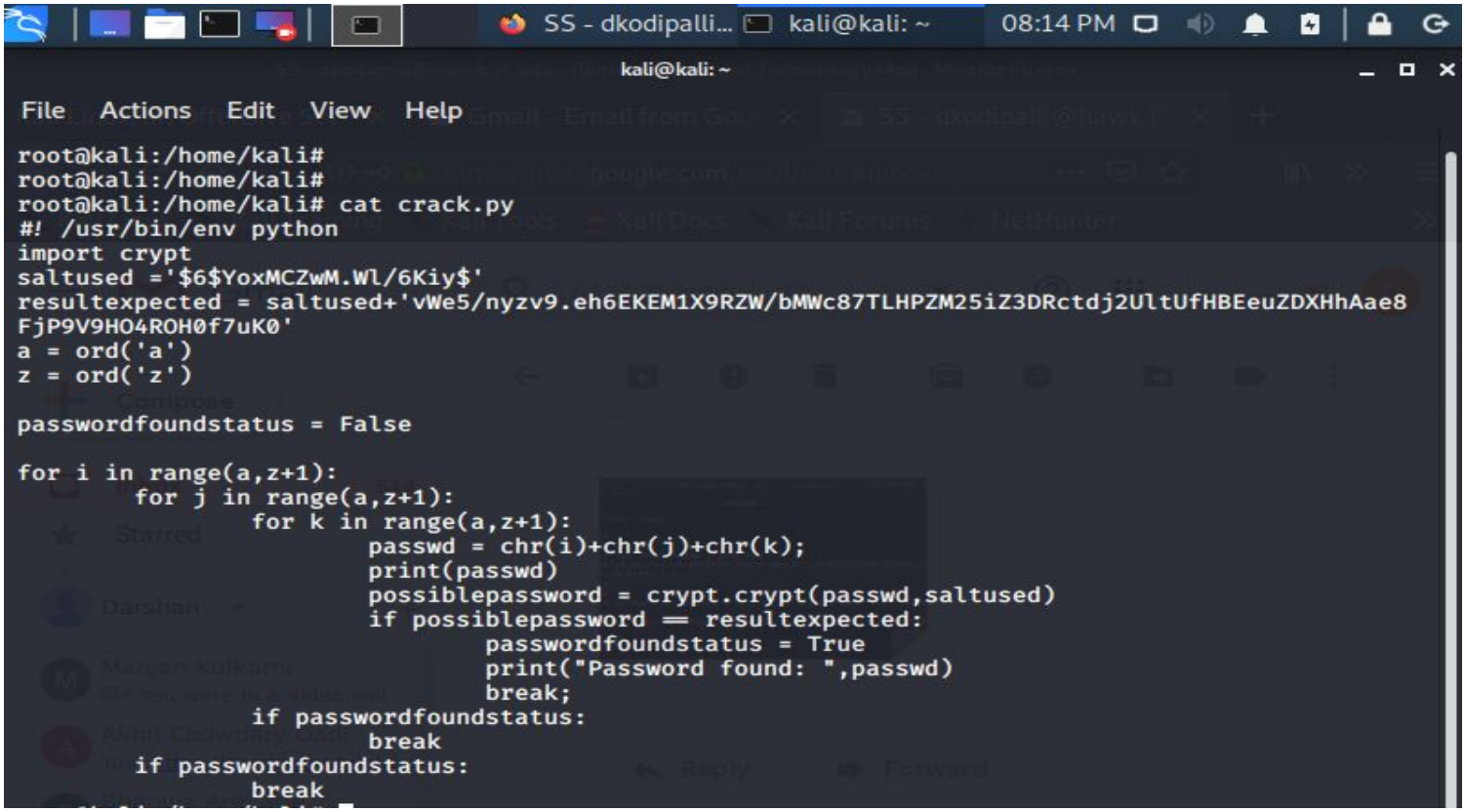


The screenshot above shows how the new user **foo's** details are extracted using both the grep command and the python command. In the output, the part of it that is delimited with : is the username ***foo*** and the one delimited with a $ is the hash of a password and the password is stored with their Salt too. The same output is displayed using the python program by passing the user foo as it's argument.

Task 2 : Cracking

This is a fun task, Here we write a Python program that tries all the possible 3 letter words available. Trying out all possible english words of the length we specify.

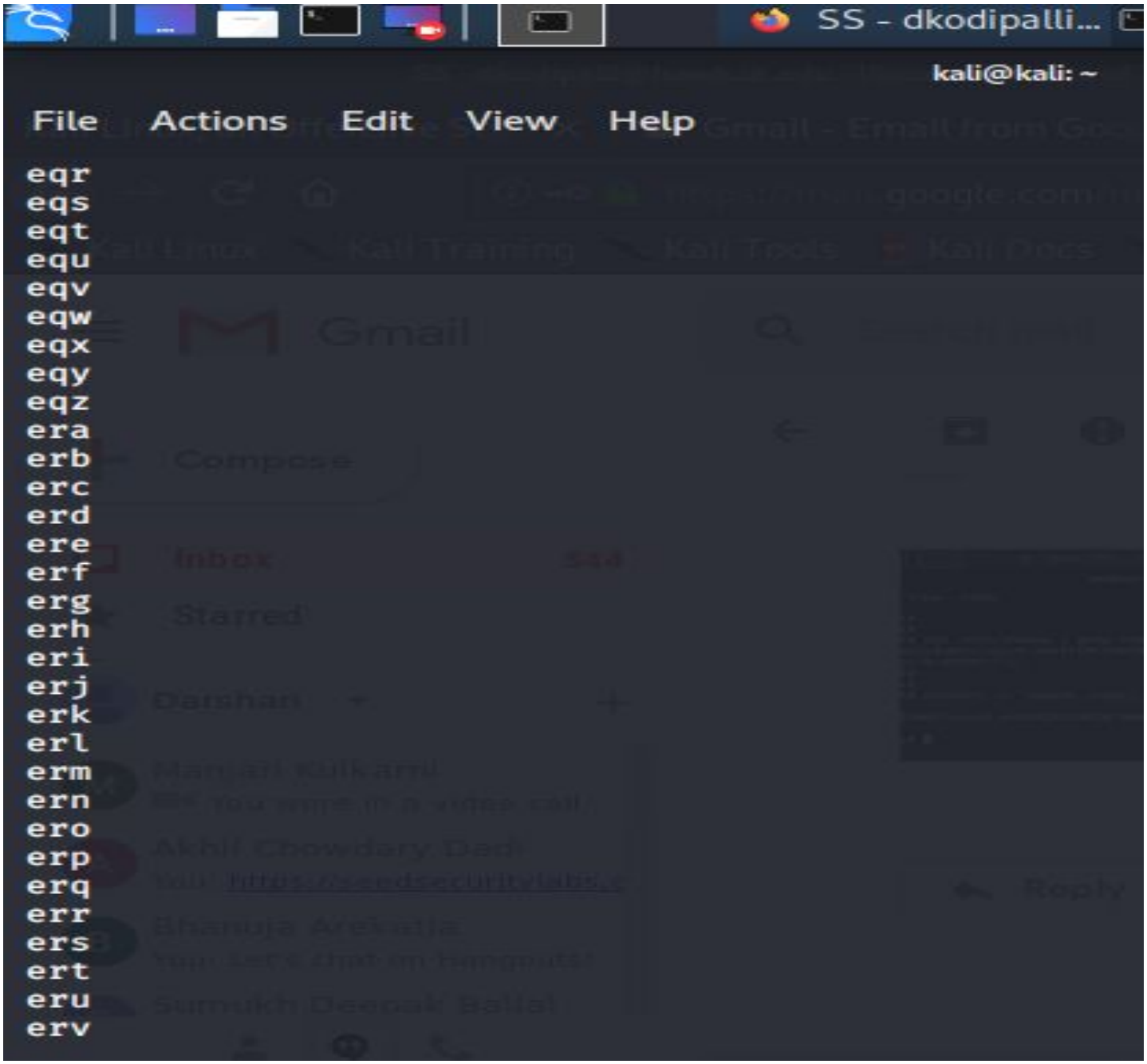
Since in this case we already know its length, we try to find all possible combinations of the english words of size 3. Since we know who the user is, we pass the name of the user ***foo*** as its argument, and also it opens the ***/etc/shadow*** file. Since that is where all the details of the user is stored. We call this program as **crack.py** and here is what it does. It has these 2 variables ***saltused*** and ***resultexpected*** that holds the salt and the final result that we're expecting the logic to return. And then we run the code in 3 loops (not efficient, but solves out purpose for now). These loops generate us the 3 letter words which are stored in the variable ***possiblepassword.*** This generated string is then sent as an argument along the salt to the crypt function and the result is compared with the ***resultexpected*** variable, we declared initially. Here is the code of the program..

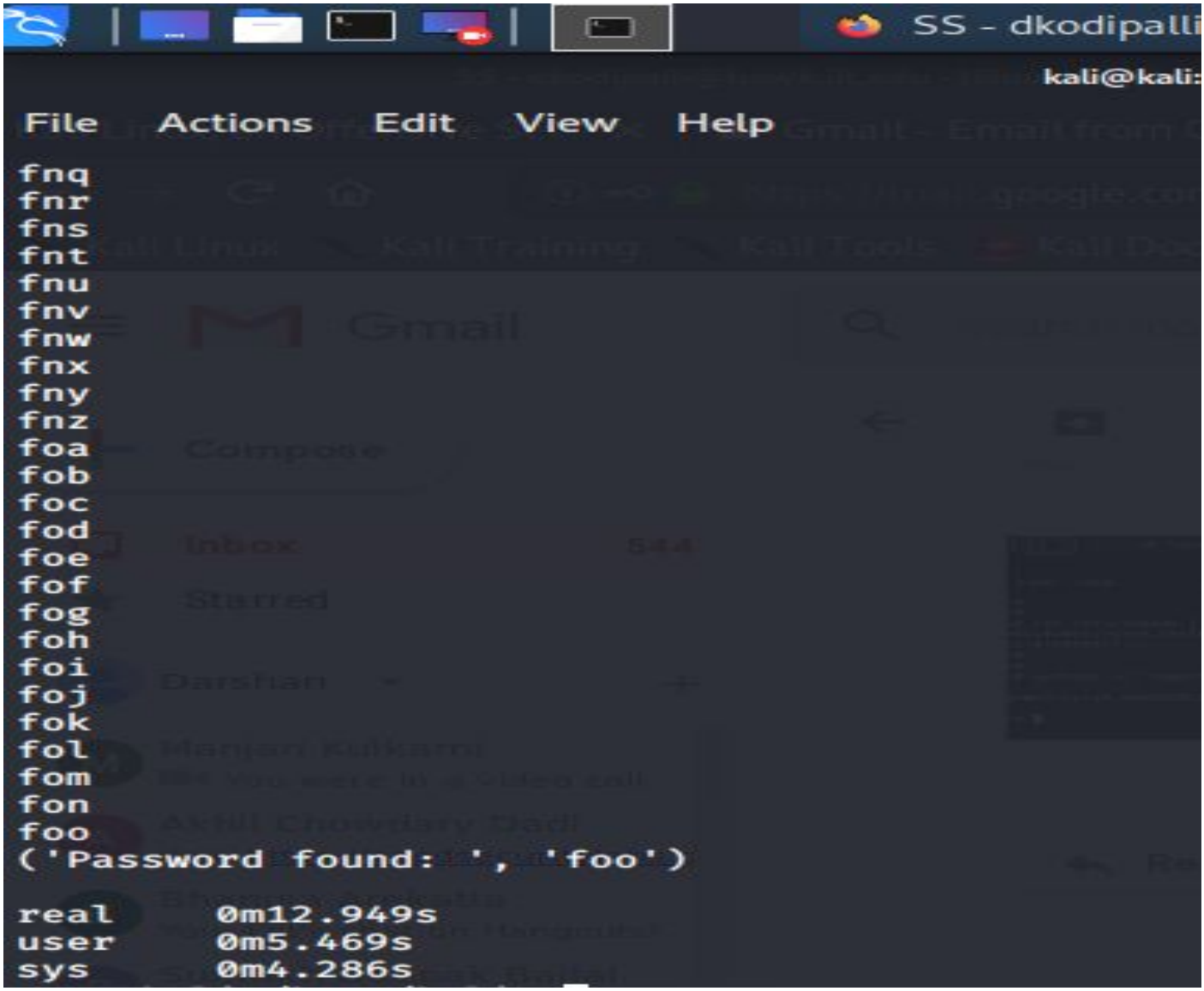


This program is then given the executable permissions and is run with the time tool command to display the time it took to run this code.

time ./crack.py foo

Is the command used to run this program. This takes the username foo as an argument. The from the output below, we can see how it generated every possible string of size 3 and then encrypts it with the salt and displays the final output by displaying the password and the time it took in executing this program.

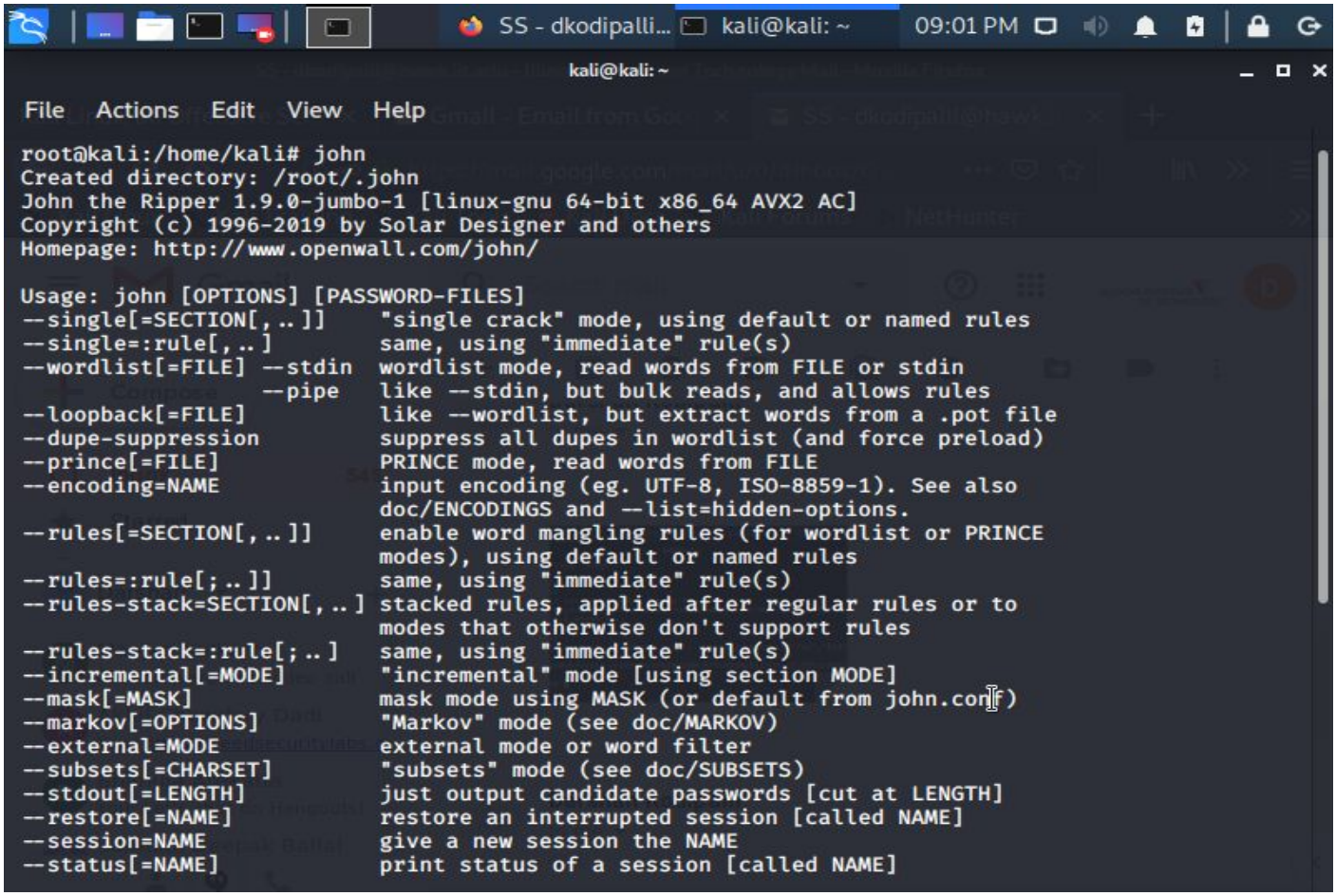

SS - dkodipalli...
kali@kali: ~
File Actions Edit View Help
eqr
eqs
eqt
equ
eqv
eqw
eqx
eqy
eqz
era
erb
erc
erd
ere
erf
erg
erh
eri
erj
erk
erl
erm
ern
ero
erp
erq
err
ers
ert
eru
erv



At the end you can see, the password found message is displayed along with the password ***foo*** and the total time of execution.

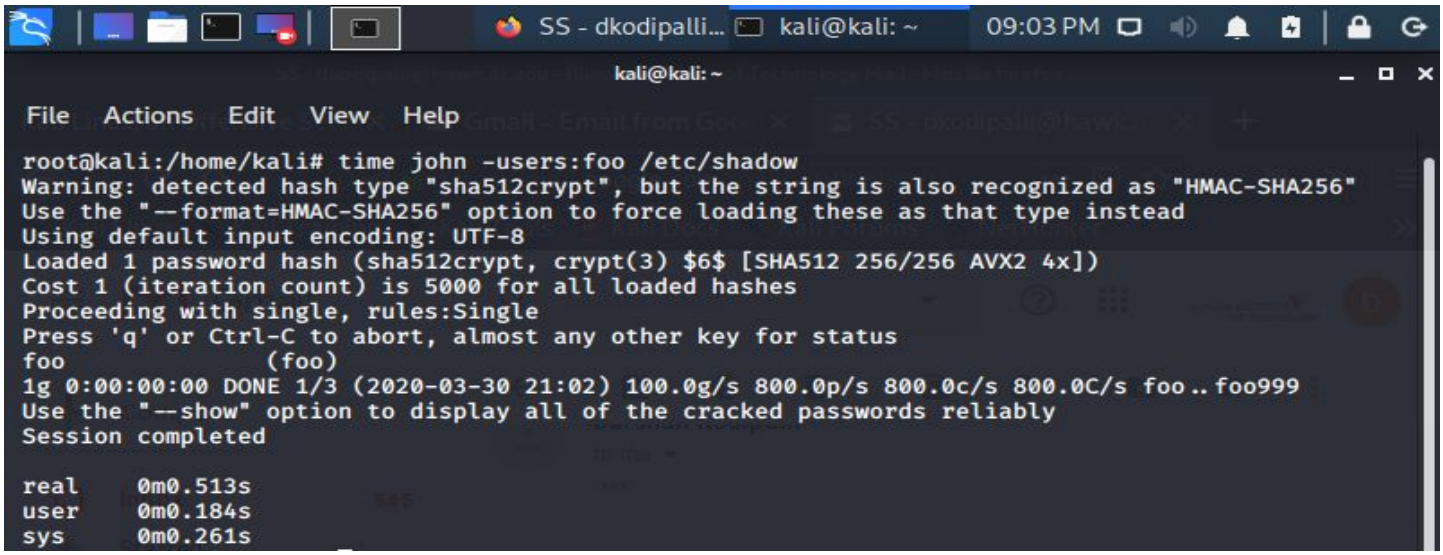
Task 3 : Cracking with John

In this task we make use of a pre-installed kali linux tool that does the same task as the task2 did. It makes use of an hierarchical approach. In the below screenshot we can see how to use the ***john*** tool. Just executing the ***john*** tool, gives the man page display of how the tool is to be used.



And, in the below screenshot we can see how the ***john*** tool is used to crack the password just by passing the username and the file that has all the passwords stored. In our case the passwords are stored in the ***/etc/shadow*** file. From the output we can see that the time taken to crack the password using this tool is .513 seconds.. Compared to 12.5 minutes taken by the custom code we've written in Task 2.

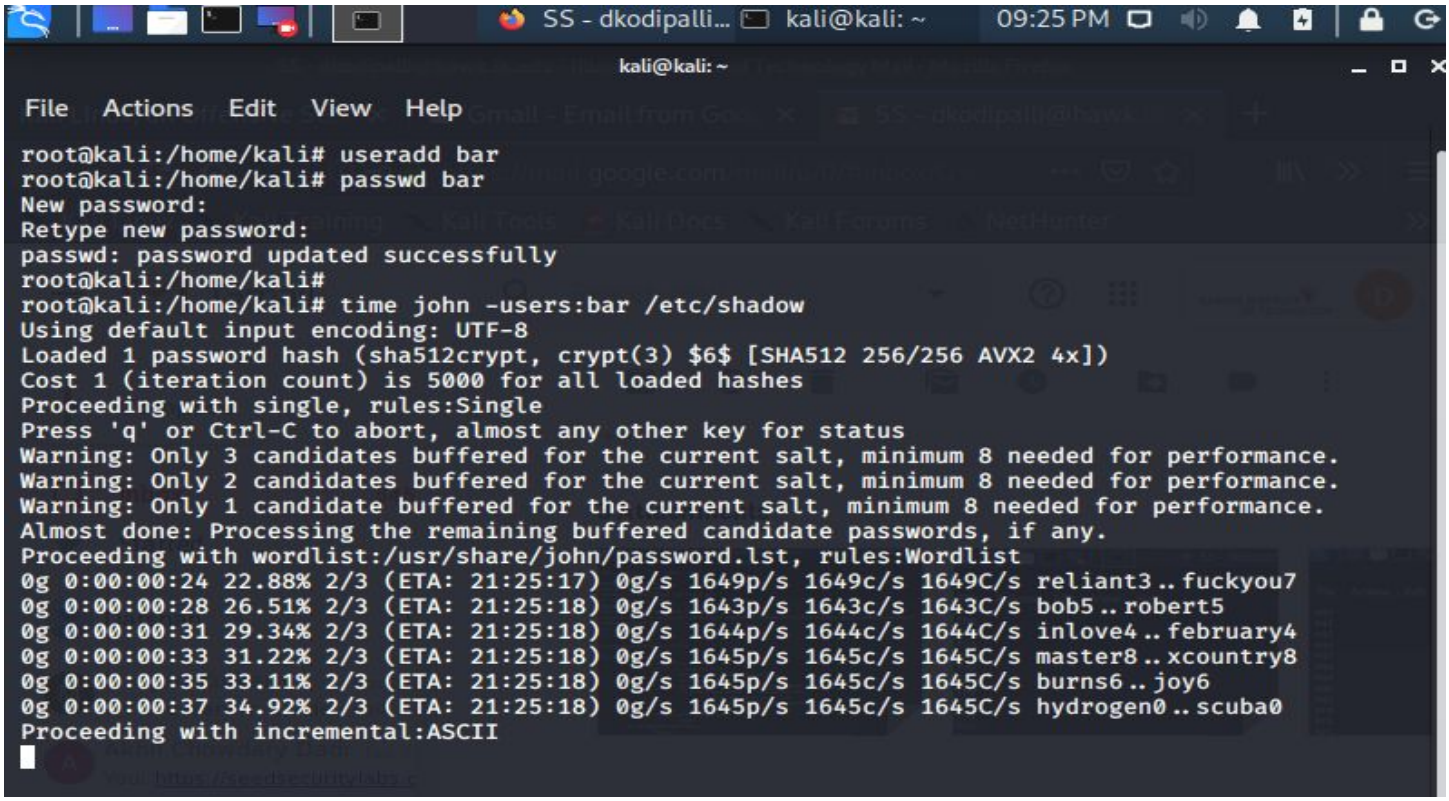
As the size of the password grows, the custom code we've written will have to be modified with **N** number of **for** loops, with N being the size of the password. Which proves to be very inefficient.
Therefore in comparing the execution times, the time taken by the john tool in cracking the password is much much lesser when compared to the time taken by the custom code.



From the output above, we can see this tool also recognises the string type and also displays various steps it went through in finding the cracked password.. It includes the iteration count, password hash. It also gives us a suggestion to use **--show** option to display all the intermediate passwords it verified in the process.

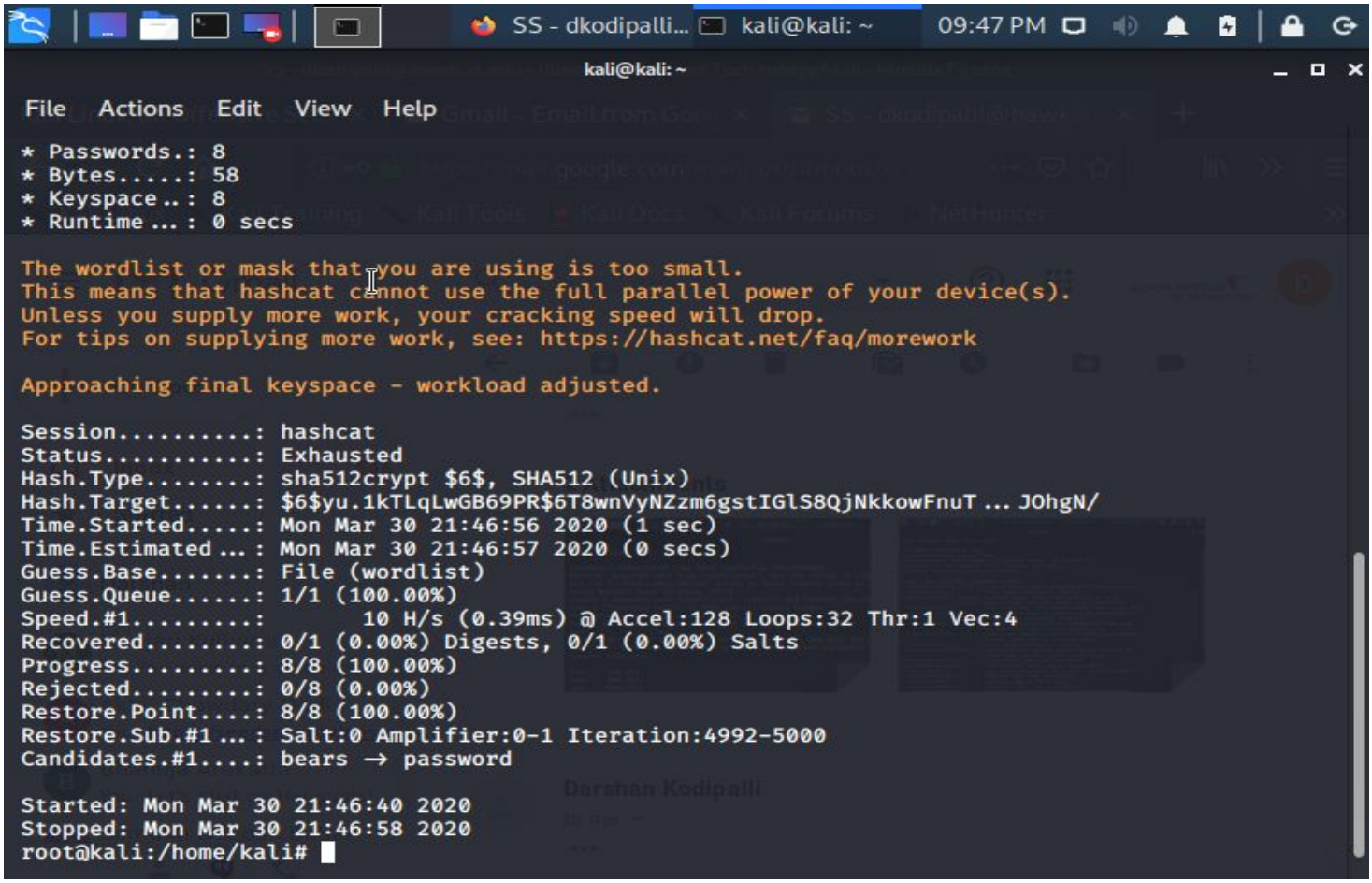
Task 4 : Dictionary Attacks with Transformation Rules

In this task, we make use of the ***Hashcat*** tool which is an inbuilt kali linux tool. This tool helps in performing dictionary attacks which has the necessity language for generating mutation rules that can be applied to transform the words into the dictionary. In this task, we create another user ***bar*** with a password ***whit3s0cks1996.***
Here we use the same approach initially we used in the previous task. We pass on the username **bar** and the file **/etc/shadow** to the john tool.

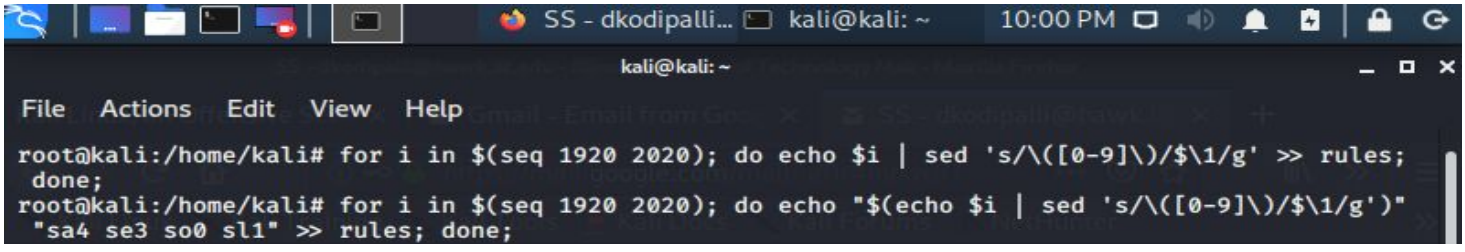


Now, since the password we used is a combination of letters and numbers, it is taking more time than usual to crack the password. Using this password combination it'd be very difficult to crack using our custom code.
Here, we make a small yet effective assumption. Since the password's scenario is somewhat related to a sports club.. We assume the organization has some serious sports fan and their password might have a hint of some sports person. Therefore, we store some sports related names in a test file called ***wordlist*** and also the contents of the shadow file for a user ***bar*** in another temp file called ***crackme.***

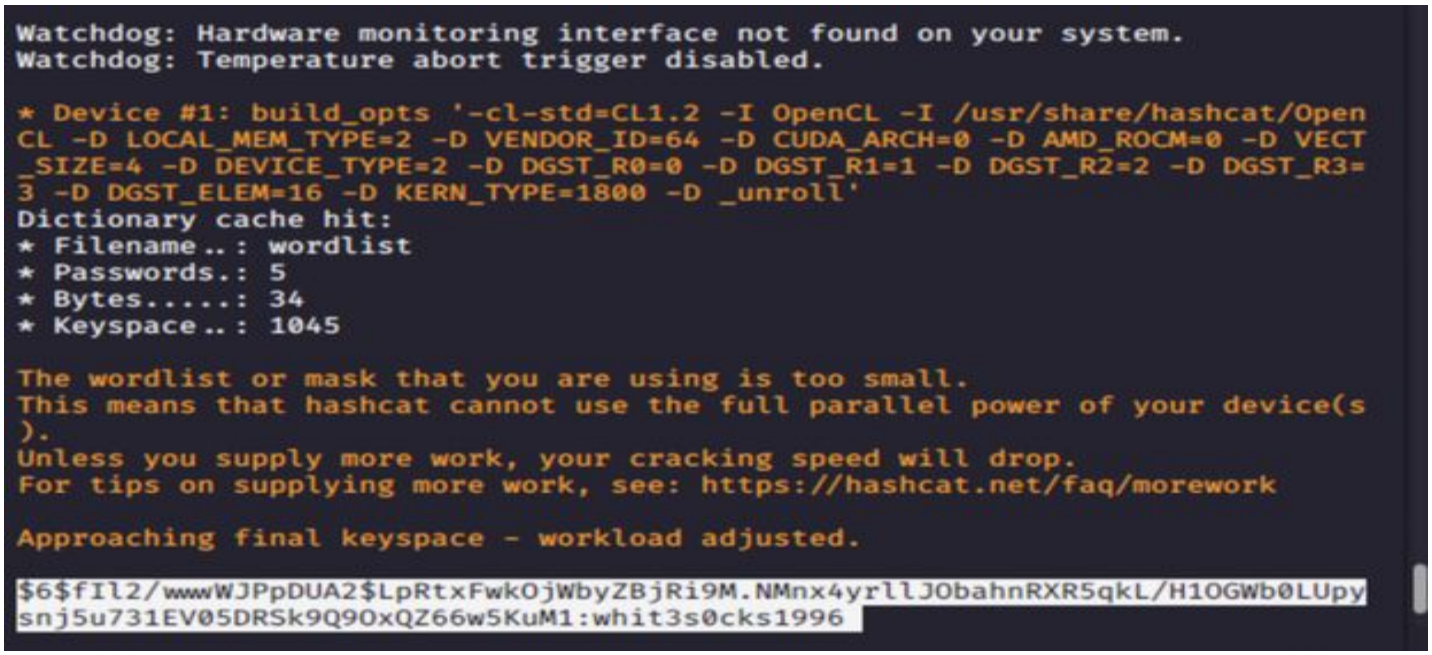
And now we pass these files to the tool ***hashcat.*** From the output we can see it finished quickly with status as ***Executed*** but didn't return the right password as the password has the **birth year** at the end and also **3** in place of **E** and **0** in place of **o.**



To handle these numbers, we make use of the hashcat's rules. In these rules we include various scenarios and also possible substitutions which includes replacing **s** with **5**, **a** with **4**, **e** with **3**, **l/i** with **1.** And for birthdays, we auto generate all the numbers from the year **1920** to **2020** and add them to the rules set.



Once we have our rules set ready and also the wordlist, we now pass these files as an input to the hashcat tool and below is the screenshot of the output.

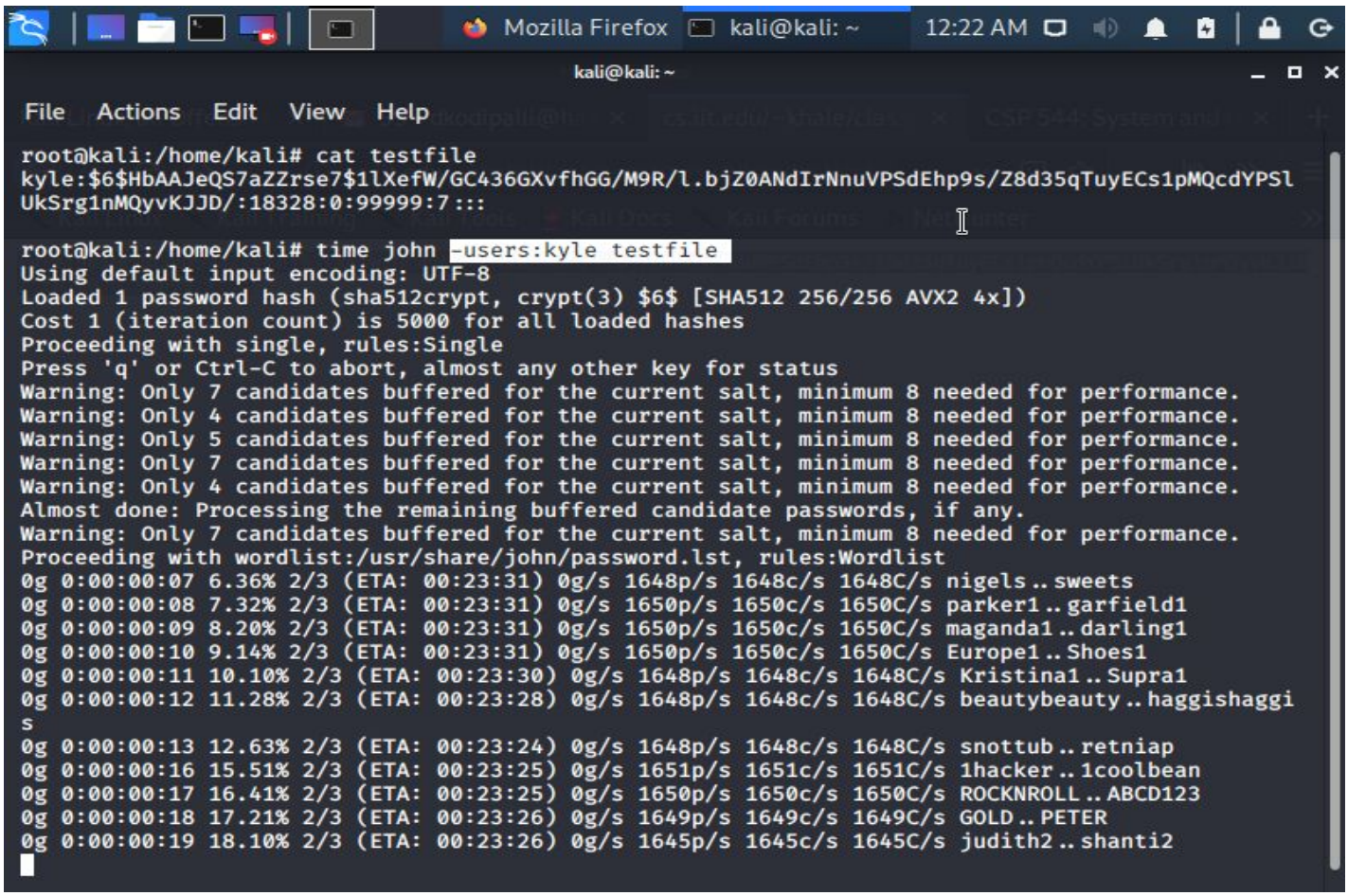


Task 5 : Putting it Together

In this task, we have to crack the password from the given salted unix password. Given the salted password as

kyle:6HbAAJeQS7aZZrse7$1lXefW/GC436GXvfhGG/M9R/l.bjZ0ANdlrNnuVPSdEhp9s/Z8d35qTuyECs1pMQcdYPSlUkSrg1nMQyvKJJD/:18328:0:99999:7:::

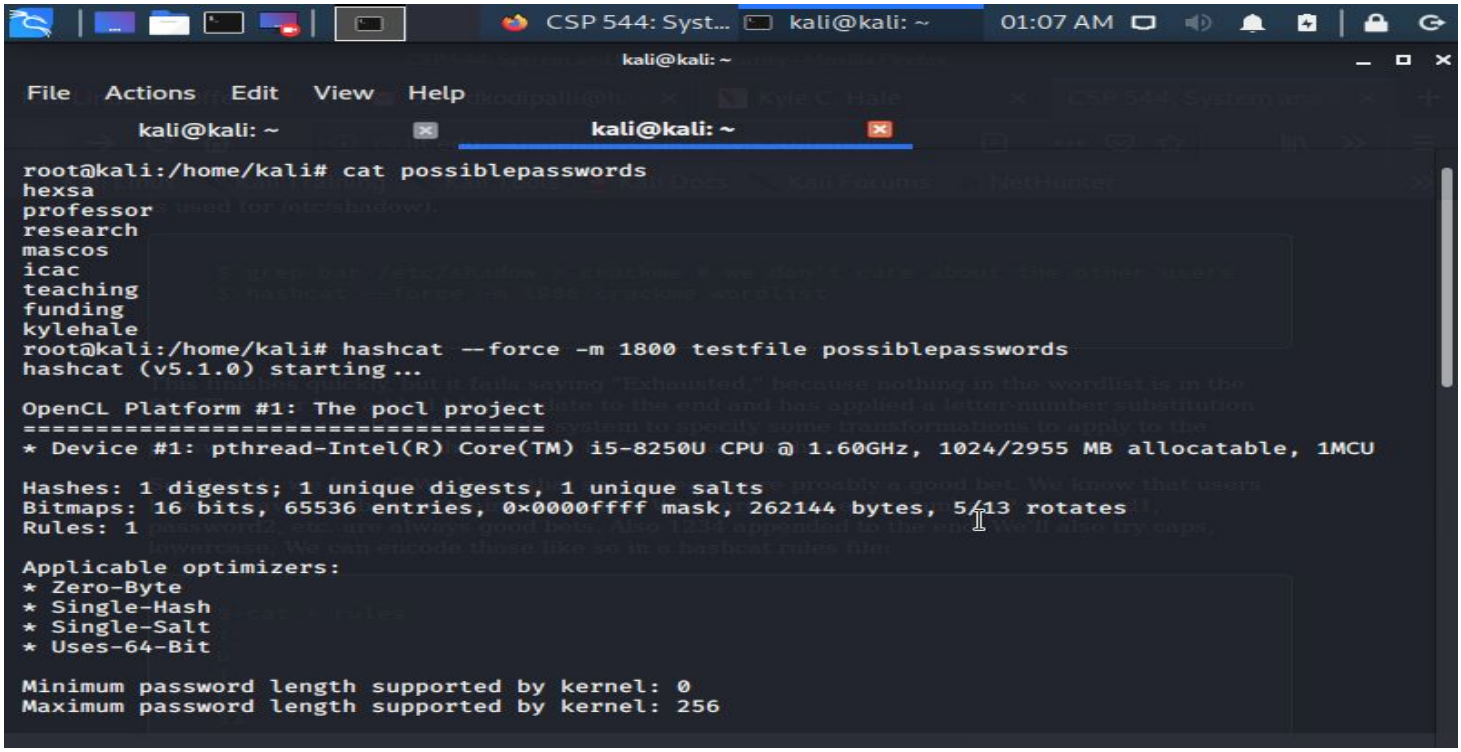
We can use the **john** tool by storing this in a temporary file and passing **kyle** as a parameter to the **-users** tag and the path to the temporary file ~ **testfile**. In the screenshot below we can see that the file **testfile** is created and the salted unix password is stored in it. We now pass this file as an argument to the **john** tool and we can see this in the screenshot below:

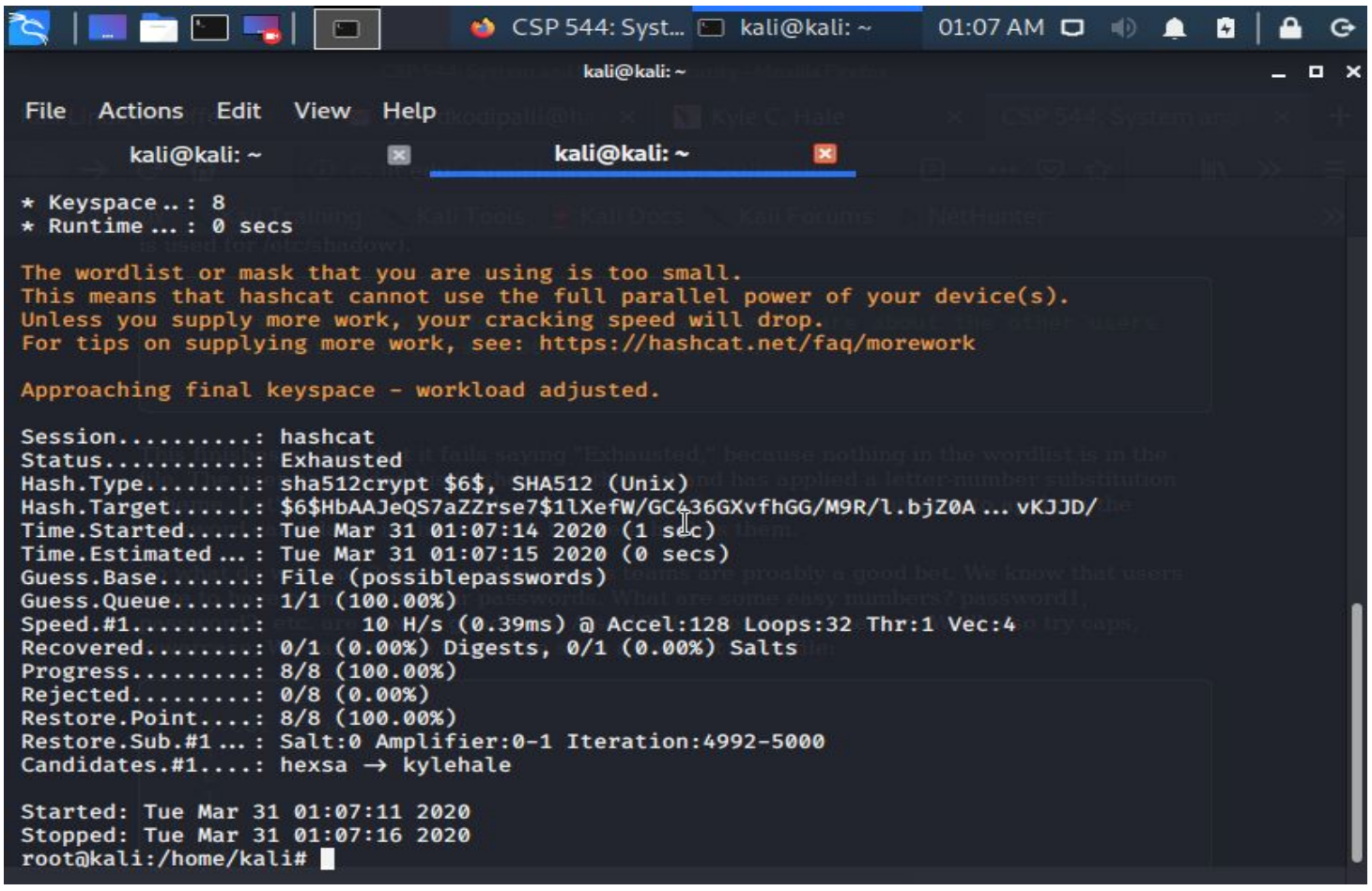


Since the **john** tool is taking more time in cracking the password, it is clear that the password is a good combination of number and characters.

The hint to the password is obtained by glancing the official site of Prof. Kyle. Going through his site and noticing all his accomplishments.. I created a **possiblepasswords** file with its contents shown in the screenshot below. And then the salted unix password is stored in the ***testfile,*** we pass these 2 files as an argument to the hashcat tool. Since the password is not one among the contents of the ***possiblepassword*** file, we see the status of the hashcat tool as Exhausted.

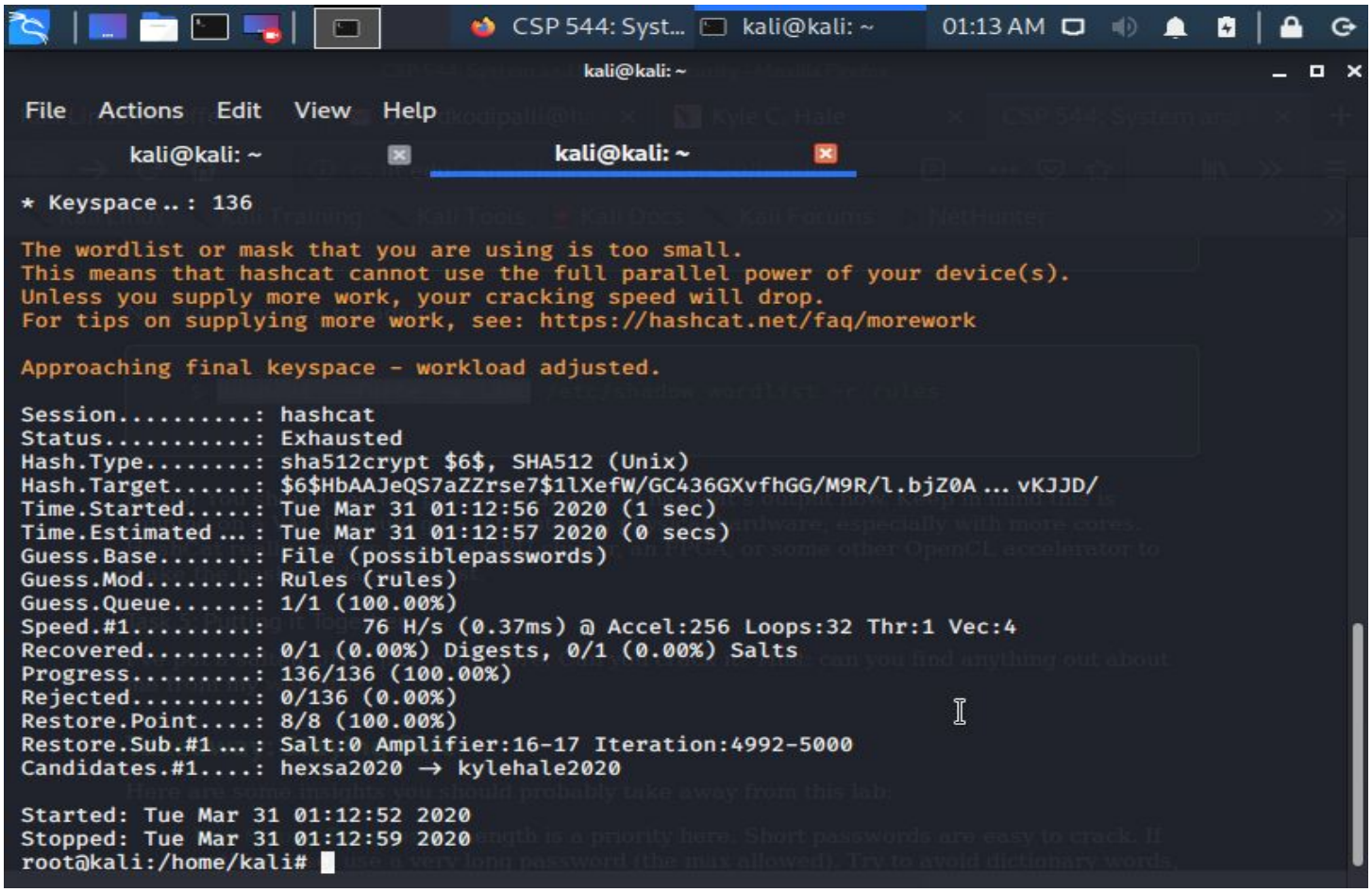
System And Network Security : Password Cracking **Darshan K (A20446137)**





Now that the password is not one of the listed above, we try with various numbers. SI started from the year ***2013*** as it is the least number found in the site. The year where Prof. completed his Masters. Therefore added the substitution parameters and the year from 2013 to 2020 into the rule set and passed this rules file along with the 2 other files as an input to the hashcat tool. Below is the screenshot of how the ***rules*** file is framed and the output when these are passed an argument.

```
root@kali:/home/kali# cat > rules
$2 $0 $1 $3
root@kali:/home/kali# cat rules
$2 $0 $1 $3
root@kali:/home/kali# for i in $(seq 2013 2020); do echo $i | sed 's/\([0-9]\)/$\1/g' >> rules;
 done;
root@kali:/home/kali# for i in $(seq 2013 2020); do echo $i | sed 's/\([0-9]\)/$\1/g' >> rules;
 done;
root@kali:/home/kali# hashcat --force -m 1800 testfile possiblepasswords -r rules
```



After trying all the possible passwords to the best of my knowledge, we can see the hashcat still fails to find the actual password as the files are not configured correctly.