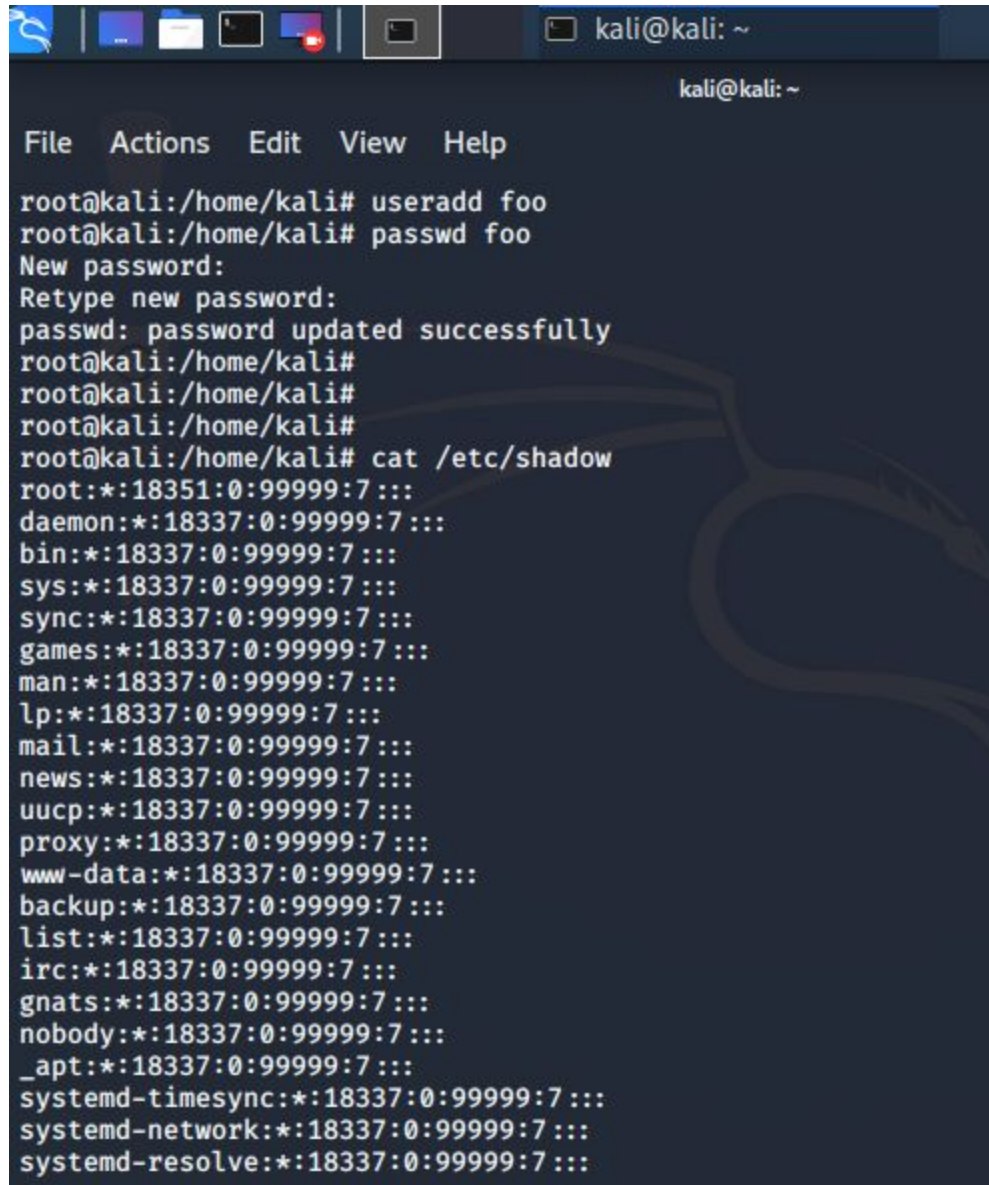


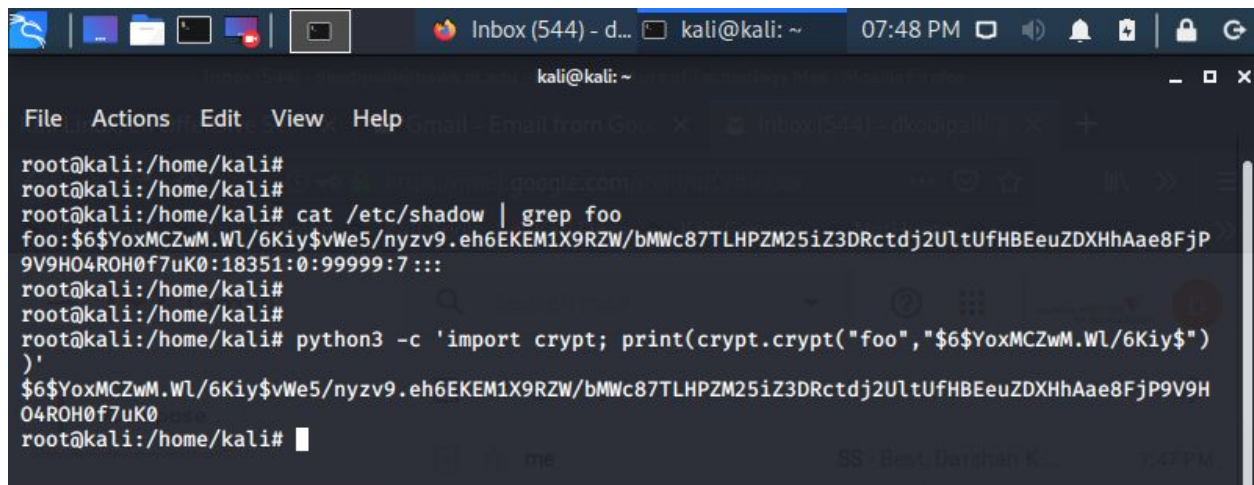
## **Task 1 : Playing the Fool**

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.



```
kali@kali: ~  
File Actions Edit View Help  
root@kali:/home/kali# useradd foo  
root@kali:/home/kali# passwd foo  
New password:  
Retype new password:  
passwd: password updated successfully  
root@kali:/home/kali#  
root@kali:/home/kali#  
root@kali:/home/kali#  
root@kali:/home/kali# cat /etc/shadow  
root:*:18351:0:99999:7:::  
daemon:*:18337:0:99999:7:::  
bin:*:18337:0:99999:7:::  
sys:*:18337:0:99999:7:::  
sync:*:18337:0:99999:7:::  
games:*:18337:0:99999:7:::  
man:*:18337:0:99999:7:::  
lp:*:18337:0:99999:7:::  
mail:*:18337:0:99999:7:::  
news:*:18337:0:99999:7:::  
uucp:*:18337:0:99999:7:::  
proxy:*:18337:0:99999:7:::  
www-data:*:18337:0:99999:7:::  
backup:*:18337:0:99999:7:::  
list:*:18337:0:99999:7:::  
irc:*:18337:0:99999:7:::  
gnats:*:18337:0:99999:7:::  
nobody:*:18337:0:99999:7:::  
_apt:*:18337:0:99999:7:::  
systemd-timesync:*:18337:0:99999:7:::  
systemd-network:*:18337:0:99999:7:::  
systemd-resolve:*:18337:0:99999:7:::
```

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.

A screenshot of a Kali Linux terminal window. The window title is 'kali@kali: ~'. The terminal shows a series of commands and their outputs. The user is root at kali. The first command is 'cat /etc/shadow | grep foo', which outputs the entry for user 'foo': 'foo:\$6\$YoxMCZwM.WL/6Kiy\$vWe5/nyzv9.eh6EKEM1X9RZW/bMwc87LHPZM25iZ3DRctdj2UltUfHBEeuZDXHhAae8FjP9V9H04ROH0f7uK0:18351:0:99999:7:::'. The second command is 'python3 -c \'import crypt; print(crypt.crypt("foo","\$6\$YoxMCZwM.WL/6Kiy\$"))\'', which outputs the same string: '\$6\$YoxMCZwM.WL/6Kiy\$vWe5/nyzv9.eh6EKEM1X9RZW/bMwc87LHPZM25iZ3DRctdj2UltUfHBEeuZDXHhAae8FjP9V9H04ROH0f7uK0'.

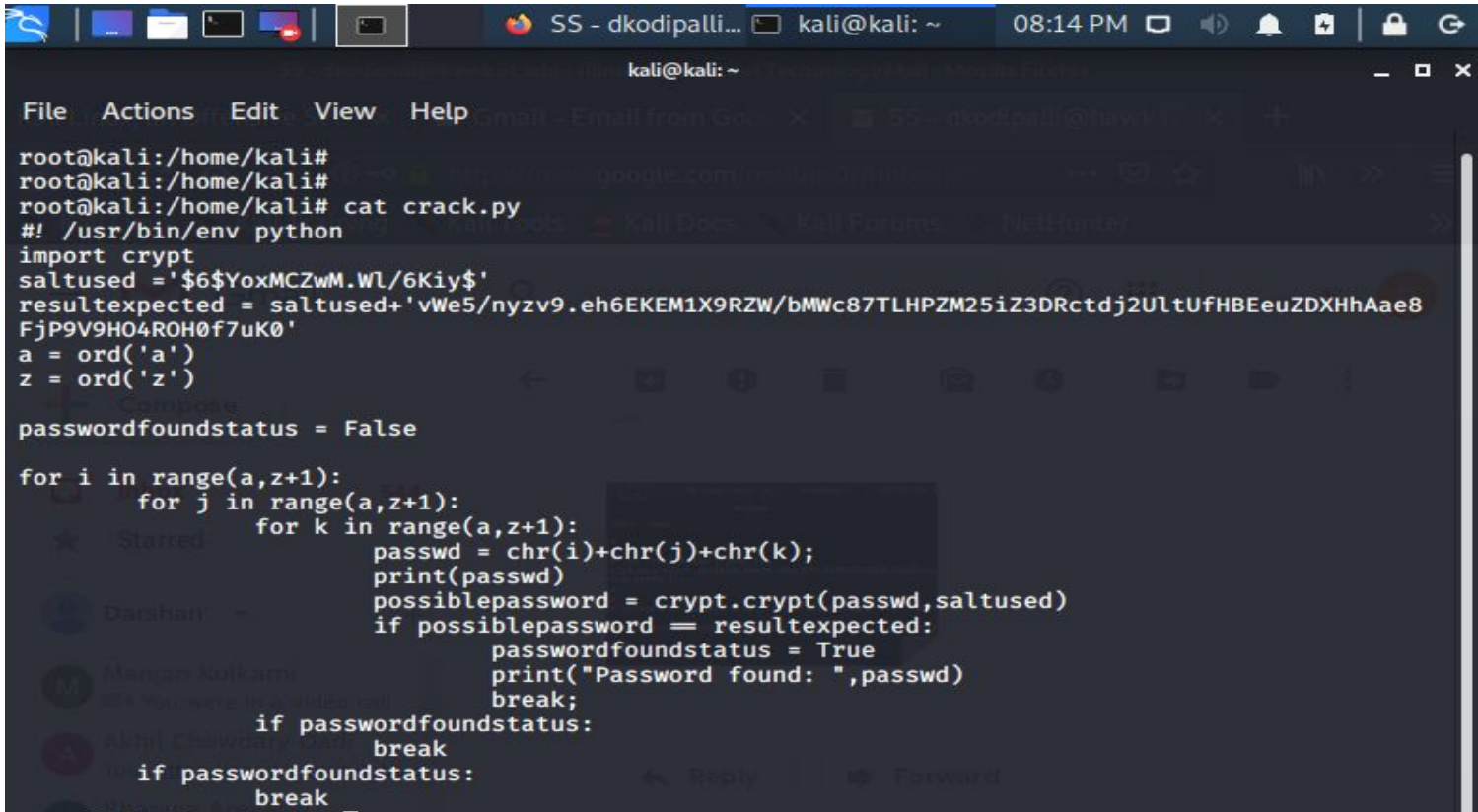
```
kali@kali: ~
File Actions Edit View Help
root@kali:/home/kali#
root@kali:/home/kali#
root@kali:/home/kali# cat /etc/shadow | grep foo
foo:$6$YoxMCZwM.WL/6Kiy$vWe5/nyzv9.eh6EKEM1X9RZW/bMwc87LHPZM25iZ3DRctdj2UltUfHBEeuZDXHhAae8FjP9V9H04ROH0f7uK0:18351:0:99999:7:::
root@kali:/home/kali#
root@kali:/home/kali#
root@kali:/home/kali# python3 -c 'import crypt; print(crypt.crypt("foo","$6$YoxMCZwM.WL/6Kiy$"))'
$6$YoxMCZwM.WL/6Kiy$vWe5/nyzv9.eh6EKEM1X9RZW/bMwc87LHPZM25iZ3DRctdj2UltUfHBEeuZDXHhAae8FjP9V9H04ROH0f7uK0
root@kali:/home/kali#
```

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..

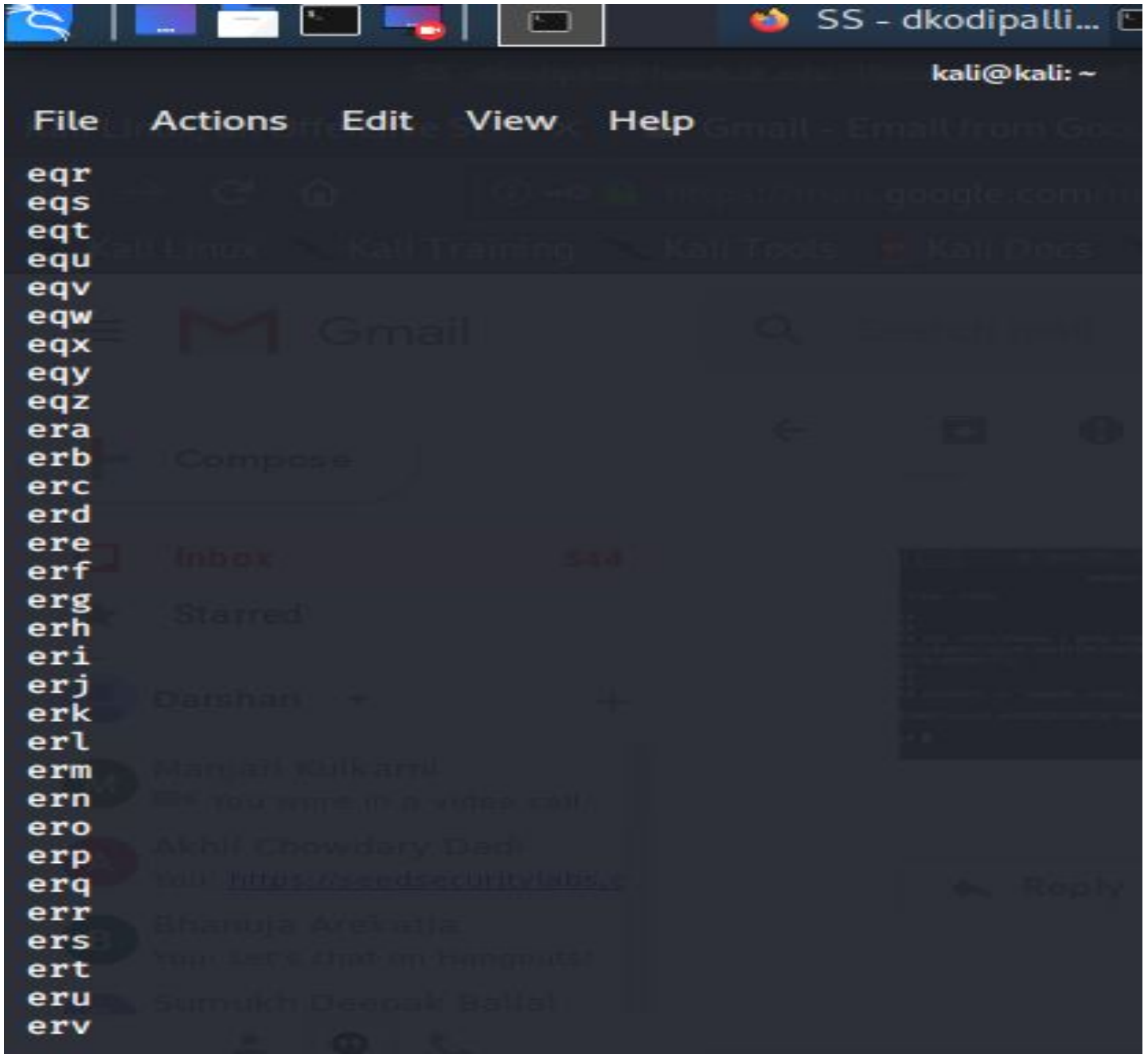
A screenshot of a Kali Linux terminal window. The terminal shows a root user at the kali machine in the /home/kali directory. The user runs 'cat crack.py' to display the contents of a Python script. The script imports the 'crypt' module and defines a salt 'saltused' and a result 'resultexpected'. It then iterates through all possible 3-character strings using characters 'a' through 'z'. For each string, it prints the string, encrypts it with the salt, and compares the result to 'resultexpected'. If a match is found, it prints the password and sets 'passwordfoundstatus' to True. The script ends with a loop that breaks once a password is found.

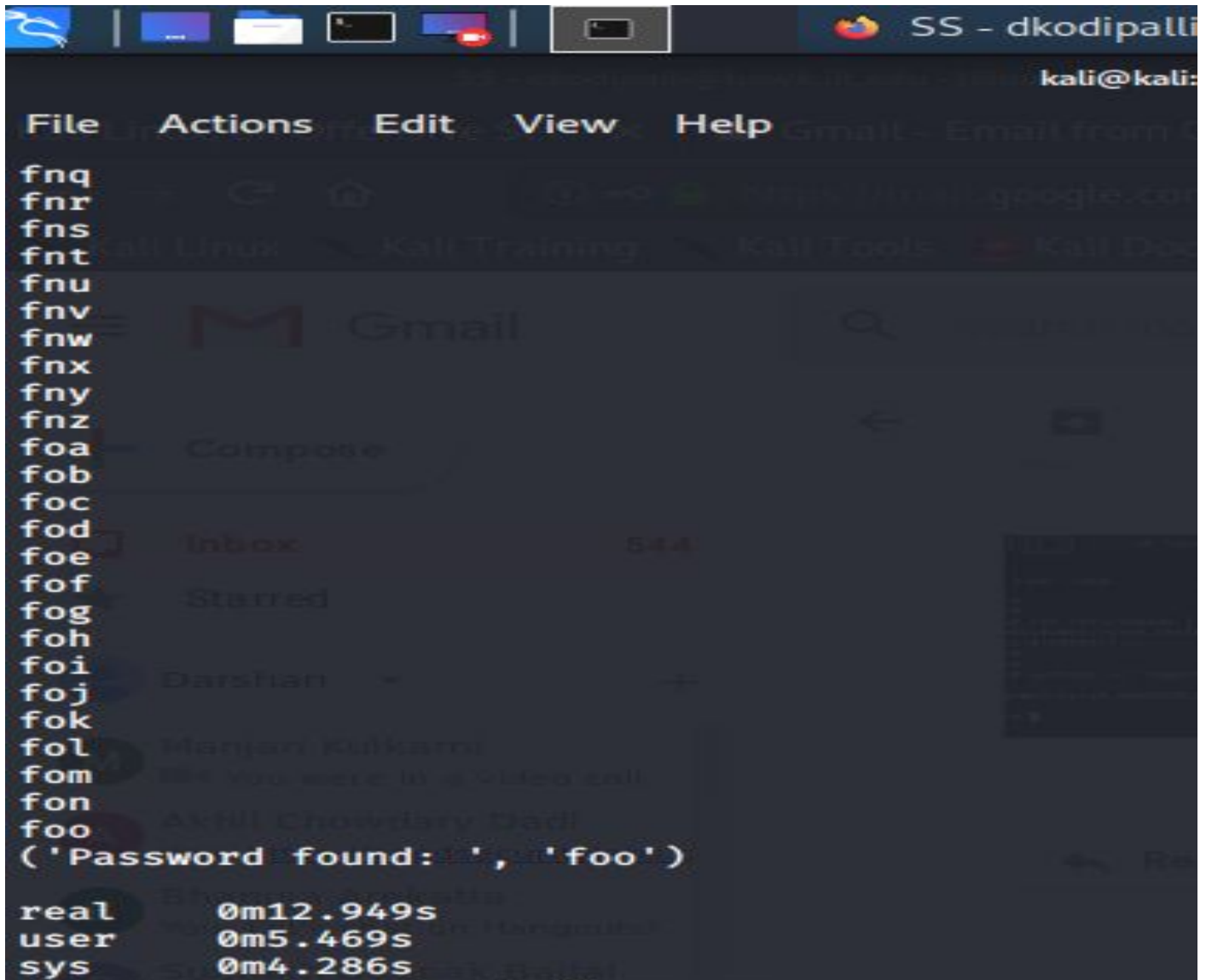
```
root@kali:/home/kali#  
root@kali:/home/kali#  
root@kali:/home/kali# cat crack.py  
#!/usr/bin/env python  
import crypt  
saltused = '$6$YoxMCZwM.WL/6Kiy$'  
resultexpected = saltused+'vWe5/nyzv9.eh6EKEM1X9RZW/bMwc87TLHPZM25iZ3DRctdj2UltUfHBEeuZDXHhAae8  
FjP9V9H04ROH0f7uK0'  
a = ord('a')  
z = ord('z')  
  
passwordfoundstatus = False  
  
for i in range(a,z+1):  
    for j in range(a,z+1):  
        for k in range(a,z+1):  
            passwd = chr(i)+chr(j)+chr(k);  
            print(passwd)  
            possiblepassword = crypt.crypt(passwd,saltused)  
            if possiblepassword == resultexpected:  
                passwordfoundstatus = True  
                print("Password found: ",passwd)  
                break;  
        if passwordfoundstatus:  
            break  
    if passwordfoundstatus:  
        break
```

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.





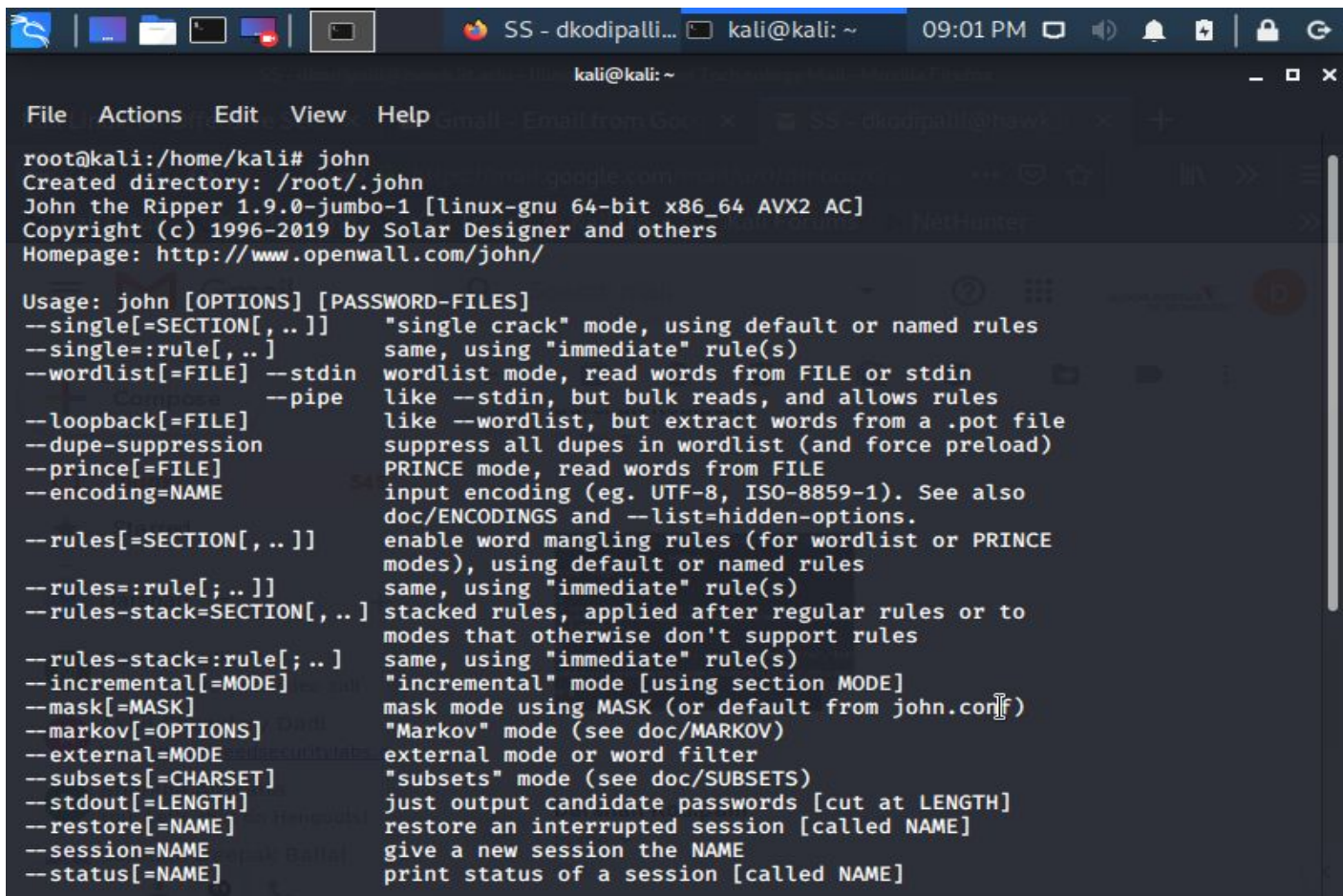
```
File Actions Edit View Help
fnq
fnr
fns
fnt
fnu
fnv
fnw
fnx
fny
fnz
foa
fob
foc
fod
foe
fof
fog
foh
foi
foj
fok
fol
fom
fon
foo
('Password found: 'foo')
real    0m12.949s
user    0m5.469s
sys     0m4.286s
```

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.

A screenshot of a Kali Linux terminal window. The window title is 'kali@kali: ~'. The terminal shows the command 'john' being executed, which displays the man page for John the Ripper. The output includes the version 'John the Ripper 1.9.0-jumbo-1 [linux-gnu 64-bit x86\_64 AVX2 AC]', copyright information, and a detailed list of usage options and their descriptions. The options listed include --single, --wordlist, --loopback, --dupe-suppression, --prince, --encoding, --rules, --rules-stack, --incremental, --mask, --markov, --external, --subsets, --stdout, --restore, --session, and --status.

```
root@kali:/home/kali# john
Created directory: /root/.john
John the Ripper 1.9.0-jumbo-1 [linux-gnu 64-bit x86_64 AVX2 AC]
Copyright (c) 1996-2019 by Solar Designer and others
Homepage: http://www.openwall.com/john/

Usage: john [OPTIONS] [PASSWORD-FILES]
--single[=SECTION[, ..]]  "single crack" mode, using default or named rules
--single=:rule[, ..]      same, using "immediate" rule(s)
--wordlist[=FILE] --stdin wordlist mode, read words from FILE or stdin
                        --pipe  like --stdin, but bulk reads, and allows rules
--loopback[=FILE]         like --wordlist, but extract words from a .pot file
--dupe-suppression        suppress all dupes in wordlist (and force preload)
--prince[=FILE]           PRINCE mode, read words from FILE
--encoding=NAME           input encoding (eg. UTF-8, ISO-8859-1). See also
                        doc/ENCODINGS and --list=hidden-options.
--rules[=SECTION[, ..]]  enable word mangling rules (for wordlist or PRINCE
                        modes), using default or named rules
--rules=:rule[; ..]      same, using "immediate" rule(s)
--rules-stack=SECTION[, ..] stacked rules, applied after regular rules or to
                        modes that otherwise don't support rules
--rules-stack=:rule[; ..] same, using "immediate" rule(s)
--incremental[=MODE]     "incremental" mode [using section MODE]
--mask[=MASK]            mask mode using MASK (or default from john.conf)
--markov[=OPTIONS]       "Markov" mode (see doc/MARKOV)
--external=MODE          external mode or word filter
--subsets[=CHARSET]      "subsets" mode (see doc/SUBSETS)
--stdout[=LENGTH]        just output candidate passwords [cut at LENGTH]
--restore[=NAME]         restore an interrupted session [called NAME]
--session=NAME           give a new session the NAME
--status[=NAME]          print status of a session [called NAME]
```

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.

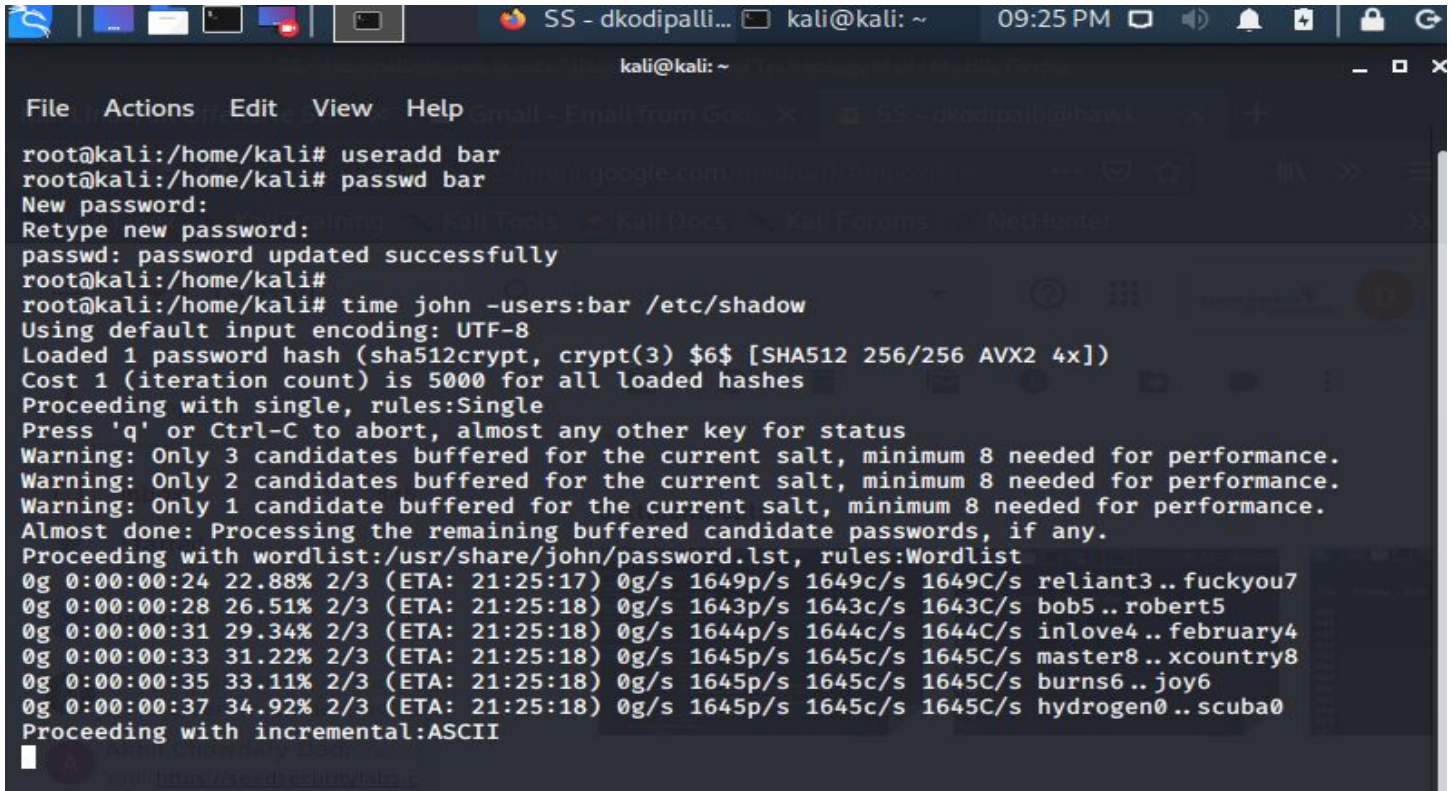


```
kali@kali: ~  
File Actions Edit View Help  
root@kali:/home/kali# time john -users:foo /etc/shadow  
Warning: detected hash type "sha512crypt", but the string is also recognized as "HMAC-SHA256"  
Use the "--format=HMAC-SHA256" option to force loading these as that type instead  
Using default input encoding: UTF-8  
Loaded 1 password hash (sha512crypt, crypt(3) $6$ [SHA512 256/256 AVX2 4x])  
Cost 1 (iteration count) is 5000 for all loaded hashes  
Proceeding with single, rules:Single  
Press 'q' or Ctrl-C to abort, almost any other key for status  
foo (foo)  
1g 0:00:00:00 DONE 1/3 (2020-03-30 21:02) 100.0g/s 800.0p/s 800.0c/s 800.0C/s foo..foo999  
Use the "--show" option to display all of the cracked passwords reliably  
Session completed  
  
real    0m0.513s  
user    0m0.184s  
sys     0m0.261s
```

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.



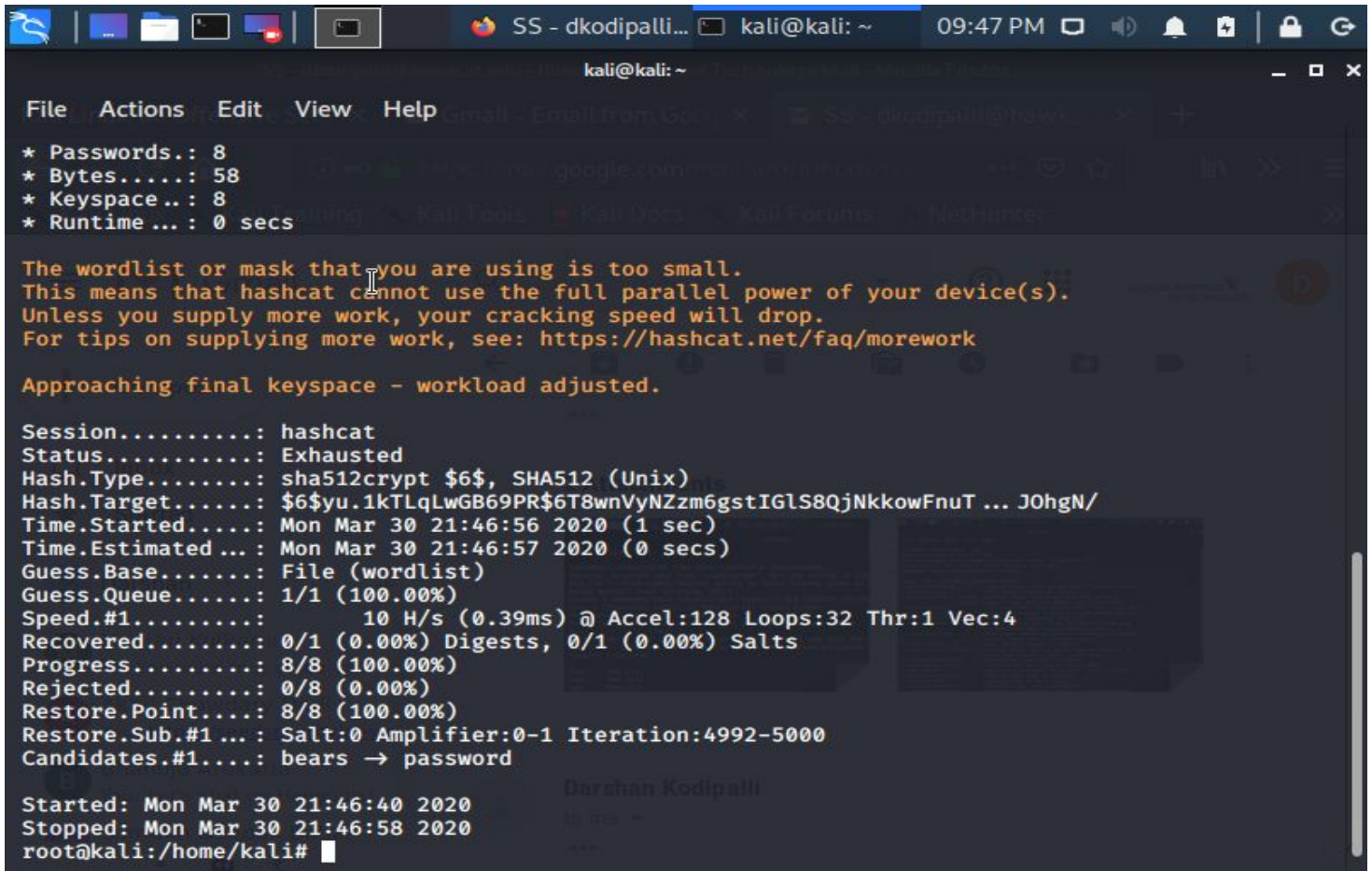
```
kali@kali: ~  
File Actions Edit View Help  
root@kali:/home/kali# useradd bar  
root@kali:/home/kali# passwd bar  
New password:  
Retype new password:  
passwd: password updated successfully  
root@kali:/home/kali#  
root@kali:/home/kali# time john -users:bar /etc/shadow  
Using default input encoding: UTF-8  
Loaded 1 password hash (sha512crypt, crypt(3) $6$ [SHA512 256/256 AVX2 4x])  
Cost 1 (iteration count) is 5000 for all loaded hashes  
Proceeding with single, rules:Single  
Press 'q' or Ctrl-C to abort, almost any other key for status  
Warning: Only 3 candidates buffered for the current salt, minimum 8 needed for performance.  
Warning: Only 2 candidates buffered for the current salt, minimum 8 needed for performance.  
Warning: Only 1 candidate buffered for the current salt, minimum 8 needed for performance.  
Almost done: Processing the remaining buffered candidate passwords, if any.  
Proceeding with wordlist:/usr/share/john/password.lst, rules:Wordlist  
0g 0:00:00:24 22.88% 2/3 (ETA: 21:25:17) 0g/s 1649p/s 1649c/s 1649C/s reliant3.. fuckyou7  
0g 0:00:00:28 26.51% 2/3 (ETA: 21:25:18) 0g/s 1643p/s 1643c/s 1643C/s bob5.. robert5  
0g 0:00:00:31 29.34% 2/3 (ETA: 21:25:18) 0g/s 1644p/s 1644c/s 1644C/s inlove4.. february4  
0g 0:00:00:33 31.22% 2/3 (ETA: 21:25:18) 0g/s 1645p/s 1645c/s 1645C/s master8.. xcountr8  
0g 0:00:00:35 33.11% 2/3 (ETA: 21:25:18) 0g/s 1645p/s 1645c/s 1645C/s burns6.. joy6  
0g 0:00:00:37 34.92% 2/3 (ETA: 21:25:18) 0g/s 1645p/s 1645c/s 1645C/s hydrogen0.. scuba0  
Proceeding with incremental:ASCII  
█
```

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**.

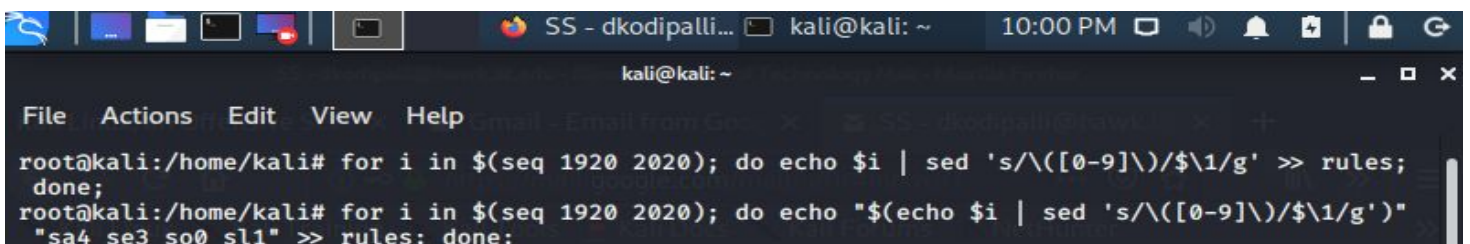


And now we pass these files to the tool **hashcat**. From the output we can see it finished quickly with status as **Executed** but didn't return the right password as the password has the **birth year** at the end and also **3** in place of **E** and **0** in place of **o**.



```
kali@kali: ~  
File Actions Edit View Help  
* Passwords.: 8  
* Bytes.....: 58  
* Keyspace..: 8  
* Runtime ... : 0 secs  
  
The wordlist or mask that you are using is too small.  
This means that hashcat cannot use the full parallel power of your device(s).  
Unless you supply more work, your cracking speed will drop.  
For tips on supplying more work, see: https://hashcat.net/faq/morework  
  
Approaching final keyspace - workload adjusted.  
  
Session.....: hashcat  
Status.....: Exhausted  
Hash.Type.....: sha512crypt $6$, SHA512 (Unix)  
Hash.Target.....: $6$yu.1kTLqLwGB69PR$6T8wnVyNZzm6gstIGLS8QjNkkowFnuT ... JOhgN/  
Time.Started.....: Mon Mar 30 21:46:56 2020 (1 sec)  
Time.Estimated...: Mon Mar 30 21:46:57 2020 (0 secs)  
Guess.Base.....: File (wordlist)  
Guess.Queue.....: 1/1 (100.00%)  
Speed.#1.....: 10 H/s (0.39ms) @ Accel:128 Loops:32 Thr:1 Vec:4  
Recovered.....: 0/1 (0.00%) Digests, 0/1 (0.00%) Salts  
Progress.....: 8/8 (100.00%)  
Rejected.....: 0/8 (0.00%)  
Restore.Point....: 8/8 (100.00%)  
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:4992-5000  
Candidates.#1....: bears -> password  
  
Started: Mon Mar 30 21:46:40 2020  
Stopped: Mon Mar 30 21:46:58 2020  
root@kali:/home/kali#
```

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.



```
kali@kali: ~  
File Actions Edit View Help  
root@kali:/home/kali# for i in $(seq 1920 2020); do echo $i | sed 's/\([0-9]\)/$\1/g' >> rules;  
done;  
root@kali:/home/kali# for i in $(seq 1920 2020); do echo "$(echo $i | sed 's/\([0-9]\)/$\1/g')"  
"sa4 se3 so0 sl1" >> rules; done;
```

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.

```
Watchdog: Hardware monitoring interface not found on your system.
Watchdog: Temperature abort trigger disabled.

* Device #1: build_opts '-cl-std=CL1.2 -I OpenCL -I /usr/share/hashcat/Open
CL -D LOCAL_MEM_TYPE=2 -D VENDOR_ID=64 -D CUDA_ARCH=0 -D AMD_ROCM=0 -D VECT
_SIZE=4 -D DEVICE_TYPE=2 -D DGST_R0=0 -D DGST_R1=1 -D DGST_R2=2 -D DGST_R3=
3 -D DGST_ELEM=16 -D KERN_TYPE=1800 -D _unroll'
Dictionary cache hit:
* Filename..: wordlist
* Passwords.: 5
* Bytes.....: 34
* Keyspace..: 1045

The wordlist or mask that you are using is too small.
This means that hashcat cannot use the full parallel power of your device(s
).
Unless you supply more work, your cracking speed will drop.
For tips on supplying more work, see: https://hashcat.net/faq/morework

Approaching final keyspace - workload adjusted.

$6$f1l2/wwwWJPpDUA2$LpRtxFwkOjWbyZBjRi9M.NMnx4yrllJ0bahnRXR5qkL/H10GWb0LUpy
snj5u731EV05DRSk9Q90xQZ66w5KuM1:whit3s0cks1996
```

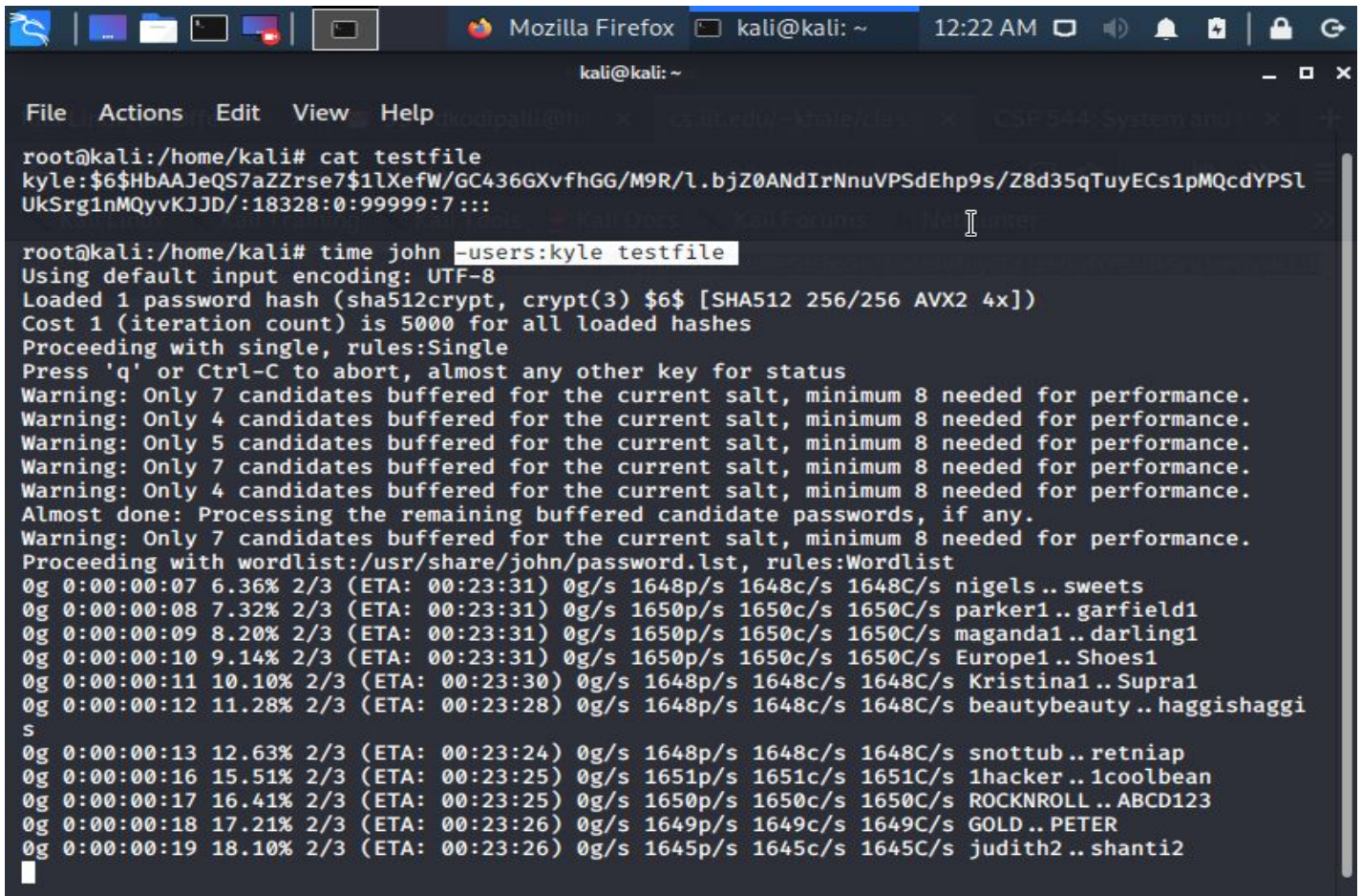
### Task 5 : Putting it Together

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

kyle:\$6\$HbAAJeQS7aZZrse7\$1IXefW/GC436GXvfhGG/M9R/l.bjZ0ANdlrNnuVPSdE  
hp9s/Z8d35qTuyECs1pMQcdYPSIUkSrg1nMQyvKJJJD/: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:





```
root@kali:/home/kali# cat testfile
kyle:$6$HbAAJeQS7aZZrse7$1lXefW/GC436GXvfHG/M9R/l.bjZ0ANDIrNnuVPSdEhp9s/Z8d35qTuyECs1pMQcdYPSl
UkSrg1nMQyvKJJJD/:18328:0:99999:7:::

root@kali:/home/kali# time john -users:kyle testfile
Using default input encoding: UTF-8
Loaded 1 password hash (sha512crypt, crypt(3) $6$ [SHA512 256/256 AVX2 4x])
Cost 1 (iteration count) is 5000 for all loaded hashes
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, almost any other key for status
Warning: Only 7 candidates buffered for the current salt, minimum 8 needed for performance.
Warning: Only 4 candidates buffered for the current salt, minimum 8 needed for performance.
Warning: Only 5 candidates buffered for the current salt, minimum 8 needed for performance.
Warning: Only 7 candidates buffered for the current salt, minimum 8 needed for performance.
Warning: Only 4 candidates buffered for the current salt, minimum 8 needed for performance.
Almost done: Processing the remaining buffered candidate passwords, if any.
Warning: Only 7 candidates buffered for the current salt, minimum 8 needed for performance.
Proceeding with wordlist:/usr/share/john/password.lst, rules:Wordlist
0g 0:00:00:07 6.36% 2/3 (ETA: 00:23:31) 0g/s 1648p/s 1648c/s 1648C/s nigels..sweets
0g 0:00:00:08 7.32% 2/3 (ETA: 00:23:31) 0g/s 1650p/s 1650c/s 1650C/s parker1..garfield1
0g 0:00:00:09 8.20% 2/3 (ETA: 00:23:31) 0g/s 1650p/s 1650c/s 1650C/s maganda1..darling1
0g 0:00:00:10 9.14% 2/3 (ETA: 00:23:31) 0g/s 1650p/s 1650c/s 1650C/s Europe1..Shoes1
0g 0:00:00:11 10.10% 2/3 (ETA: 00:23:30) 0g/s 1648p/s 1648c/s 1648C/s Kristina1..Supra1
0g 0:00:00:12 11.28% 2/3 (ETA: 00:23:28) 0g/s 1648p/s 1648c/s 1648C/s beautybeauty..haggishaggi
s
0g 0:00:00:13 12.63% 2/3 (ETA: 00:23:24) 0g/s 1648p/s 1648c/s 1648C/s snottub..retniap
0g 0:00:00:16 15.51% 2/3 (ETA: 00:23:25) 0g/s 1651p/s 1651c/s 1651C/s 1hacker..1coolbean
0g 0:00:00:17 16.41% 2/3 (ETA: 00:23:25) 0g/s 1650p/s 1650c/s 1650C/s ROCKNROLL..ABCD123
0g 0:00:00:18 17.21% 2/3 (ETA: 00:23:26) 0g/s 1649p/s 1649c/s 1649C/s GOLD..PETER
0g 0:00:00:19 18.10% 2/3 (ETA: 00:23:26) 0g/s 1645p/s 1645c/s 1645C/s judith2..shanti2
```

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.

```

kali@kali: ~
File Actions Edit View Help
kali@kali: ~
root@kali:/home/kali# cat possiblepasswords
hexsa
professor
research
mascos
icac
teaching
funding
kylehale
root@kali:/home/kali# hashcat --force -m 1800 testfile possiblepasswords
hashcat (v5.1.0) starting...

OpenCL Platform #1: The pocl project
=====
* Device #1: pthread-Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1024/2955 MB allocatable, 1MCU

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Applicable optimizers:
* Zero-Byte
* Single-Hash
* Single-Salt
* Uses-64-Bit

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256

```

```

kali@kali: ~
File Actions Edit View Help
kali@kali: ~
* Keyspace .. : 8
* Runtime ... : 0 secs

The wordlist or mask that you are using is too small.
This means that hashcat cannot use the full parallel power of your device(s).
Unless you supply more work, your cracking speed will drop.
For tips on supplying more work, see: https://hashcat.net/faq/morework

Approaching final keyspace - workload adjusted.

Session.....: hashcat
Status.....: Exhausted
Hash.Type.....: sha512crypt $6$, SHA512 (Unix)
Hash.Target.....: $6$HbAAJeQS7aZZrse7$1lXefW/GC436GXvfHG/M9R/l.bjZ0A ... vKJJD/
Time.Started.....: Tue Mar 31 01:07:14 2020 (1 sec)
Time.Estimated...: Tue Mar 31 01:07:15 2020 (0 secs)
Guess.Base.....: File (possiblepasswords)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 10 H/s (0.39ms) @ Accel:128 Loops:32 Thr:1 Vec:4
Recovered.....: 0/1 (0.00%) Digests, 0/1 (0.00%) Salts
Progress.....: 8/8 (100.00%)
Rejected.....: 0/8 (0.00%)
Restore.Point....: 8/8 (100.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:4992-5000
Candidates.#1....: hexsa -> kylehale

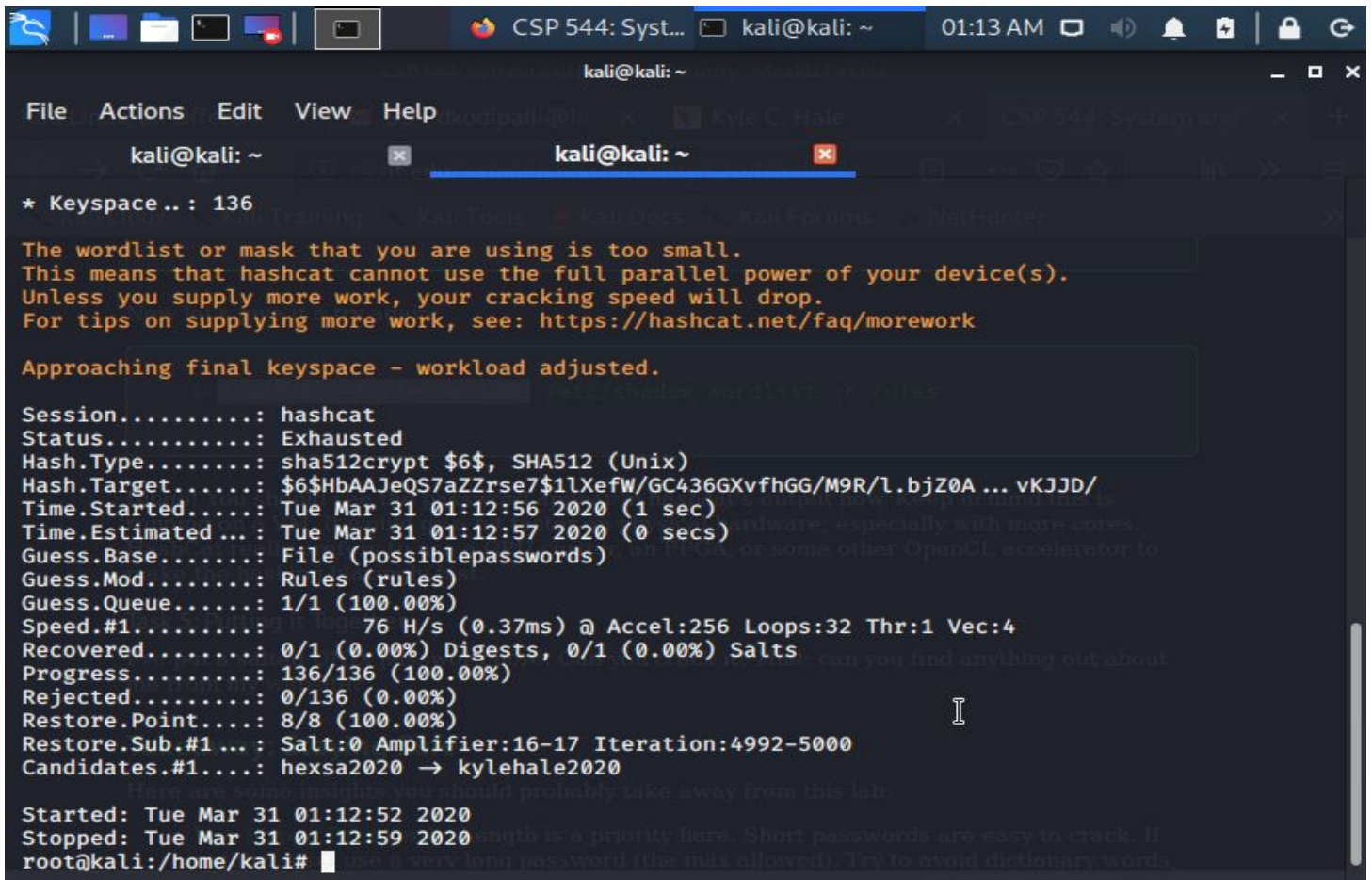
Started: Tue Mar 31 01:07:11 2020
Stopped: Tue Mar 31 01:07:16 2020
root@kali:/home/kali#

```



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
```



```
kali@kali: ~
File Actions Edit View Help
kali@kali: ~ kali@kali: ~
* Keyspace... : 136
The wordlist or mask that you are using is too small.
This means that hashcat cannot use the full parallel power of your device(s).
Unless you supply more work, your cracking speed will drop.
For tips on supplying more work, see: https://hashcat.net/faq/morework
Approaching final keypace - workload adjusted.
Session.....: hashcat
Status.....: Exhausted
Hash.Type.....: sha512crypt $6$, SHA512 (Unix)
Hash.Target.....: $6$HbAAJeQS7aZZrse7$1lXefW/GC436GXvfHGG/M9R/l.bjZ0A ... vKJJD/
Time.Started.....: Tue Mar 31 01:12:56 2020 (1 sec)
Time.Estimated...: Tue Mar 31 01:12:57 2020 (0 secs)
Guess.Base.....: File (possiblepasswords)
Guess.Mod.....: Rules (rules)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 76 H/s (0.37ms) @ Accel:256 Loops:32 Thr:1 Vec:4
Recovered.....: 0/1 (0.00%) Digests, 0/1 (0.00%) Salts
Progress.....: 136/136 (100.00%)
Rejected.....: 0/136 (0.00%)
Restore.Point....: 8/8 (100.00%)
Restore.Sub.#1...: Salt:0 Amplifier:16-17 Iteration:4992-5000
Candidates.#1....: hexsa2020 -> kylehale2020
Started: Tue Mar 31 01:12:52 2020
Stopped: Tue Mar 31 01:12:59 2020
root@kali:/home/kali#
```

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.