Graham Dungan
March 23, 2025
UIN: 332001764

CSCE 413: Software Security
Class 24: Password Cracking

Included Files

Due to the nature of this assignment, there is no demo file associated with this report. Instead, the scripts and password files have been included. What follows is a list of included files and their uses;

- `mypasswd` - The unshadowed password hashes of all test users.
- `create-user.sh` - A script for creating a user with an n length random digit password.
- `passwords.txt` - A list of the passwords for all generated test users.
- `run_jtr.sh` - A script to automate running John the Ripper and recording its time for a test user.
- `john.pot` - A pot file containing the cracked hashes of the test users.
- `times.txt` - A text file containing the times taken to crack passwords.

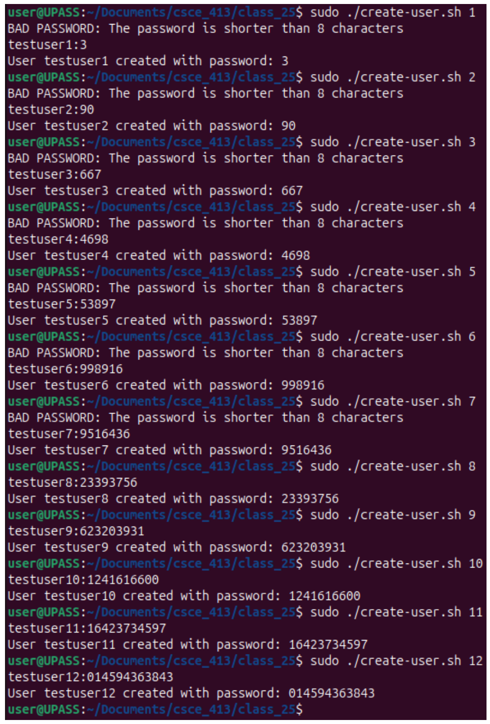
Creating Test Users

For this assignment, I used John the Ripper Jumbo on Ubuntu LTS 24.04.2 LTS. To automate the creation of users, I created the script `create-user.sh`.

```
#!/bin/bash
i=$1
USERNAME="testuser$i"
PASSWORD=$(cat /dev/urandom | tr -dc '0-9' | head -c $i)
useradd -m -s /bin/bash "$USERNAME"
echo "$USERNAME:$PASSWORD" | chpasswd
echo "$USERNAME:$PASSWORD" | tee -a passwords.txt
echo "User $USERNAME created with password: $PASSWORD"
```

This script creates a user named "testuser#", where the number at the end of the user's name represents the length of the password. The script will use the input number to generate a n-length password composed of n random digits. It will then write the username, password pair to a file named `passwords.txt`.

As seen here, I generated 12 passwords for this demonstration.



I chose to use n-length digit-only passwords for this assignment to decrease the keyspace for demonstrative purposes. Decreasing the keyspace allows for a smaller search yet still demonstrates the exponential nature of password cracking versus password length. Suppose, instead, I used the entire ASCII character set for passwords. There are 128 ASCII characters, so for every character in a password, there are 128 choices. For an n-length password, this would result in 128^n possible passwords. Limiting the password characters to digits 0-9, there are only 10 possible choices per character, resulting in 10^n possible passwords. To avoid spending needless hours demonstrating the same behavior with the ASCII set, we use a smaller digits set.

Dumping `passwd` and `shadow`

John the Ripper (JtR) requires the `passwd` and `shadow` files in order to begin cracking passwords. Originally, password hashes were stored in the `/etc/passwd` file in UNIX, however, this made them world-readable. To avoid this, user information is kept in `/etc/passwd`, and corresponding password hashes are kept in `/etc/shadow`. As per JtR documentation, these files can be "unshadowed" via;

```
sudo ~/src/john/run/unshadow /etc/passwd /etc/shadow | grep -E "^testuser[0-9].*$" > mypasswd
```

The first part of this command runs the JtR `unshadow` command with `/etc/passwd` and `/etc/shadow`. It then uses grep to obtain any line that includes the `testuser`s created earlier. It then redirects this output to a file called `mypasswd`, which can be found below;

```
testuser1:$y$j9T$bPYiTV8zR01fL46lD4RzR/$o1yKDf6zSY6SPzgwP5hr4a/gi0l7mqITD9nw0fcJeN5:1001:1001::/home/testuser1:/bin/bash
testuser2:$y$j9T$ekVWFqUpovpFoyXaPYGij.$ytajtg3KDhE3jGSTfuGfodGcmEcRHYpipbTwGEfCheB:1002:1002::/home/testuser2:/bin/bash
testuser3:$y$j9T$hQ1gmo.wYT6.9mGi9hV9M1$Kfn4hlzT0nBPYcT45XuhxtZcVLabRqkl2lnVgyDQqc4:1003:1003::/home/testuser3:/bin/bash
testuser4:$y$j9T$3NFMyNMHUJQVFFwofm0y/.$fB9bpPjCksAQdBAYd0EJXNXZLt0bTQMGlF0U413Oc2.:1004:1004::/home/testuser4:/bin/bash
testuser5:$y$j9T$CWFWohAYR19ZpKfZyGoC41$vQuiwv2RaihE7THjTbOWf4P.UYLhYH9WDoEc0twAgv.:1005:1005::/home/testuser5:/bin/bash
testuser6:$y$j9T$p7W3FKPiNO4s5p87qC9up.$M80ezvHAZJB16ZCU2Qg.XFSdq9rk8peURl4bT5DwTv9:1006:1006::/home/testuser6:/bin/bash
testuser7:$y$j9T$bzPIgZffml4cMzFZuXHGs.$.5tw7Dzl3.bavHaoq20v229IyPJ7uGabOMgIOqLm0qD:1007:1007::/home/testuser7:/bin/bash
testuser8:$y$j9T$dQDEWqAnH6rQTqFZBCN3s1$tgWFxVJ3jzVFwQnALRfjqEP.aF2Xu8s3Ovp3.Cr9BHA:1008:1008::/home/testuser8:/bin/bash
testuser9:$y$j9T$o7oQyS1pMRGIqmQOu0POa/$7vpKf9h.0MYym1VZpHZ3HKvpLhAptSY6tpK1Lo//XS9:1009:1009::/home/testuser9:/bin/bash
testuser10:$y$j9T$2MxuTJQCnFTLokFYCuSYo/$xQkQc/dBYrGG.nRlEETBxPqNWiSMwEJSyMbSGqpY4GD:1010:1010::/home/testuser10:/bin/bash
testuser11:$y$j9T$z8LXZA.Au6wE3vxFeyAbC/$7lvbNNyWGoqZihSWKoWLhKa/2.zwiU/2euWUBYkMOAC:1011:1011::/home/testuser11:/bin/bash
testuser12:$y$j9T$Olzun08ZlDOQNVJcxQQJ5/$4H4f3YrUuUIfw.Iky0MKF4W15.B3q9fnsQJQ8pfwZC0:1012:1012::/home/testuser12:/bin/bash
```

This tells us the hash of the password for each user and the type of hash. The `$y$` at the beginning of these hashes indicates that hashes use yescrypt to hash the passwords, as specified by Ubuntu's password hashing security feature. JtR uses crypt/generic crypt(3) for cracking yescrypt. What follows is a screenshot of the output of this command,

```
user@UPASS:~/Documents/csce_413/class_25$ sudo ~/src/john/run/unshadow /etc/passwd /etc/shadow | grep -E "^testuser[0-9].*$" > mypasswd
user@UPASS:~/Documents/csce_413/class_25$ cat mypasswd
testuser1:$y$j9T$bPYiTV8zR01fL46lD4RzR/$o1yKDf6zSY6SPzgwP5hr4a/gi0l7mqITD9nw0fcJeN5:1001:1001::/home/testuser1:/bin/bash
testuser2:$y$j9T$ekVWFqUpovpFoyXaPYGij.$ytajtg3KDhE3jGSTfuGfodGcmEcRHYpipbTwGEfCheB:1002:1002::/home/testuser2:/bin/bash
testuser3:$y$j9T$hQ1gmo.wYT6.9mGi9hV9M1$Kfn4hlzT0nBPYcT45XuhxtZcVLabRqkl2lnVgyDQqc4:1003:1003::/home/testuser3:/bin/bash
testuser4:$y$j9T$3NFMyNMHUJQVFFwofm0y/.$fB9bpPjCksAQdBAYd0EJXNXZLt0bTQMGlF0U413Oc2.:1004:1004::/home/testuser4:/bin/bash
testuser5:$y$j9T$CWFWohAYR19ZpKfZyGoC41$vQuiwv2RaihE7THjTbOWf4P.UYLhYH9WDoEc0twAgv.:1005:1005::/home/testuser5:/bin/bash
testuser6:$y$j9T$p7W3FKPiNO4s5p87qC9up.$M80ezvHAZJB16ZCU2Qg.XFSdq9rk8peURl4bT5DwTv9:1006:1006::/home/testuser6:/bin/bash
testuser7:$y$j9T$bzPIgZffml4cMzFZuXHGs.$.5tw7Dzl3.bavHaoq20v229IyPJ7uGabOMgIOqLm0qD:1007:1007::/home/testuser7:/bin/bash
testuser8:$y$j9T$dQDEWqAnH6rQTqFZBCN3s1$tgWFxVJ3jzVFwQnALRfjqEP.aF2Xu8s3Ovp3.Cr9BHA:1008:1008::/home/testuser8:/bin/bash
testuser9:$y$j9T$o7oQyS1pMRGIqmQOu0POa/$7vpKf9h.0MYym1VZpHZ3HKvpLhAptSY6tpK1Lo//XS9:1009:1009::/home/testuser9:/bin/bash
testuser10:$y$j9T$2MxuTJQCnFTLokFYCuSYo/$xQkQc/dBYrGG.nRlEETBxPqNWiSMwEJSyMbSGqpY4GD:1010:1010::/home/testuser10:/bin/bash
testuser11:$y$j9T$z8LXZA.Au6wE3vxFeyAbC/$7lvbNNyWGoqZihSWKoWLhKa/2.zwiU/2euWUBYkMOAC:1011:1011::/home/testuser11:/bin/bash
testuser12:$y$j9T$Olzun08ZlDOQNVJcxQQJ5/$4H4f3YrUuUIfw.Iky0MKF4W15.B3q9fnsQJQ8pfwZC0:1012:1012::/home/testuser12:/bin/bash
user@UPASS:~/Documents/csce_413/class_25$
```

Using John the Ripper

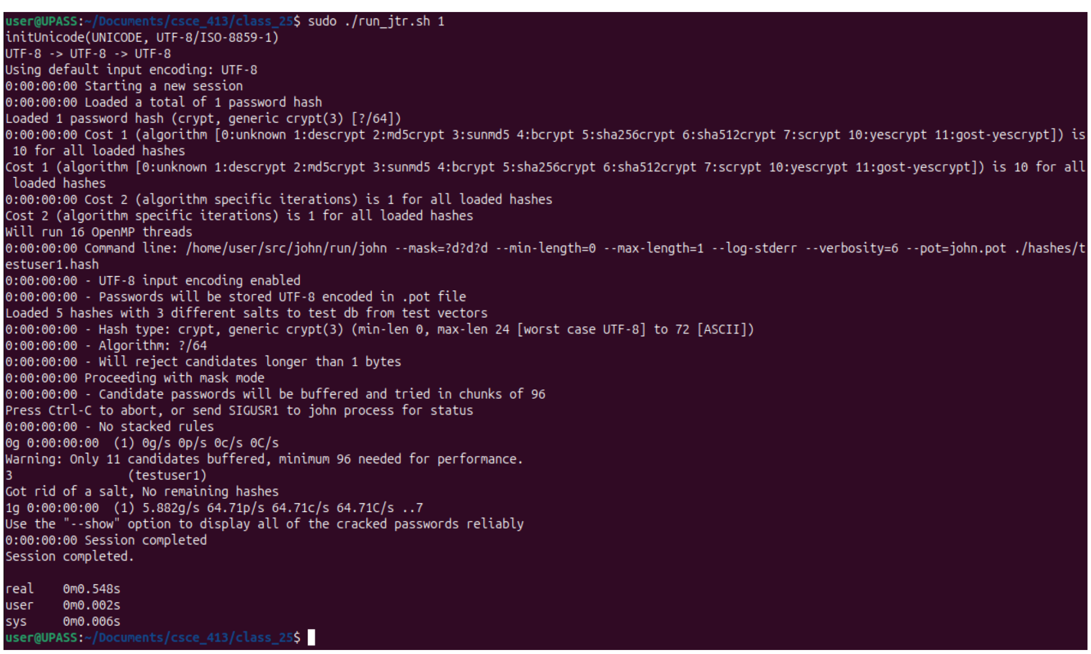
JtR offers multiple modes for password cracking. Incremental mode offers the ability to try all possible character combinations as passwords. However, due to the fact that I have digit-only passwords as a means to reduce the keyspace, I will be using the mask mode (a similar effect could have been achieved by specifying a digit charset for incremental mode). I created a simple script named `run_jtr.sh` to record the runtime of cracking each password,

```
#!/bin/bash
line=$(sed -n "${1}p" mypasswd)
echo "$line" > "./hashes/testuser${1}.hash"
{
    time sudo /home/user/src/john/run/john --mask=?d?d?d --min-length=0 --max-length=$1 --log-
        stderr --verbosity=6 --pot=john.pot "./hashes/testuser${1}.hash"
} 2>&1 | tee output.txt
echo -e "testuser${1}:" >> times.txt
grep -E "real|user|sys" output.txt | tail -3 >> times.txt
echo "" >> times.txt
rm output.txt
```

This script accepts a number (indicating which testuser to select from the unshadowed passwords file `mypasswd`) to crack. It then stores this hash in a separate hash file and runs JtR with the time command to record its time. What follows are the components of the JtR command;

- `--mask=?d?d?d` applies a mask of digits. This is used to generate strings of digits to attempt guessing.
- `--min-length=0 --max-length=$1` specifies that the minimum length of the password is 0, and the max length is the length specified by the user/testuser. Since this mode increments the guess, it wouldn't necessarily need a max length as it would eventually find it in that space.
- `--log-stderr --verbosity=6` enables logging for debugging purposes.
- `--pot=john.pot` specifies that when the password is cracked, it is stored in a local file `john.pot`. Typically this is kept elsewhere in the system.
- `"./hashes/testuser${1}.hash"` specifies that we will use the hash previously generated by the script for cracking.

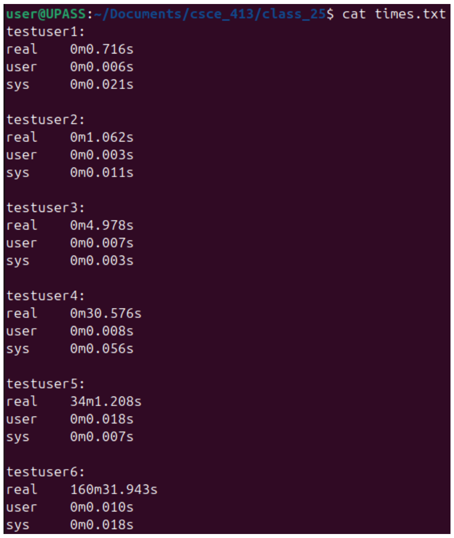
The remainder of this script obtains the output of the `time` command and appends it to the file called `times.txt`. What follows is a screenshot of a length-1 password being cracked,



```
user@UPASS:~/Documents/csce_413/class_25$ sudo ./run_jtr.sh 1
initUnicode(UNICODE, UTF-8/ISO-8859-1)
UTF-8 -> UTF-8 -> UTF-8
Using default input encoding: UTF-8
0:00:00:00 Starting a new session
0:00:00:00 Loaded a total of 1 password hash
Loaded 1 password hash (crypt, generic crypt(3) [?/64])
0:00:00:00 Cost 1 (algorithm [0:unknown 1:descrypt 2:md5crypt 3:sunmd5 4:bcrypt 5:sha256crypt 6:sha512crypt 7:scrypt 10:yescrypt 11:gost-yescrypt]) is
 10 for all loaded hashes
Cost 1 (algorithm [0:unknown 1:descrypt 2:md5crypt 3:sunmd5 4:bcrypt 5:sha256crypt 6:sha512crypt 7:scrypt 10:yescrypt 11:gost-yescrypt]) is 10 for all
 loaded hashes
0:00:00:00 Cost 2 (algorithm specific iterations) is 1 for all loaded hashes
Cost 2 (algorithm specific iterations) is 1 for all loaded hashes
Will run 16 OpenMP threads
0:00:00:00 Command line: /home/user/src/john/run/john --mask=?d?d?d --min-length=0 --max-length=1 --log-stderr --verbosity=6 --pot=john.pot ./hashes/t
estuser1.hash
0:00:00:00 - UTF-8 input encoding enabled
0:00:00:00 - Passwords will be stored UTF-8 encoded in .pot file
Loaded 5 hashes with 3 different salts to test db from test vectors
0:00:00:00 - Hash type: crypt, generic crypt(3) (min-len 0, max-len 24 [worst case UTF-8] to 72 [ASCII])
0:00:00:00 - Algorithm: ?/64
0:00:00:00 - Will reject candidates longer than 1 bytes
0:00:00:00 Proceeding with mask mode
0:00:00:00 - Candidate passwords will be buffered and tried in chunks of 96
Press Ctrl-C to abort, or send SIGUSR1 to john process for status
0:00:00:00 - No stacked rules
0g 0:00:00:00  (1) 0g/s 0p/s 0c/s 0C/s
Warning: Only 11 candidates buffered, minimum 96 needed for performance.
3                (testuser1)
Got rid of a salt, No remaining hashes
1g 0:00:00:00  (1) 5.882g/s 64.71p/s 64.71c/s 64.71C/s ..7
Use the "--show" option to display all of the cracked passwords reliably
0:00:00:00 Session completed
Session completed.

real    0m0.548s
user    0m0.002s
sys     0m0.006s
user@UPASS:~/Documents/csce_413/class_25$
```

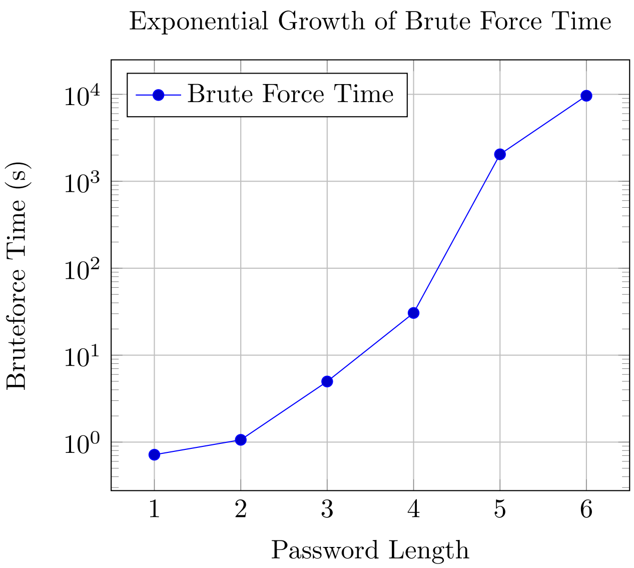
Brute-Force Time Analysis

I have run `run_jtr.sh` on testusers 1-6, as the time to crack these passwords had exponentially increased to a point where it would be infeasible to continue running this program for such a demonstration. We can view the password cracking times by examining the aforementioned `times.txt` file,



```
user@UPASS:~/Documents/csce_413/class_25$ cat times.txt
testuser1:
real	0m0.716s
user	0m0.006s
sys	0m0.021s

testuser2:
real	0m1.062s
user	0m0.003s
sys	0m0.011s

testuser3:
real	0m4.978s
user	0m0.007s
sys	0m0.003s

testuser4:
real	0m30.576s
user	0m0.008s
sys	0m0.056s

testuser5:
real	34m1.208s
user	0m0.018s
sys	0m0.007s

testuser6:
real	160m31.943s
user	0m0.010s
sys	0m0.018s
```

We can now plot these times as a function of password length,



This plot displays the increasing time needed to crack n-length passwords on a logarithmic scale. Mathematically, we know that each character in a password can assume one of ten values, 0-9. Since each character in an n-length password has ten choices, a n-length password has `10^n` possible values. It follows that for passwords of length 1-6, the keyspace will grow from 10 to 100, 1,000, 10,000, 100,000, and 1,000,000. The reason for the outlier in time when the password length was 6 could have been that JtR's guessing scheme prioritized a specific set of numbers and "got lucky". Ultimately, it is seen that the time to discover passwords of increasing complexities is exponential.