ICS344

Password Toolkit

HW2

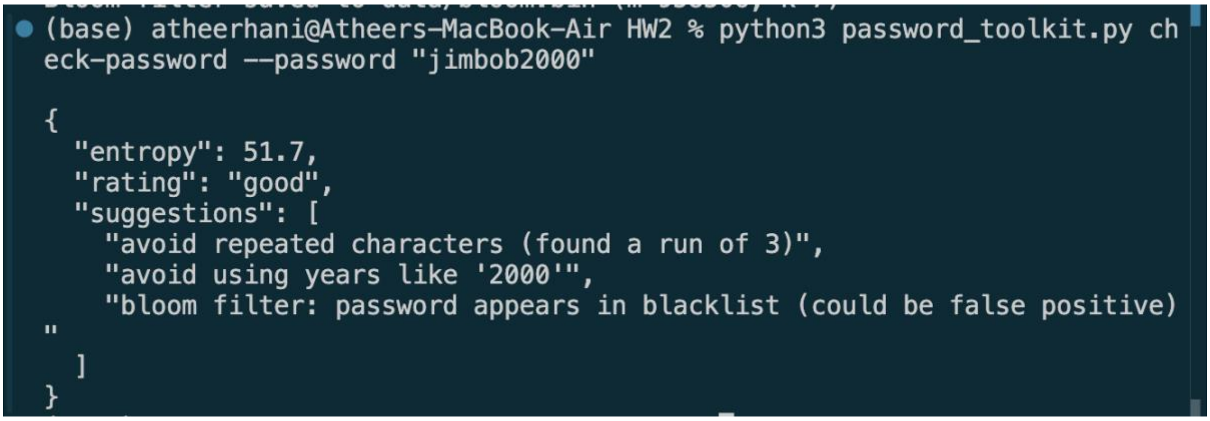
202246520

Atheer Almomtan

Note: three demo runs were asked from me but I wanted to include all the commands that were mentioned in the instructions file that's why there are six demo runs.

Demo run #1: Password Strength check

Command: python password_toolkit.py check-password –password jimbob2000



```
● (base) atheerhani@Atheers-MacBook-Air HW2 % python3 password_toolkit.py ch
  eck-password --password "jimbob2000"

  {
    "entropy": 51.7,
    "rating": "good",
    "suggestions": [
      "avoid repeated characters (found a run of 3)",
      "avoid using years like '2000'",
      "bloom filter: password appears in blacklist (could be false positive)
  "
    ]
  }
```

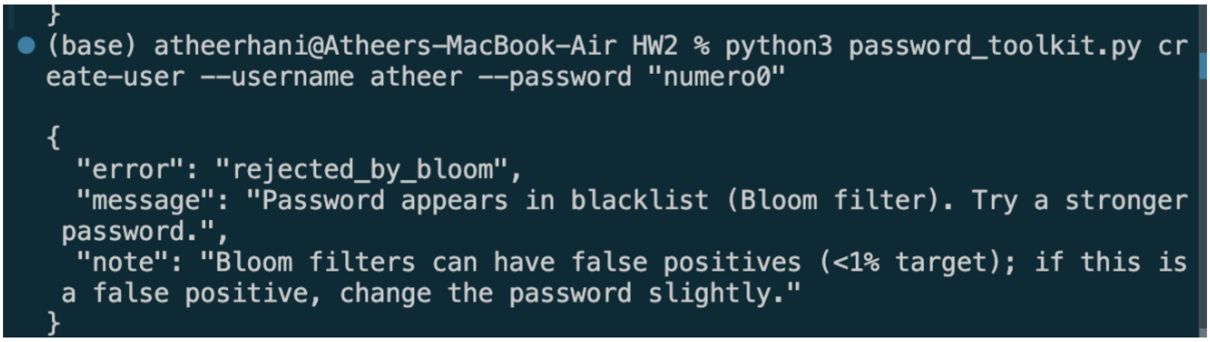
The system calculated an **entropy of ~51.7 bits**, giving the password a **"good" rating**.

However, it detected weak patterns:

- Repeated characters
- A 4-digit year (2000)
- A **Bloom filter hit**, meaning the password was found (or falsely flagged) in the blacklist.

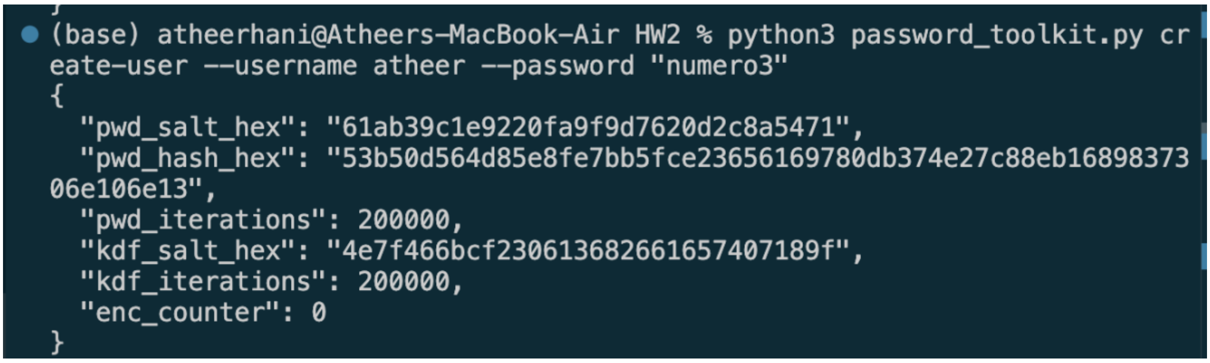
Demo run #2: User creation and bloom rejection

Command: python3 password_toolkit.py create-user --username atheer --password numero0



```
  }
● (base) atheerhani@Atheers-MacBook-Air HW2 % python3 password_toolkit.py cr
  eate-user --username atheer --password "numero0"

  {
    "error": "rejected_by_bloom",
    "message": "Password appears in blacklist (Bloom filter). Try a stronger
   password.",
    "note": "Bloom filters can have false positives (<1% target); if this is
   a false positive, change the password slightly."
  }
```

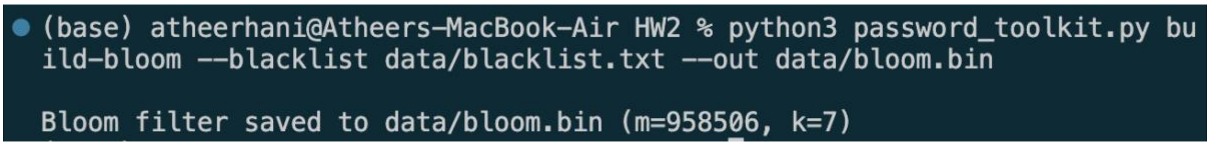


```
● (base) atheerhani@Atheers-MacBook-Air HW2 % python3 password_toolkit.py cr
  eate-user --username atheer --password "numero3"
  {
    "pwd_salt_hex": "61ab39c1e9220fa9f9d7620d2c8a5471",
    "pwd_hash_hex": "53b50d564d85e8fe7bb5fce23656169780db374e27c88eb16898373
  06e106e13",
    "pwd_iterations": 200000,
    "kdf_salt_hex": "4e7f466bcf230613682661657407189f",
    "kdf_iterations": 200000,
    "enc_counter": 0
  }
```

Here I changed one character in the passwrod and the user was created

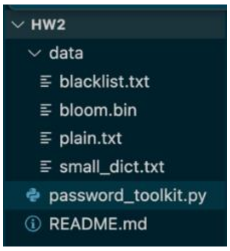
Demo run #3: Building the bloom filter

Command: python3 password_toolkit.py build-bloom --blacklist data/blacklist.txt --out data/bloom.bin



```
● (base) atheerhani@Atheers-MacBook-Air HW2 % python3 password_toolkit.py bu
  ild-bloom --blacklist data/blacklist.txt --out data/bloom.bin

  Bloom filter saved to data/bloom.bin (m=958506, k=7)
```



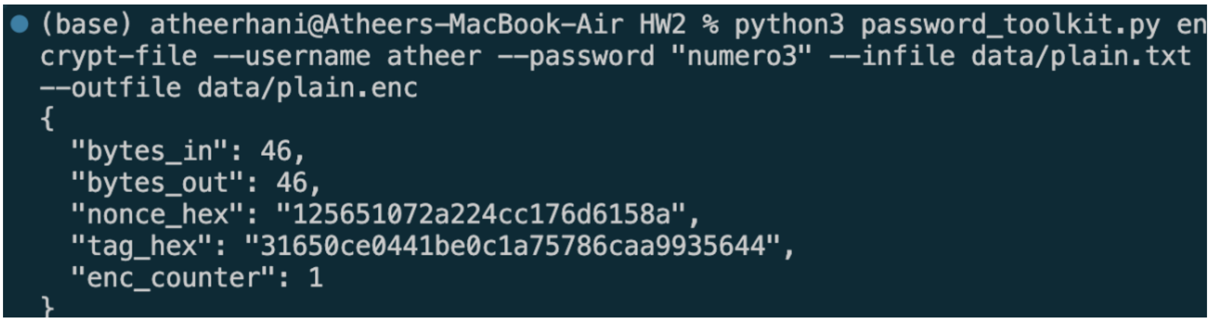
```
∨ HW2
  ∨ data
      blacklist.txt
      bloom.bin
      plain.txt
      small_dict.txt
    password_toolkit.py
    README.md
```

The toolkit built a Bloom filter with parameters:

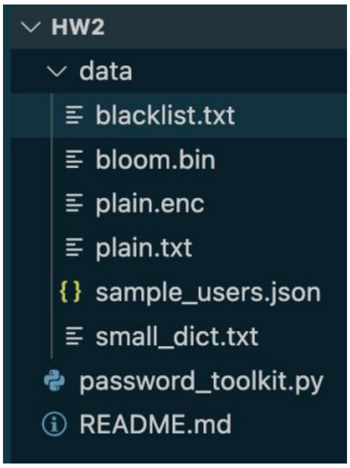
m = 958506 bits, k = 7 hash functions

Demo run#4: File Encryption

Command: python3 password_toolkit.py encrypt-file --username atheer --password numero3 --infile data/plain.txt --outfile data/plain.enc



```
● (base) atheerhani@Atheers-MacBook-Air HW2 % python3 password_toolkit.py en
  crypt-file --username atheer --password "numero3" --infile data/plain.txt
  --outfile data/plain.enc
  {
    "bytes_in": 46,
    "bytes_out": 46,
    "nonce_hex": "125651072a224cc176d6158a",
    "tag_hex": "31650ce0441be0c1a75786caa9935644",
    "enc_counter": 1
  }
```

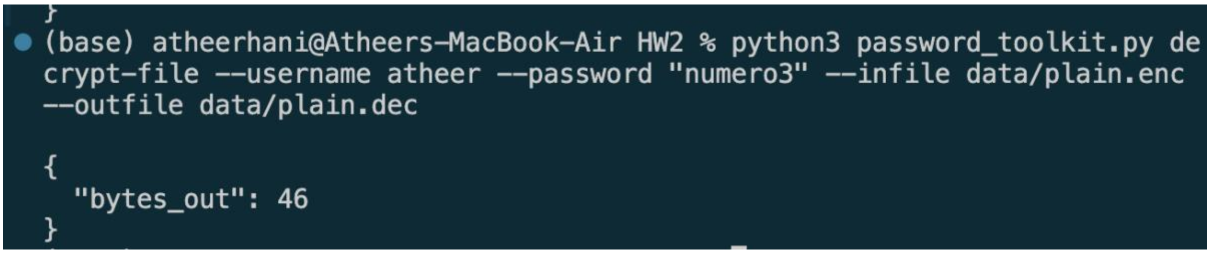


```
HW2
  data
    blacklist.txt
    bloom.bin
    plain.enc
    plain.txt
    sample_users.json
    small_dict.txt
  password_toolkit.py
  README.md
```

This demonstrates that the encryption module works as intended, generating a **random nonce**, an **authentication tag**, and incrementing the encryption counter.

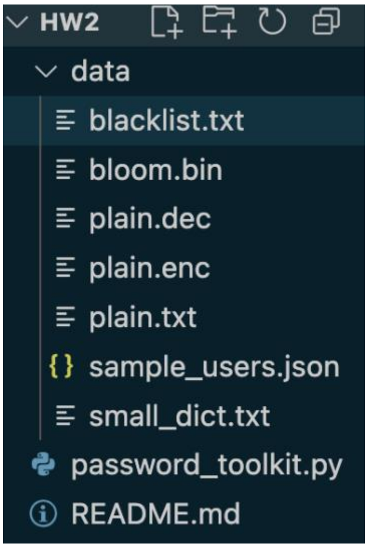
Demo run#5: File Decryption(positive test)

Command: python3 password_toolkit.py decrypt-file --username atheer --password numero3 --infile data/plain.enc --outfile data/plain.dec



```
}
● (base) atheerhani@Atheers-MacBook-Air HW2 % python3 password_toolkit.py de
  crypt-file --username atheer --password "numero3" --infile data/plain.enc
  --outfile data/plain.dec

  {
    "bytes_out": 46
  }
```

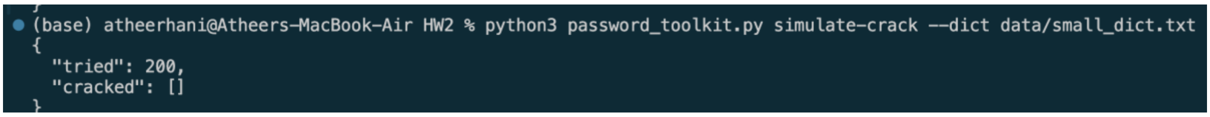


```
HW2
  data
    blacklist.txt
    bloom.bin
    plain.dec
    plain.enc
    plain.txt
    sample_users.json
    small_dict.txt
  password_toolkit.py
  README.md
```

The decrypted file matched the original, **Positive Test Passed**

Demo run#6: Cracker Simulation

Command: python3 password_toolkit.py simulate-crack --dict data/small_dict.txt



```
● (base) atheerhani@Atheers-MacBook-Air HW2 % python3 password_toolkit.py simulate-crack --dict data/small_dict.txt
  {
    "tried": 200,
    "cracked": []
  }
```

Cracker results: This shows that no stored user credentials were cracked. Why is that? Because as seen in the previous demos, when I tried to create a user with the password “numero0” it was rejected in the first place because it was foind the blacklist file. So, I used “numero3” insteas which was not in the file, hence there are no cracked passwords. If I were to try other passwords and they were in the small_dict.txt file, the cracker simulator would crack the password.

Encryption choices and Misuse notes:

Encryption choices§:

- My program uses **PBKDF2-HMAC-SHA256** to create a strong key from the user’s password.
- Each user has two random salts:
  - One salt is used for checking the password.
  - Another salt is used for creating the encryption key.

This separation makes the system safer.

- The encryption method is **AES-GCM**, which is a modern and secure standard.
  - It protects the data and checks that it has not been changed.
  - A **new random nonce** (a one-time number) is made for every encryption.
  - A **tag** is also created to make sure that the file was not tampered with.

What can go wrong (misuse notes):

- If I reuse the same nonce for more than one encryption with the same password, the data can become unsafe. The program prevents this by always creating a new random nonce.
- If I enter the wrong password, the decryption will fail, which is good because it protects against attackers.
- If even one bit in the encrypted file changes, it will fail to open because the tag check will not match.

- If I change the AAD (extra data that must match during decryption), it will fail to open.
- If I use a weak or common password, the encryption itself will still be strong, but someone could guess my password. That is why the Bloom filter blocks weak ones.
- If I lower the PBKDF2 iteration count, password cracking becomes easier because it takes less time to test each password guess.