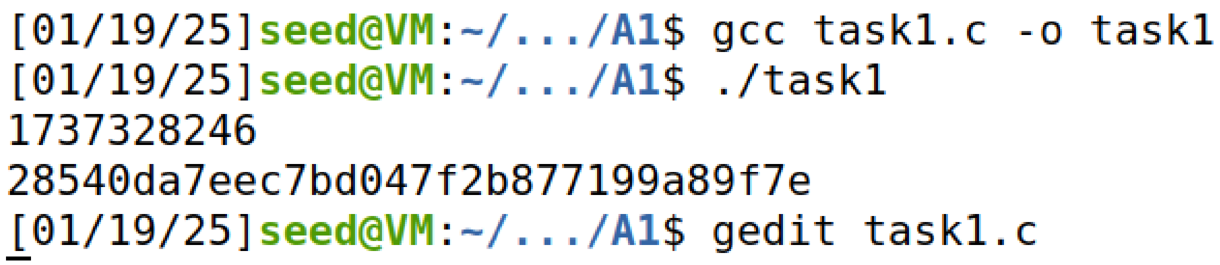
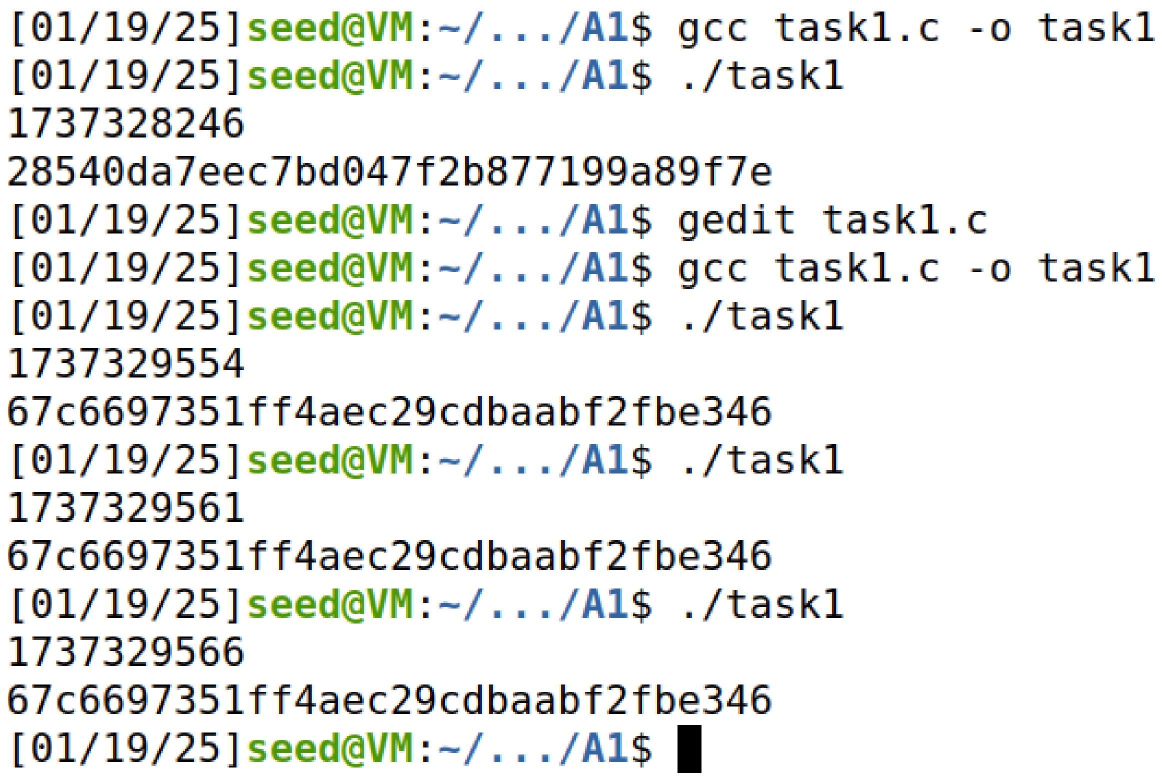
Task 1:

After running the given code, 2 numbers are outputted. The first being the time since the Unix Epoch, the second being the random 16 byte key in hexadecimal format.



```
[01/19/25]seed@VM:~/.../A1$ gcc task1.c -o task1
[01/19/25]seed@VM:~/.../A1$ ./task1
1737328246
28540da7eec7bd047f2b877199a89f7e
[01/19/25]seed@VM:~/.../A1$ gedit task1.c
```

After commenting out the srand() line, the seed remains the same every time the code is run, resulting in the same key being output every time, with the first number still being the time since Epoch.

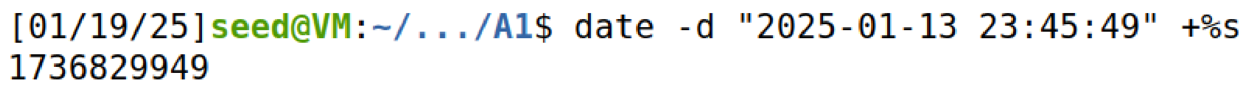


```
[01/19/25]seed@VM:~/.../A1$ gcc task1.c -o task1
[01/19/25]seed@VM:~/.../A1$ ./task1
1737328246
28540da7eec7bd047f2b877199a89f7e
[01/19/25]seed@VM:~/.../A1$ gedit task1.c
[01/19/25]seed@VM:~/.../A1$ gcc task1.c -o task1
[01/19/25]seed@VM:~/.../A1$ ./task1
1737329554
67c6697351ff4aec29cdbaabf2fbe346
[01/19/25]seed@VM:~/.../A1$ ./task1
1737329561
67c6697351ff4aec29cdbaabf2fbe346
[01/19/25]seed@VM:~/.../A1$ ./task1
1737329566
67c6697351ff4aec29cdbaabf2fbe346
[01/19/25]seed@VM:~/.../A1$
```

The usage of `time` and `srand(t)` are important because time gives: a unique and unpredictable number per run, while srand() uses this number as the seed to generate the key. The alternative without srand() uses the same seed value for random number generation resulting in the same key per run of the code.

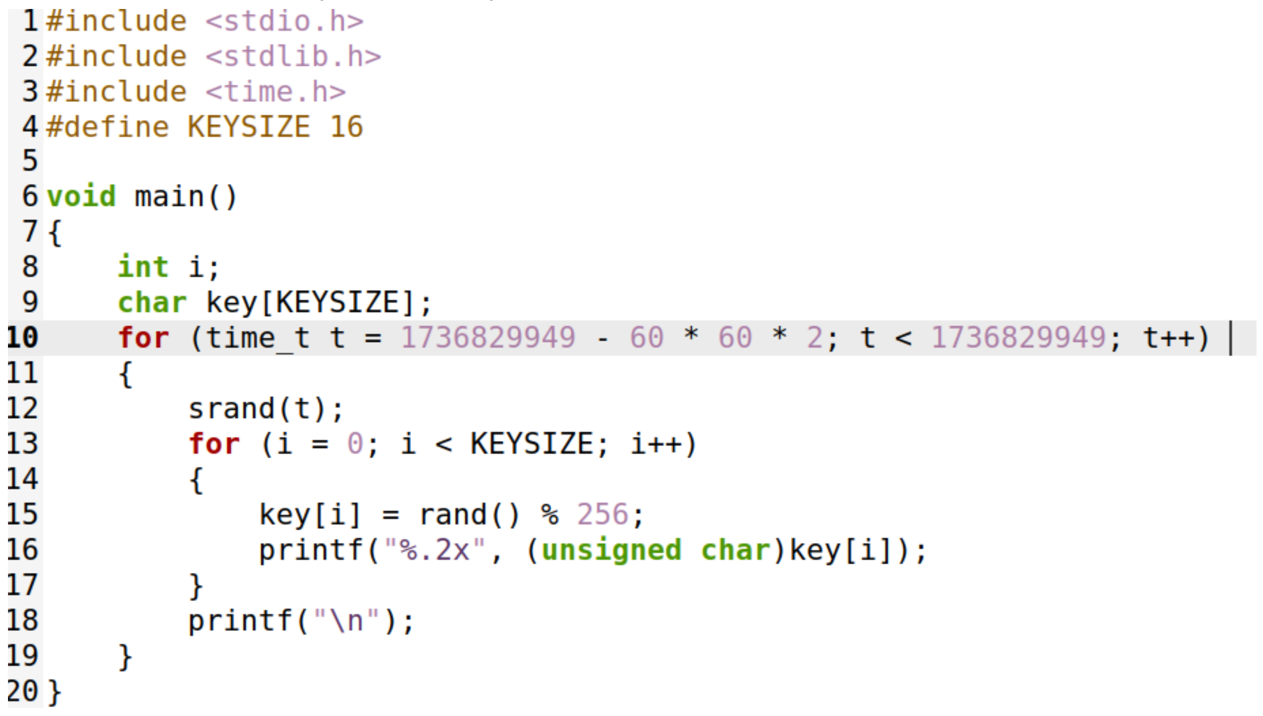
Task 2:

Run `date -d "2025-01-13 23:45:49" +%s` to find the Epoch of 2025-01-13 23:45:49
This results in 1736829949



```
[01/19/25]seed@VM:~/.../A1$ date -d "2025-01-13 23:45:49" +%s
1736829949
```

Alter task1.c to test every possible key within the two hour time frame

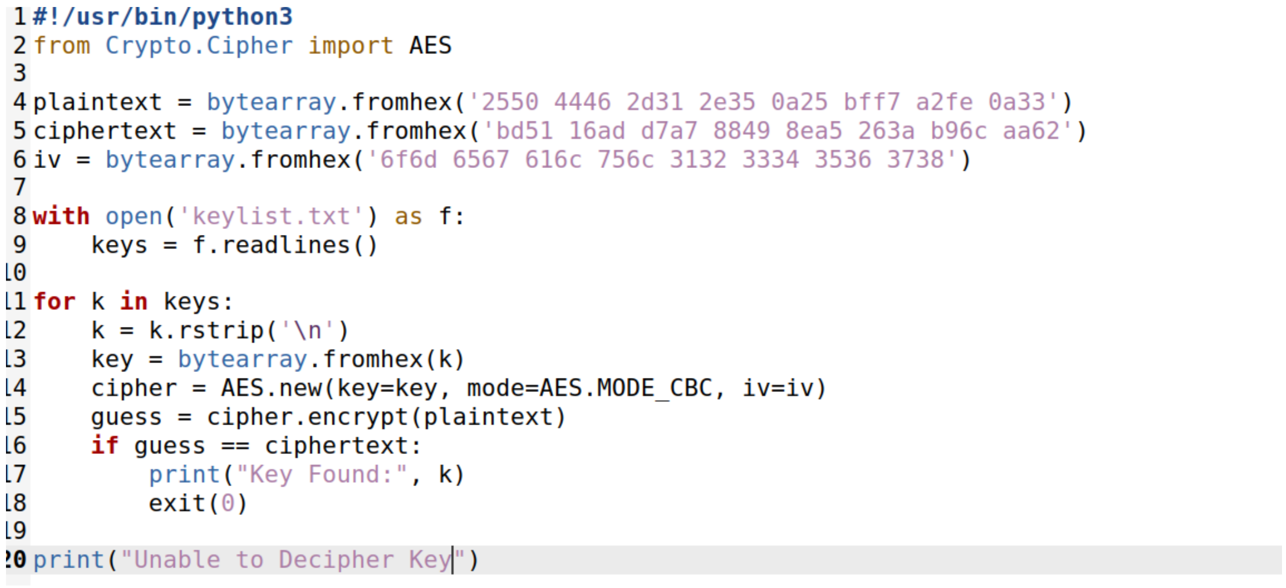


```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16

void main()
{
    int i;
    char key[KEYSIZE];
    for (time_t t = 1736829949 - 60 * 60 * 2; t < 1736829949; t++)
    {
        srand(t);
        for (i = 0; i < KEYSIZE; i++)
        {
            key[i] = rand() % 256;
            printf("%.2x", (unsigned char)key[i]);
        }
        printf("\n");
    }
}
```

Run `gcc task1.c -o task1` to compile, then run the code and feed all output into keylist.txt using `task1.c > keylist.txt`

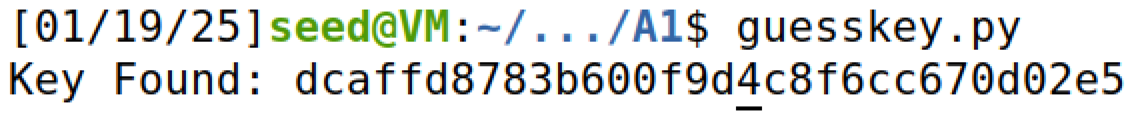
```
I wrote a Python script to brute force check all possible combinations
of keys till the correct one is found.
```

guesskey.py



```python
#!/usr/bin/python3
from Crypto.Cipher import AES

plaintext = bytearray.fromhex('2550 4446 2d31 2e35 0a25 bff7 a2fe 0a33')
ciphertext = bytearray.fromhex('bd51 16ad d7a7 8849 8ea5 263a b96c aa62')
iv = bytearray.fromhex('6f6d 6567 616c 756c 3132 3334 3536 3738')

with open('keylist.txt') as f:
    keys = f.readlines()

for k in keys:
    k = k.rstrip('\n')
    key = bytearray.fromhex(k)
    cipher = AES.new(key=key, mode=AES.MODE_CBC, iv=iv)
    guess = cipher.encrypt(plaintext)
    if guess == ciphertext:
        print("Key Found:", k)
        exit(0)

print("Unable to Decipher Key")
```

Run chmod +x guesskey.py then guesskey.py for this output:



```
[01/19/25]seed@VM:~/.../A1$ guesskey.py
Key Found: dcaffd8783b600f9d4c8f6cc670d02e5
```

Key Ideas:

The Crypto.Cipher library provides the functionality to preform AES encryption

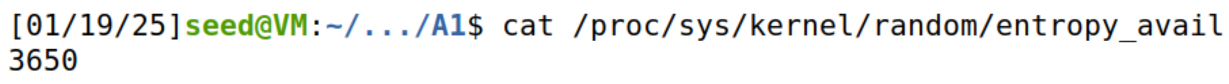
The three variables: plaintext, ciphertext, and IV are the known partial values given

- Plaintext being the original message
- Ciphertext being the encrypted message
- And IV being the initialization vector that for sufficient variability of encryption and decryption

The code iterates through all keys in keylist.txt and encrypts the plaintext with the current key being checked, and stores it in the guess variable. Then it checks if guess and the known piece of ciphertext are a match; if so, the program outputs the current key.

Task 3:

Current entropy available after running `cat /proc/sys/kernel/random/entropy_avail`



```
[01/19/25]seed@VM:~/.../A1$ cat /proc/sys/kernel/random/entropy_avail
3650
```

After running `watch -n .1 cat /proc/sys/kernel/random/entropy_avail`
When making no actions, the entropy rises very slowly, approx 1 per 1.5 seconds
Clicking the mouse, keystrokes, and highlighting seem to raise the entropy by 1 value per action: 1 key pressed, 1 mouse click, 1 space highlighted etc. Visiting a webpage seems to jump the number by about 3.
It seems visiting a webpage grants the most entropy per action, but the fastest gain comes from keyboard strokes
Taking a screenshot seemed to jump the entropy by 2 points similar to the opening of a webpage

Task 4:

Entropy increases slowly until a certain value, then drops back down

The entropy increase happens much quicker when I type and move my mouse. During which time, when the entropy drops, a new line of numbers is printed in the terminal running `cat /dev/random | hexdump`. Due to this, I suppose the command consumes entropy created from user actions to generate a random number

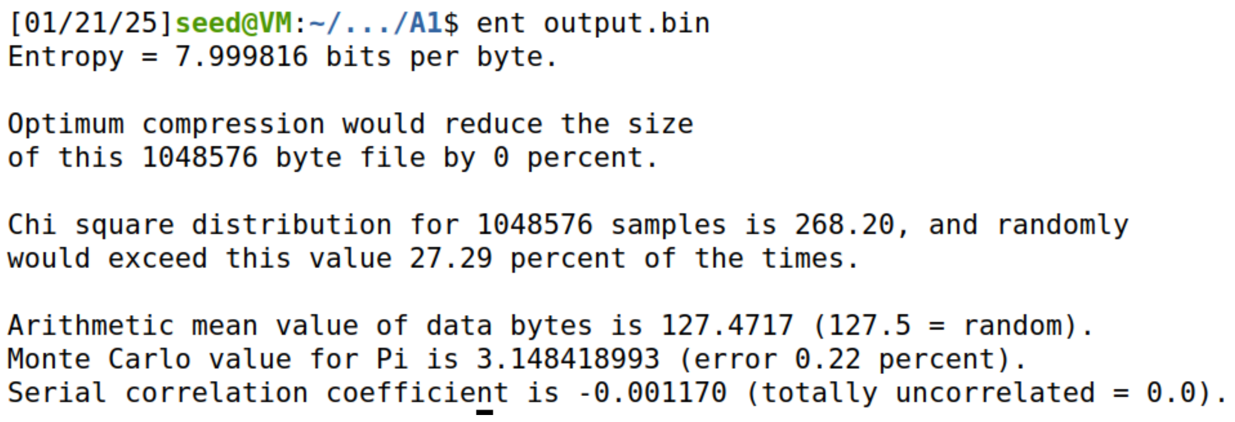
To launch a DOS attack on such a server, Attackers could keep requesting said server for establishing connections, which makes the server run out of the available entropy for `/dev/random`. Then the random number generator is blocked. It wouldn’t be able to clear the available entropy fast enough to keep accepting new connections, which would in turn, halt its random number generation process.

Task 5:

Moving the mouse while running `cat /dev/urandom | hexdump` may indirectly affect the output by adding entropy, but the effect on the PRNG is negligible. The movement of the mouse seems to add 1 extra random number per every 50 or so, it is generating too fast to record the affect of just the mouse movement.

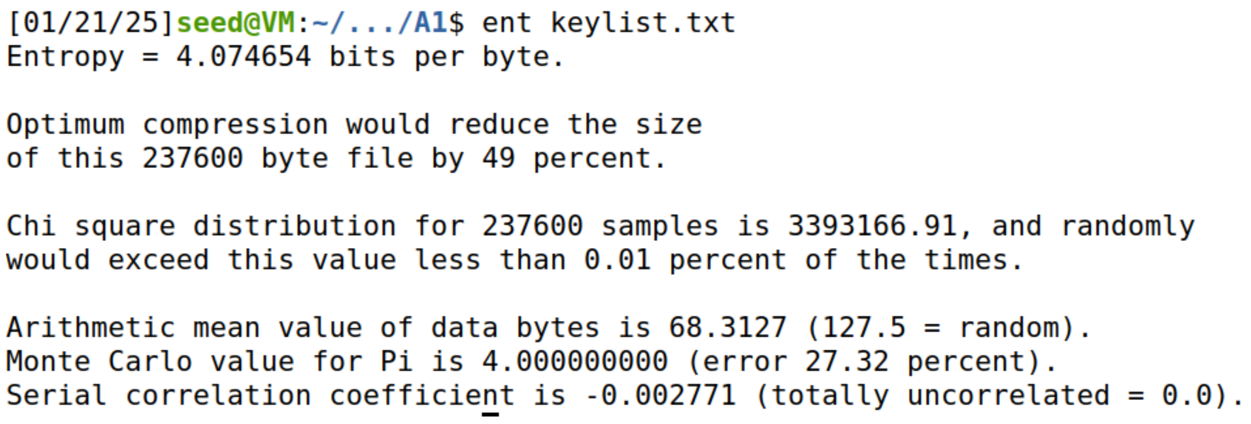
Task 6:

`ent output.bin`



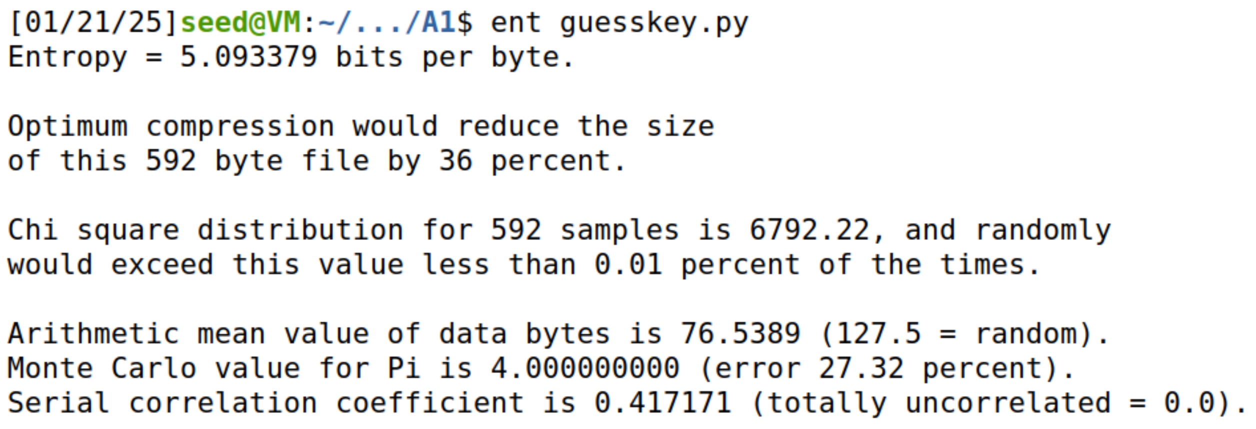
```
[01/21/25]seed@VM:~/.../A1$ ent output.bin
Entropy = 7.999816 bits per byte.

Optimum compression would reduce the size
of this 1048576 byte file by 0 percent.

Chi square distribution for 1048576 samples is 268.20, and randomly
would exceed this value 27.29 percent of the times.

Arithmetic mean value of data bytes is 127.4717 (127.5 = random).
Monte Carlo value for Pi is 3.148418993 (error 0.22 percent).
Serial correlation coefficient is -0.001170 (totally uncorrelated = 0.0).
```

`ent keylist.txt`



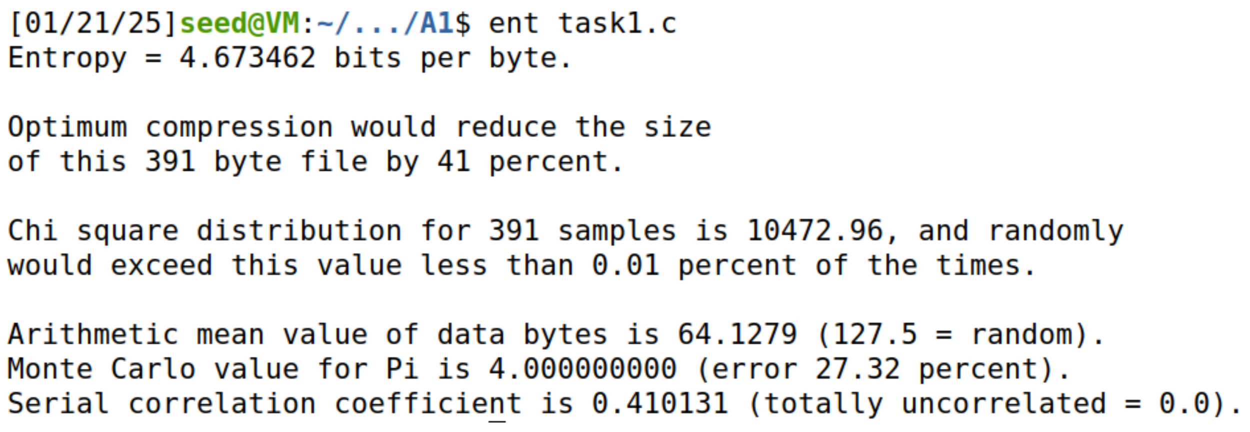
```
[01/21/25]seed@VM:~/.../A1$ ent keylist.txt
Entropy = 4.074654 bits per byte.

Optimum compression would reduce the size
of this 237600 byte file by 49 percent.

Chi square distribution for 237600 samples is 3393166.91, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 68.3127 (127.5 = random).
Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).
Serial correlation coefficient is -0.002771 (totally uncorrelated = 0.0).
```

ent guesskey.py



```
[01/21/25]seed@VM:~/.../A1$ ent guesskey.py
Entropy = 5.093379 bits per byte.

Optimum compression would reduce the size
of this 592 byte file by 36 percent.

Chi square distribution for 592 samples is 6792.22, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 76.5389 (127.5 = random).
Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).
Serial correlation coefficient is 0.417171 (totally uncorrelated = 0.0).
```

ent task1.c



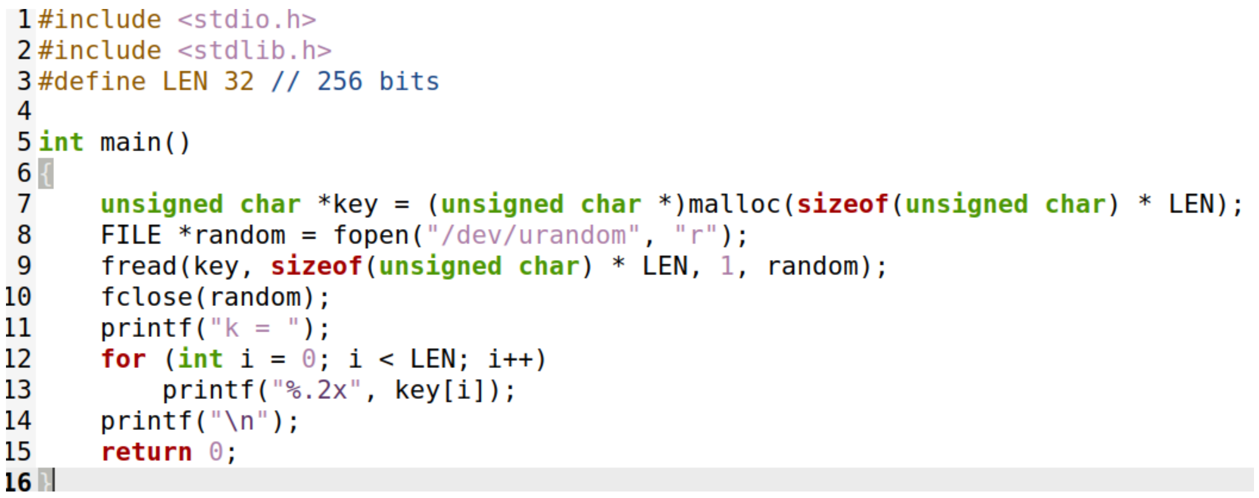
```
[01/21/25]seed@VM:~/.../A1$ ent task1.c
Entropy = 4.673462 bits per byte.

Optimum compression would reduce the size
of this 391 byte file by 41 percent.

Chi square distribution for 391 samples is 10472.96, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 64.1279 (127.5 = random).
Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).
Serial correlation coefficient is 0.410131 (totally uncorrelated = 0.0).
```

Interpretation

- Entropy - closer to 8 higher randomness
- Compression % - closer to 0 higher randomness
- Chi^2 Test - higher value = more randomness
- Arithmetic Mean - Closer to 127.5 = more randomness
- Monte Carlo Estimate of Pi - Closer to 3.141592 = more randomness
- Serial Correlation Coefficient - CLoser to 0 = more randomness

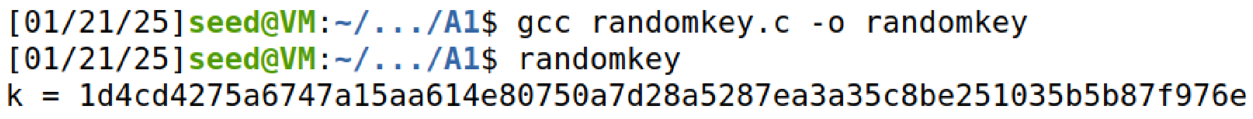
Task 7:

Randomkey.c



```c
#include <stdio.h>
#include <stdlib.h>
#define LEN 32 // 256 bits

int main()
{
    unsigned char *key = (unsigned char *)malloc(sizeof(unsigned char) * LEN);
    FILE *random = fopen("/dev/urandom", "r");
    fread(key, sizeof(unsigned char) * LEN, 1, random);
    fclose(random);
    printf("k = ");
    for (int i = 0; i < LEN; i++)
        printf("%.2x", key[i]);
    printf("\n");
    return 0;
}
```

Output



```
[01/21/25]seed@VM:~/.../A1$ gcc randomkey.c -o randomkey
[01/21/25]seed@VM:~/.../A1$ randomkey
k = 1d4cd4275a6747a15aa614e80750a7d28a5287ea3a35c8be251035b5b87f976e
```