Pseudo Random Number Generation Lab Report

Emre Can Tüzer

Date: November 4, 2024

BGK-503

Kemal Bıçakcı

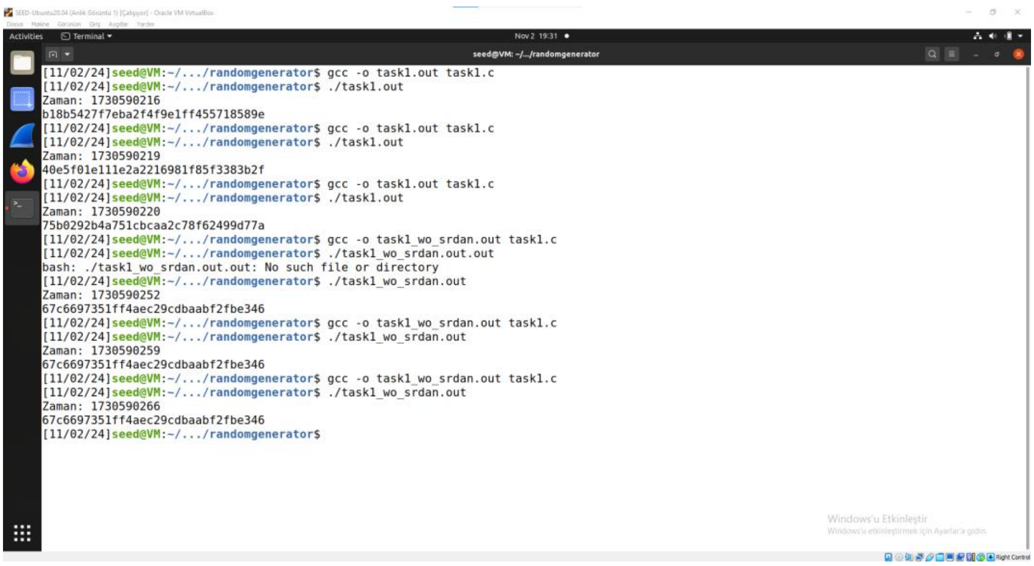
The findings of a lab research addressing security and randomness concerns in Linux operating systems are presented in this article. Secure random number generators, or PRNGs, are essential in applications involving cryptography and security where randomness is crucial. There can be significant security problems since the numbers generated by incorrect approaches can be predicted. The significance of this issue and the security evaluation of various approaches are explained in this study.

Task 1: Generate Encryption Key in a Wrong Way

We concentrate on its application as a PRNG seed. We are instructed to produce an encryption key using this method in order to illustrate that it is not safe because of its predictability. The output of the time() function is utilized to generate the encryption key in the pertinent C code, producing a predictable result.

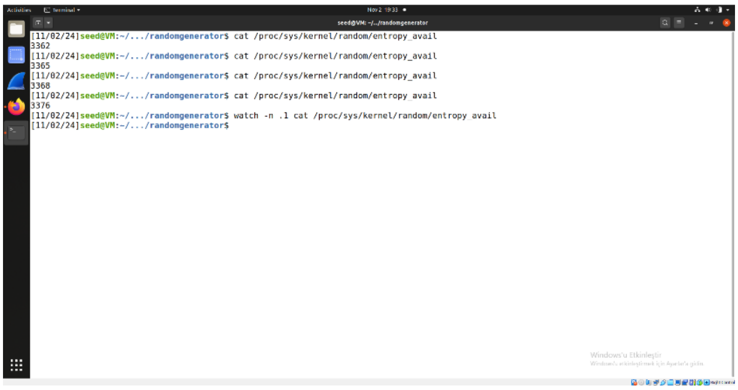
```c
#define KEYSIZE 16

int main() {
    int i;
    char key[KEYSIZE];

    printf("Zaman: %lld\n", (long long) time(NULL));
    //srand(time(NULL));

    for (i = 0; i < KEYSIZE; i++) {
        key[i] = rand() % 256;
        printf("%.2x", (unsigned char) key[i]);
    }
    printf("\n");
    return 0;
}
```

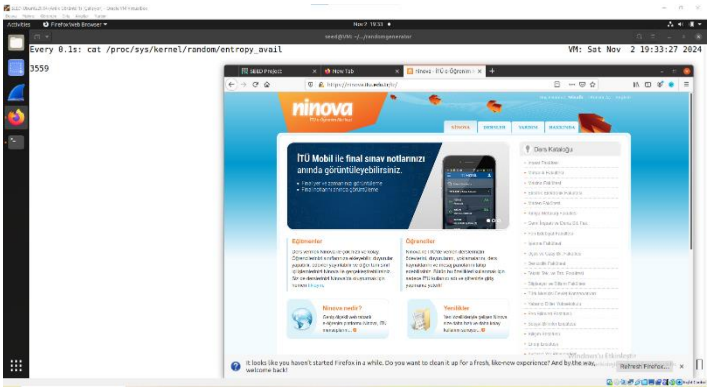
The produced randomness goes through the same loop if the srand() method is not used. Because the PRNG is provided a default seed, the same key is created once more and printed to the screen.



Task 3: Measure the Entropy of Kernel

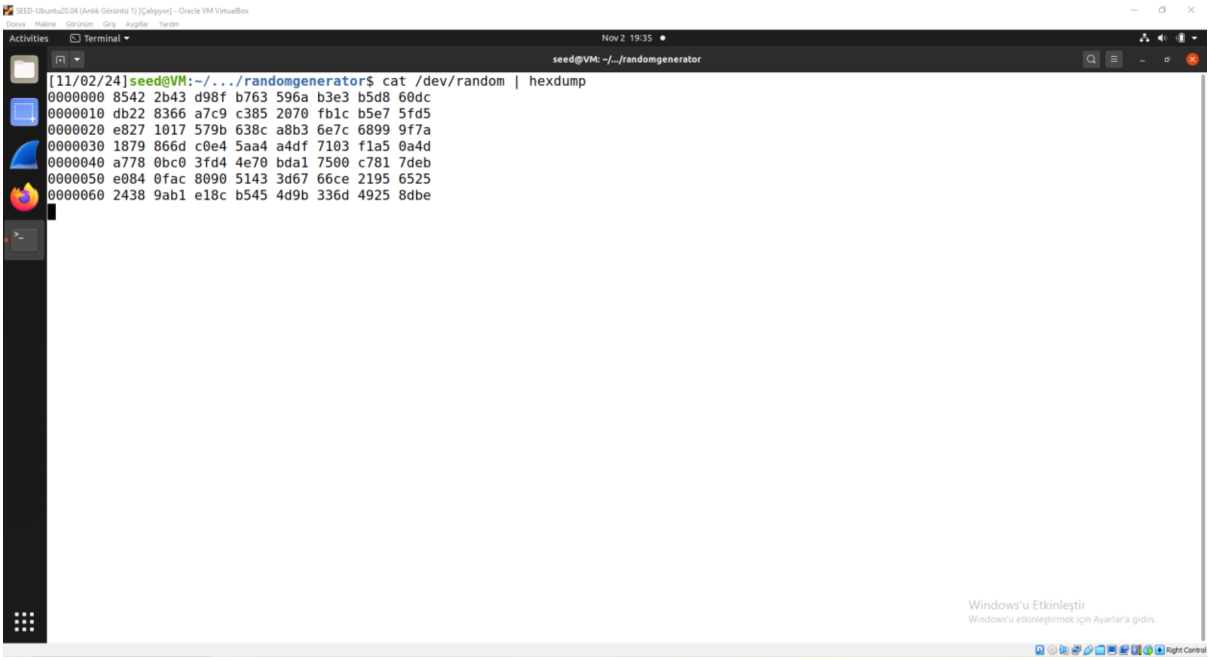
We examine how to use Linux systems' physical resources to create randomization. Linux systems generate unpredictability (entropy) through the utilization of physical resources like a keyboard, mouse, interrupts, and block devices. Entropy is increased by actions like mouse and keyboard motions and decreased while the system is idle. was shown to decline.



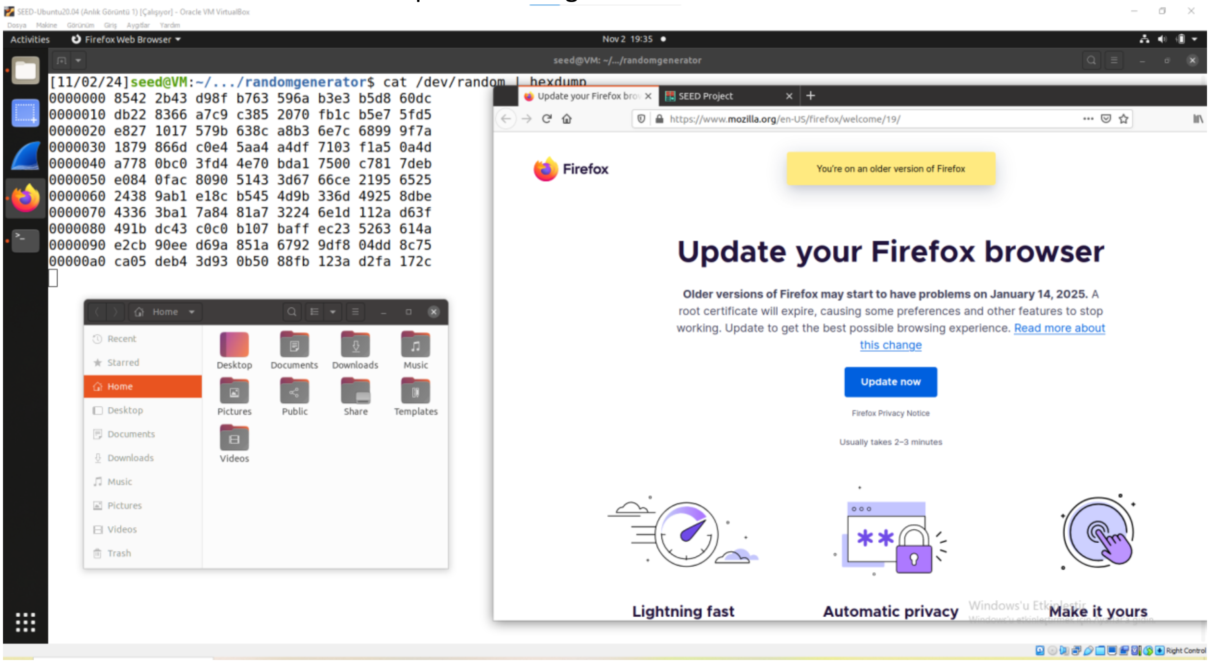


Task 4: Get Pseudo Random Numbers from /dev/random

The following command creates random numbers and uses hexdump to display them



When we launch a computer application, mouse movement, keyboard movement, etc., we can observe how the randomness component changes.

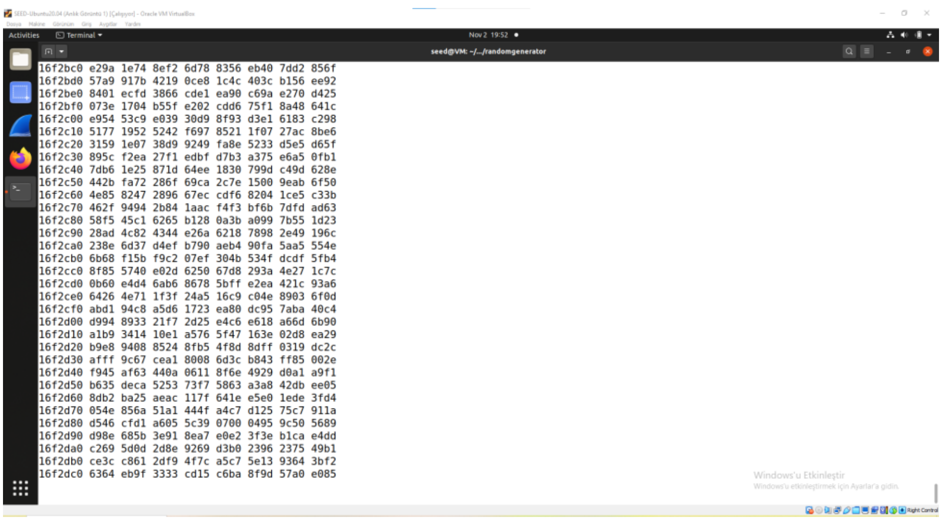


Question: If a server uses /dev/random to generate the random session key with a client. Please describe how you can launch a Denial-Of-Service (DOS) attack on such a server.

Answer: By continuously using up the entropy pool to fulfill these requests, we can stop this service from operating correctly by lowering the current entropy and blocking it. Or we can slow down entropy collection by minimizing or resetting actions such as mouse movement and keyboard use. In this way, we ensure that the entropy remains constant and we have the chance to take action to make the attack we want.

Task 5: Get Random Numbers from /dev/urandom

In contrast to /dev/random, /dev/urandom does not block and keeps producing random numbers even when entropy is absent.



Using this command, random data can be written to a file and its quality and unpredictability properties examined.

```
3da4a70 90d5 3b4f dd0c 1c6b 55cc 601d 14c8 f73d
3da4a80 1081 6a61 a0ef e0b0 9f57 7a84 c098 0849
3da4a90 38c7 8ccd b943 e9a4 cb41 1199 8051 ba78
3da4aa0 045c fef6 a585 c71d a6da 0e74 422b 1680
3da4ab0 59a2 d237 d902 b816 62dd a778 1482 ecb3
3da4ac0 fc9e c8bc 2381 376e 19e3 51d5 2cd9 6b47
3da4ad0 able 82e1 cd62 2de1 0019 5f13 e20c e924
3da4ae0 2916 8d4c 064b cf0b 7eb4 4df0 3775 82aa
3da4af0 185a 1887 5f79 934e be73 5b97 595e 9588
3da4b00 252e 614f a6a7 534c 640c b3b3 a47e ea20
3da4b10 00ec a446 8580 0ad1 85cd 8af3 c68f cd53
3da4b20 ba29 32aa 754e 6790 8907 cdef 8ddd b855
3da4b30 c9cf 0157 3ff9 23ec 9994 6016 ebd9 b96d
3da4b40 bf52 ef82 c186 a6ec e33c 60ba d36b 16ca
3da4b50 5ef6 5c0b a804 5983 9d94 2b68 34a9 a2ed
3da4b60 6fb8 a41b 30fc acca c3a4 9730 139e ec24
3da4b70 65f5 db6c a876 151a 4a80 d8b3 77cd ab67
3da4b80 968a 7964 3099 9b48 3f9f ec6b e12b d113
3da4b90 7d2d f465 151d 059b 66e1 4b8c 2d86 76c7
3da4ba0 c2d5 f453 b9f6 5dd5 626f 98b8 d5b3 f96e^C
[11/02/24]seed@VM:~/.../randomgenerator$ head -c 1M /dev/urandom > output.bin
[11/02/24]seed@VM:~/.../randomgenerator$ ent output.bin
Entropy = 7.999842 bits per byte.

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

Chi square distribution for 1048576 samples is 229.82, and randomly
would exceed this value 86.96 percent of the times.

Arithmetic mean value of data bytes is 127.5274 (127.5 = random).
Monte Carlo value for Pi is 3.139172131 (error 0.08 percent).
Serial correlation coefficient is -0.001048 (totally uncorrelated = 0.0).
[11/02/24]seed@VM:~/.../randomgenerator$
```

This application produces a 256-bit encryption key in hexadecimal format after reading it from /dev/urandom. Even with low entropy, /dev/urandom still generates random numbers. Theoretically, /dev/random is safer since it lacks entropy, even if numbers created using /dev/urandom are often regarded as secure enough.

```
int main() {
    unsigned char *key = (unsigned char *) malloc(sizeof(unsigned char) * LEN);

    FILE* random = fopen("/dev/urandom", "r");
    if (!random) {
    perror("Not allowed open  /dev/urandom");
    return EXIT_FAILURE; }

    if (!key) {
    perror("Not allowed to allocate memory ");
    return EXIT_FAILURE; }



    fread(key, sizeof(unsigned char), LEN, random);
    fclose(random);

    printf("KEY: ");
    for (int i = 0; i < LEN; i++) {
        printf("%.2x", key[i]);
    }
    printf("\n");
    free(key);
    return 0;
```

It is found that the code still generates distinct keys even if the entropy drops with each execution of the program.