МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Факультет електроніки та інформаційних технологій

Кафедра кібербезпеки

# **Курсова робота**

**з дисципліни «Алгоритми захисту інформації» на тему**

**«**Пристрої джерела випадкових чисел ядра Linux**»**

Студента(ки) групи КБ-01

Борща Дмитра Олександровича

*(прізвище, ім’я та по-батькові)*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

підпис

\_\_\_\_ . \_\_\_\_\_\_\_\_\_\_ . 2022 р.

дата здачі

**Керівник:**

ст. викладач Лаврик Т.В.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Суми - 2022

## ПЕРЕЛІК ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

1. **PRNG** - Pseudorandom Number Generator - Генератор псевдовипадкових послідовностей.
2. **TRNG** - True Random Number Generator - Генератор істинно випадкових послідовностей.
3. **CSPRNG** - Cryptographically-secure Pseudorandom Number Generator.
4. **RNG** - Random Number Generator - Генератор випадкових послідовностей.
5. **TPM** - Trusted Platform Module.
6. **LFSR** - Linear Feedback Shift Register - Регістр зсуву з лінійним зворотним зв'язком; поширений спосіб отримання псевдовипадкових послідовностей.

## **ЗМІСТ**

[**ПЕРЕЛІК ПОЗНАЧЕНЬ І СКОРОЧЕНЬ**](#_lelkydkrkttt) **2**

[**ВСТУП**](#_upte6zjnim9) **4**

[**1.ЗАГАЛЬНИЙ ОГЛЯД ГЕНЕРАТОРІВ ВИПАДКОВИХ ЧИСЕЛ**](#_teac6rwrgxz8) **5**

[**2.ГЕНЕРАТОРИ ВИПАДКОВИХ ЧИСЕЛ В LINUX**](#_d6x3w0k60o) **7**

[2.1 Загальна структура генерації рандомних чисел](#_qvhi4ovhi2nf) 7

[2.2 Як влаштований CSPRNG в ядрі Linux 5.4](#_5e4gs870e4va) 11

[**3.ВИКОРИСТАННЯ /DEV/RANDOM ТА /DEV/URANDOM**](#_b3rc9qvz9ycf) **14**

[3.1 Програми, що використовують /dev/random та /dev/urandom](#_979msav2atjb) 14

[3.2 Доступ до системного генератору випадкових чисел власноруч](#_10c2mwwfbf49) 15

[**ВИСНОВОК**](#_lzyuwtm3sx01) **20**

[**ВИКОРИСТАНІ ДЖЕРЕЛА**](#_3vor3n9kaqgo) **21**

## **ВСТУП**

Проблема знаходження істинно випадкових послідовностей чисел в обчислювальних машинах була і залишається активним топіком для обговорень та досліджень. ЕОМ(електронно-обчислювальна машина) це дискретний пристрій. Більшість ЕОМ працюють в двійковій системі числення та за законами алгебри логіки. Як відомо, будь-яка операція зі сталими змінними в математиці завжди дає однаковий результат. Комп’ютер, на відміну від людини, не має уяви та не може “придумати” значення для змінної просто так. З часом, під час розвитку ЕОМ, гострою стала проблема генерації випадкових послідовностей даних, як для рутинного використання, так і в потребностях криптографії. Остання дуже потребує “якісних” випадкових чисел. Так з’явилась потреба в **генераторах випадкових чисел**. На даний момент найбільш популярні операційні системи мають власні системні генератори випадкових чисел, які можуть використовуватись як для потреб власне системи, так і може даватись доступ до цих генераторів стороннім процесам. Окрім програмних генераторів випадкових чисел, багато комп’ютерів зараз мають фізичні TPM(trusted platform module) модулі, що являють собой окремий пристрій, який містить в собі генератор випадкових чисел.

Метою даної курсової роботи буде дослідження системних генераторів випадкових чисел в ядрі Linux та їх використання на практиці.

## 1.**ЗАГАЛЬНИЙ ОГЛЯД** **ГЕНЕРАТОРІВ ВИПАДКОВИХ ЧИСЕЛ**

Загалом всі генератори можна поділити на Pseudorandom Number Generator(PRNG, генератор псевдовипадкових чисел) та True Random Number Generator(TRNG, генератор істинно рандомних чисел). Вони в корені відрізняються принципом генерації випадкових послідовностей.

Принципи генерації також прийнято ділити на дві категорії. Перший принцип базується на використанні якогось фізичного явища, яке прийнято вважати непередбачуваним. Наприклад, електромагнітний шум, що надходить з відкритого космосу, квантові явища, чи рух речовини в лавових лампах, які наприклад використовує CloudFlare для власної генерації випадкових чисел, що використовуються в криптографії(рис. 1). Дані, що можна зібрати з цих явищ використовуються в якості ентропії для подальшого перетворення її в число, яке можна вважати істинно рандомним. Другий же принцип базується виключно на розрахункових алгоритмах, що можуть виробляти достатньо довгі ланцюги випадкових чисел. Такому методу потрібне лише початкове значення, яке також називається ключем, або сідом(seed). Ключ може братися з ентропії генератору істинно випадкових чисел.



Рисунок 1 - Лампи з проекту Lavarand.

Таким чином, генератори істинно випадкових чисел дають “якісні” випадкові числа за наявності “якісної” ентропії. В той час як генератори псевдовипадкових чисел здатні генерувати в тисячі разів довші послідовності, використовуючи короткий початковий ключ.

Частим об’єктом суперечок серед фахівців є “якість” чисел, що генеруються за допомогою псевдовипадкових генераторів. Багато хто стверджує, що ланцюги даних не є криптостійкими та можуть бути скомпроментовані. Перш за все, небезпечним псевдовипадковий генератор є через те, що використовуючи один сід, рано чи пізно послідовність чисел почне повторюватися і такий ланцюг не можна буде вважати криптостійким. Таким чином зловмисник може знайти початковий сід та передбачити генерацію чисел в майбутньому.

То ж варто розібратися, чи дійсно системний PRNG в Linux є не криптостійким та чи може поставити ваш проект під загрозу через “неякісні” випадкові числа.

## 2.ГЕНЕРАТОРИ ВИПАДКОВИХ ЧИСЕЛ В LINUX

### 2.1 Загальна структура генерації рандомних чисел

Предметом дослідження є генератори в ядрі Linux. Прийнято вважати, що в ньому реалізовано обидва типи генераторів (/dev/random та /dev/urandom). За усталеною думкою /dev/urandom - псевдовипадкові та некриптостійкі числа, а /dev/random - істинно рандомні числа. Але правда є дещо іншою. Насправді вони обидва є криптографічно стійкими генераторами псевдовипадкових чисел(CSPRNG), що є різновидом генераторів псевдовипадкових послідовностей. На цьому етапі може виникнути дисонанс, чого тоді /dev/random - безпечний, а /dev/urandom - ні. Насправді достатньо розібратися в справжній відмінності цих генераторів та їх структурі в системі.

Серед користувачів існувало застаріле уявлення про ці генератори. Воно продемонстровано на рис. 2 та мало сенс до ядра версії 5.6. За цим уявленням система отримує “істинно” рандомну ентропію з переліку джерел(шум мікрофону, натискання клавіш, рухи миші). Після додавання “білого шуму” система додає кількість ентропії до лічильника, та за необхідністю цю “істинно” випадкову послідовність можна отримати в /dev/random та /dev/urandom. Так це повинно працювати за теорією, що /dev/random це TRNG. /dev/urandom також отримує “якісну” ентропію, за її наявності, але при її закінченні /dev/random блокується, а /dev/urandom починає отримувати “неякісні” рандомні числа від CSPRNG.

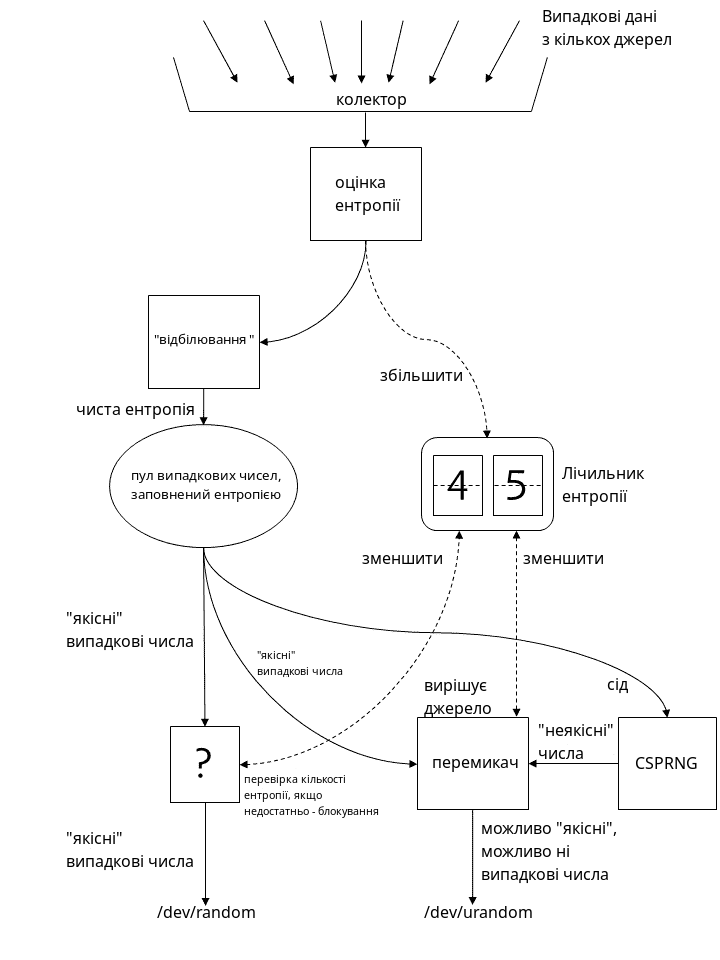


Рисунок 2 - Застаріле уявлення про генерацію рандомних чисел в Linux[4].

Насправді це уявлення є невірним в версіях ядра 5.6>. Справжня структура генераторів зображена на рис. 3. Насправді ж початкова “відбілена” ентропія використовується CSPRNG та цей пул доступний в /dev/random та /dev/urandom. CSPRNG ніколи не блокується, навіть коли в системі зовсім немає ентропії, він буде генерувати досить “неякісні”, але псевдовипадкові числа. Просто в цьому випадку /dev/random блокується, а /dev/urandom не блокується ніколи.

Тож /dev/random та /dev/urandom будуть брати дані з одного рандомного пулу, до того моменту, поки “свіжа” ентропія на вході CSPRNG не закінчиться. Після цього “якість” ланцюгу рандомних даних знижується.

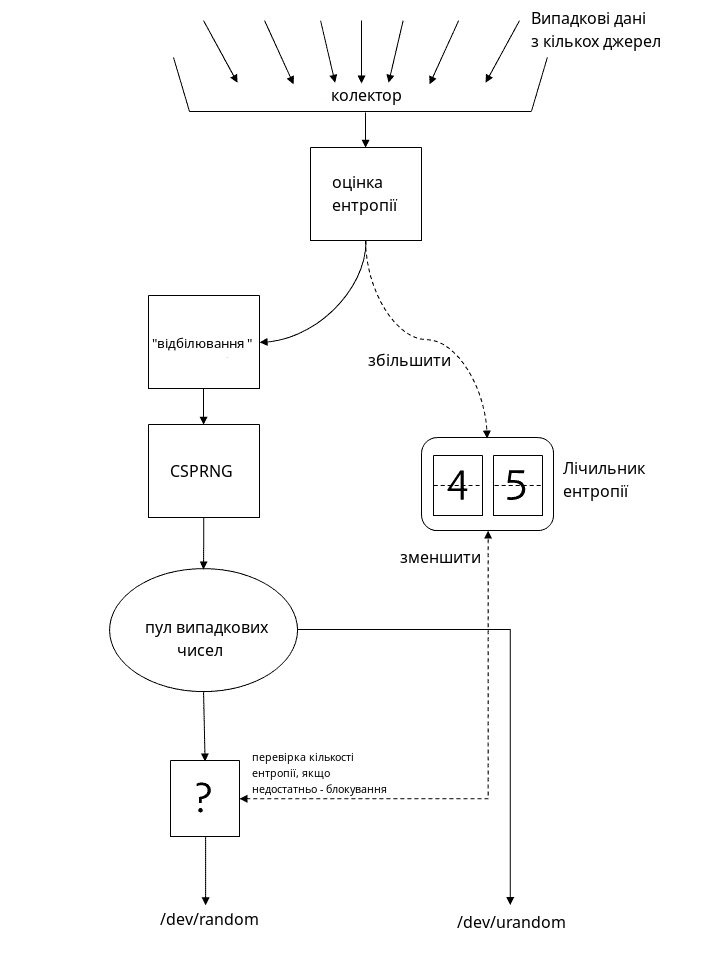


Рисунок 3 - Справжня структура генераторів випадкових чисел в Linux[4].

Насправді варто зазначити, що “свіжа” ентропія для CSPRNG потрібна лише для оновлення внутрішнього стану (re-seeding). Таким чином, навіть якщо ентропія в системі закінчилась, але зловмисник не знає внутрішнього стану CSPRNG, то ланцюг чисел, що він генерує можна вважати досить криптостійким. Експерти вважають, що якщо генератор на початку роботи отримав достатньо “якісної” ентропії(за деякими джерелами не більше 256 байт), то ланцюг чисел, який він буде генерувати, не можна відрізнити від істинно випадкових, за допомогою існуючих технологій.

Thomas Pornin(Криптограф, один з найпопулярніших користувачів на stackexchange) писав у відповідь на запитання про те “чи можна використовувати /dev/urandom для генерації випадкової строки довжиною 1024 біти”:

|  |
| --- |
| *Коротка відповідь - так. Довга відповідь також так. /dev/urandom дає дані, які неможливо відрізнити від справжньої випадковості, враховуючи існуючу технологію. Отримувати «кращу» випадковість, ніж те, що надає /dev/urandom, не має сенсу, якщо ви не використовуєте один з небагатьох «теоретико-інформаційних» криптографічних алгоритмів, що не у вашому випадку (ви це знаєте).* |

### 2.2 Як влаштований CSPRNG в ядрі Linux 5.4

Одним з основних понять в CSPRNG в Linux є пу́ли(pools). Пулом називається певна вибірка даних фіксованого розміру. Всього CSPRNG оперує трьома пулами: input\_pool, blocking\_pool, crng\_state[5]. Детальніше про кожен з них далі.

Як уже з’ясовано, під час роботи операційної системи збирається рандомна ентропія з різних джерел(наприклад час доступу до жорсткого диску, деякі мережеві дані, рухи миші, натискання клавіш клавіатури). Ці дані формуються в пул розміром 4096 біт, який називається input\_pool. Оскільки криптостійкість цих даних не дуже важлива, то до них використовується швидкий алгоритм LFSR (linear-feedback shift register), який в ядрі реалізований функцією \_mix\_pool\_bytes(). Також, одним з джерел вхідного пулу є interrupts(преривання). Це таймштампи, адреси функцій в пам’яті, та ін. Загалом дані з преривань збираються в декілька fast\_pool(по 128 біт) за допомогою функції fast\_mix(), що використовує XORing з теперішнім станом пулу. Тобто функція fast\_mix() збирає дані з переривань та побітово проводить операцію XOR з минулим станом пулу fast\_pool. Варто зазначити, що таких fast\_pool-ів існує декілька, а саме по одному на кожне ядро процесору. fast\_pool-и додаються в якості ентропії в input\_pool. Таке “підмішування” проводиться щосекунди. Сформований input\_pool є лише джерелом ентропії для CSPRNG і доступ до нього користувач отримати не може.

Витяг ентропії з пулу завжди виконується за одним і тим же алгоритмом, за допомогою функції extract\_buf() (за винятком fast\_pool, які більше схожі на локальні буфери, що використовуються у функції). Обчислюється хеш SHA-1 даних пулу, тобто 160 біт, які негайно змішуються назад у пул, щоб уникнути атак зворотного відстеження. Потім цей хеш «згортається» назад на себе (XOR з 80 MSB на 80 LSB), і виводиться лише цей половинний хеш. Коли ЦП надає інструкцію RDSEED, початковий стан алгоритму SHA-1 ініціалізується з нею, таким чином змішуючи апаратно згенеровані випадкові дані в пул і витягнуті дані. Таким чином, вилучення ентропії здійснюється блоками розміром не більше 80 біт. Більше того, кожна екстракція перемішує внутрішній стан пулу. Загалом, цей алгоритм досить повільний, оскільки використовує SHA-1. Використання криптографічного хешу забезпечує рівномірний розподіл вихідних даних (зчитування), які не відрізняються від випадкових даних, що дозволяє без ризику використовувати функцію швидкого змішування при введенні (записі) в пул.

Другим пулом є blocking\_pool. Він має ту ж структуру, що і input\_pool, тільки зі зменшеним розміром в 1024 біти. Під час читання з цього пулу він передає стільки ентропії, скільки потрібно, з input\_pool, використовуючи алгоритм вилучення, описаний вище. Кожен байт, прочитаний таким чином, змішується один за іншим у blocking\_pool до тих пір, поки не буде змішано необхідну кількість байтів (не більше, не менше). Вимірювання ентропії input\_pool зменшується від кількості прочитаних байтів, а ентропія blocking\_pool збільшується однаково. Потім дані витягуються з blocking\_pool з використанням того ж алгоритму вилучення, а їх вимірювання ентропії зменшується на суму видобутого. Ці витягнуті дані можуть бути прочитані користувачем або шляхом читання спеціального файлу пристрою /dev/random або шляхом виклику системного виклику getrandom().

Третій ентропійний буфер — це внутрішній стан CSPRNG, crng\_state, розміром 512 біт. Вимірювання ентропії його внутрішнього стану не проводиться. Його ініціалізація виконується шляхом вилучення даних з input\_pool, знову ж таки, дотримуючись алгоритму вилучення, описаного вище. Заради ефективності, після ініціалізації цього генератора вилучення даних із input\_pool виконується щонайбільше кожні 5 хвилин (якщо дані не потрібні, вони не витягуються). Зчитування даних із crng\_state здійснюється за допомогою потокового шифру ChaCha20, який швидше, ніж використання функції хешування, як це робиться в алгоритмі вилучення ентропії. Зчитування результатів шифру можна виконати або читанням спеціального файлу пристрою /dev/urandom (зверніть увагу на «u», що означає «розблокування»), або викликом системного виклику getrandom() без параметрів. Читання з цього /dev/urandom ніколи не блокується, незалежно від внутрішньої ентропії crng\_state. Однак виклик getrandom() блокується, доки crng\_state не буде ініціалізовано шляхом вилучення 128 біт з input\_pool (за допомогою функції crng\_reseed()) або поки він не буде ініціалізований 512 бітами (функцією crng\_fast\_load()) з переривань або апаратного RNG процесора(за наявності). Немає ризику «вичерпання» ентропії за допомогою /dev/urandom або getrandom(). Дельта часу в 5 хвилин достатньо коротка, щоб повторно заповнити стан перед повторенням шифру потоку, навіть у найгіршому випадку.

На момент написання роботи генератор випадкових чисел в Linux переживає значні зміни[1]. Наприклад, як описано вище з версії 5.6 /dev/random вже не використовує blocking\_pool, а отримує дані з CSPRNG як і /dev/urandom, а з версії 5.17(20 березня 2022) алгоритм SHA-1, який використовувався для екстракції з input\_pool, був замінений на BLAKE2, що є послідовником ChaCha20. Також тепер ядро може визначати, що працює на скопійованій віртуальній машині, що дуже важливо, адже дочірній екземпляр може мати ідентичний батьківському пул ентропії, що означає, що їх генератори будуть повертати однаковий результат. Копіювання віртуальних машин є досить частим процесом на віртуальних хостингах, отже це нововведення є досить важливим в покращенні безпеки.

Тому на даний момент точно описати структуру генератора рандомних чисел в Linux[3] досить складно, враховуючи відсутність повної актуальної документації. Досить вірогідно, що в найближчих версіях станеться повна його трансформацію.

## 3.ВИКОРИСТАННЯ /DEV/RANDOM ТА /DEV/URANDOM

### 3.1 Програми, що використовують /dev/random та /dev/urandom

Насправді, так багато чуток та суперечок на рахунок генерації випадкових чисел утворилося не на пустому місці. Генерація “якісних” рандомних ланцюгів чисел є чи не найважливішою складовою криптографії.

Одначе, використання генераторів випадкових послідовностей може бути зовсім різним. Наприклад, заповнення дискового простору певними випадковими даними, де зовсім не важлива “якість” ентропії, або ж генерація ключів PGP, SSH(рис. 4), SSl, тощо. Так, використовуючи /dev/urandom можемо заповнювати терабайти на диску випадковими даними, щоб стерти інформацію, що зберігалась на ньому. Саме /dev/urandom буде зручним у цьому випадку, бо він ніколи не досягне блокування.

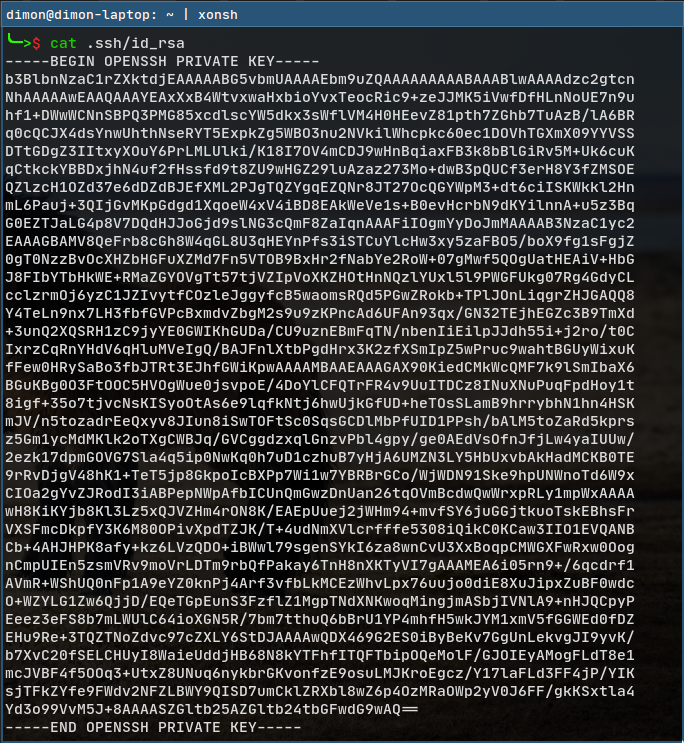


Рисунок 4 - Приклад згенерованого SSH ключу.

### 

### 3.2 Доступ до системного генератору випадкових чисел власноруч

Існує два основні методи доступу до вихідного пулу[6]: за допомогою файлу псевдо пристрою та за допомогою системного виклику[2]. Перший варіант є більш універсальний. Його можна використовувати майже в будь-якій мові програмування. До того-ж доступ до нього може отримати будь-який користувач. На відміну від файлу, до системного виклику не вдається отримати доступ зі скриптових мов програмування (напр. Python або PHP).

З /dev/{u}random можемо працювати як зі звичайним файлом та зчитувати необхідну кількість байт. Нижче подано приклад зчитування строки довжиною 50 символів, що містить лише літери англійського алфавіту.

|  |
| --- |
| #include <iostream> #include <fstream> #include <cctype>  const unsigned int data\_size = 50;  int main() {   FILE\* randomData = fopen("/dev/random", "r");  char myRandomData[data\_size];  char temp\_data = ' ';   if (randomData == NULL) {  std::cout << "Something went wrong!!!" << std::endl;  }  else {  for (int i = 0; i < data\_size; i++) {  retry:  ssize\_t result = fread(&temp\_data, 1, 1, randomData);  if (result < 0) {  std::cout << "Something went wrong!!!" << std::endl;  goto retry;  }  else if (!isalpha(temp\_data)) goto retry;  myRandomData[i] = temp\_data;  }  fclose(randomData);  std::cout << "Collected data: " << myRandomData << std::endl;  }  return 0; } |

Принцип роботи програми досить простий. Спочатку відкривається файловий дескриптор для файлу /dev/random. Одразу перевіряється, чи відкрився файл, якщо покажчик дорівнює NULL, значить не вдалося відкрити /dev/random і програма завершується. Якщо файл відкритий успішно та покажчик набув значення, то починається цикл заповнення масиву char-ів. Як відомо з курсу програмування, char займає 1 байт данних. Для того, щоб заповнити масив літерами алфавіту зчитуються дані з /dev/random по одному байту та перевіряються за допомогою функції isalpha() кожен зчитаний байт. Якщо функція повертає true, тоді записується літеру в масив, інакше повторюємо зчитування. Коли масив заповнений, то закривається файловий дескриптор та виводиться зчитаний масив літер. Приклад роботи програми на рис. 5.

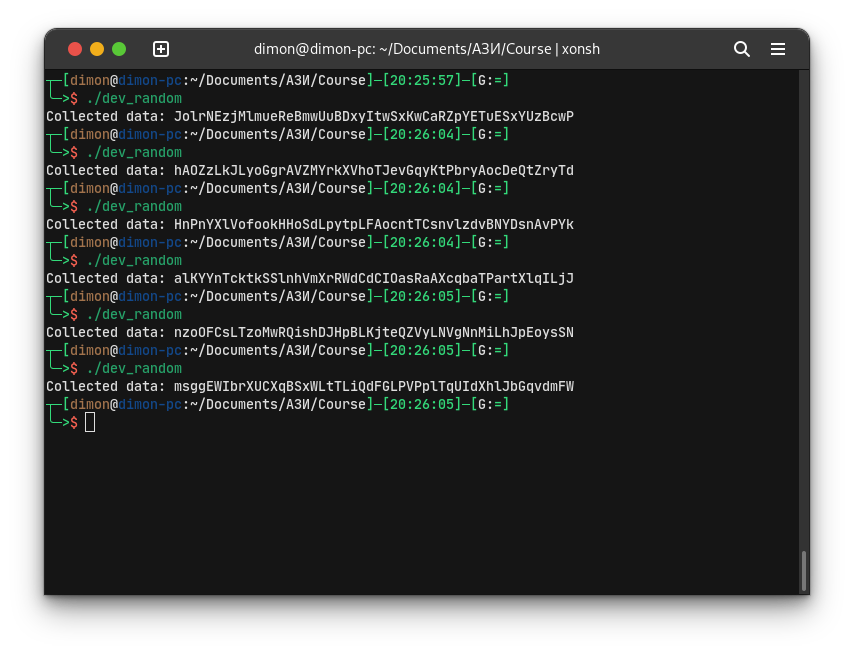


Рисунок 5 - Приклад роботи програми, що зчитує дані з /dev/random.

Для спрощення життя програмістам, розробники ядра Linux створили системний виклик getrandom(), що спрощує доступ до системного генератору рандомних чисел. Згідно документації, getrandom() має наступний синтаксис в мові C:

**ssize\_t getrandom(void \****buf***, size\_t** *buflen***, unsigned int** *flags***);**

* **ssize\_t**: результат, який повертає функція (-1 в разі помилки);
* **void** *\*buf*: буфер, в який буде записано дані;
* **size\_t** *buflen*: розмір буфера в байтах;
* **unsigned int** *flags*: прапори функцій.

Доступні прапори функції:

* **GRND\_RANDOM:**

Якщо цей біт встановлено, то випадкові байти витягуються з *random* (тобто те саме, що й */dev/random*) замість *urandom*. Якщо кількість доступних байтів у *random* менше, ніж запитується в *buflen*, то виклик повертає лише доступні випадкові байти. Якщо немає доступних випадкових байтів, поведінка залежить від наявність **GRND\_NONBLOCK** в аргументі *flags*.

* **GRND\_NONBLOCK:**

За замовчуванням, під час читання з random, getrandom() блокується, якщо немає доступних рандомних байтів, і під час зчитування з urandom він блокує, якщо пул ентропії ще не ініціалізовано. Якщо встановлено прапор GRND\_NONBLOCK, тоді getrandom() не блокується у цих випадках, але замість цього негайно повертає -1.

Отже створено просту програму, яка виконує ту ж задачу, що й в попередньому прикладі з генерацією строки довжиною 50 літер англійського алфавіту. Нижче подано код даної програми, що використовує системний виклик getrandom().

|  |
| --- |
| #include <iostream> #include <cctype> #include <sys/random.h>  const unsigned int data\_size = 50;  int main() {   char myRandomData[data\_size];  char temp\_data = ' ';   for (int i = 0; i < data\_size; i++) {  retry:  ssize\_t result = getrandom(&temp\_data, 1, GRND\_RANDOM);  if (result < 0) {  std::cout << "Something went wrong!!!" << std::endl;  return 0;  }  else if (!isalpha(temp\_data)) goto retry;  myRandomData[i] = temp\_data;  }   std::cout << "Collected data: " << myRandomData << std::endl;   return 0;  } |

Програма працює за аналогічним принципом, але звертається до генератору рандомних чисел за допомогою системного виклику getrandom(). На рис. 6 подано приклад роботи програми.

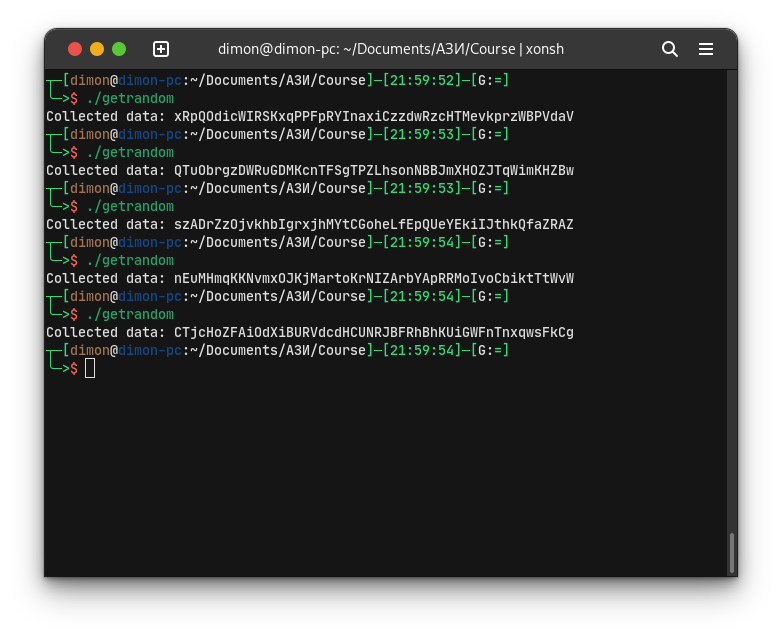


Рисунок 6 - Приклад роботи програми, що використовує системний виклик getrandom().

## ВИСНОВОК

Використання операційної системи Linux у хмарних технологіях набуває все більшої популярності з кожним роком. Для захисту інформації досить важливо генерувати надійні ключі, джерелом яких повинен бути надійний генератор рандомних чисел.

В процесі виконання курсової роботи було проведено обширне дослідження системного генератору випадкових чисел в ядрі Linux. Досліджено його архітектуру та джерела даних для файлів псевдо пристроїв /dev/random та /dev/urandom. Linux постійно розвивається та оновлюється. Так системний CSPRNG зазнав досить значних змін протягом свого існування, та ймовірно ще більші зміни чекають його в майбутньому, щоб забезпечити надійну генерації “якісних” рандомних даних, задля підвищення криптостійкосі та безпеки інформації. Це те, що оточує нас в повсякденному житті: SSL сертифікати, SSH, PGP ключі. З великою долею вірогідності, більшість з них були згенеровані з /dev/random.

## ВИКОРИСТАНІ ДЖЕРЕЛА

1. *BSI - Bundesamt für Sicherheit in der Informationstechnik*. URL:<https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/LinuxRNG/LinuxRNG_EN.pdf?__blob=publicationFile> (date of access: 28.05.2022).
2. getrandom(2) - Linux manual page. *Michael Kerrisk - man7.org*. URL:<https://man7.org/linux/man-pages/man2/getrandom.2.html> (date of access: 28.05.2022).
3. GitHub - torvalds/linux: Linux kernel source tree. *GitHub*. URL:<https://github.com/torvalds/linux> (date of access: 28.05.2022).
4. Hühn T. Myths about /dev/urandom. *Thomas' Digital Garden*. URL:<https://www.2uo.de/myths-about-urandom/> (date of access: 28.05.2022).
5. Linux RNG architecture. *AMOSSYS – Expertise & Innovation en Cybersécurité*. URL:<https://www.amossys.fr/fr/ressources/blog-technique/linux-csprng-architecture/> (date of access: 28.05.2022).
6. random(4) - Linux manual page. *Michael Kerrisk - man7.org*. URL:<https://man7.org/linux/man-pages/man4/random.4.html> (date of access: 28.05.2022).