**Звіт**

**До практичної роботи**

**з дисципліни «Основи інформаційної безпеки»**

Студента Київського Національного Університету ім. Тараса Шевченко

Переверзева Олексія Сергійовича

Факультет інформаційних технологій

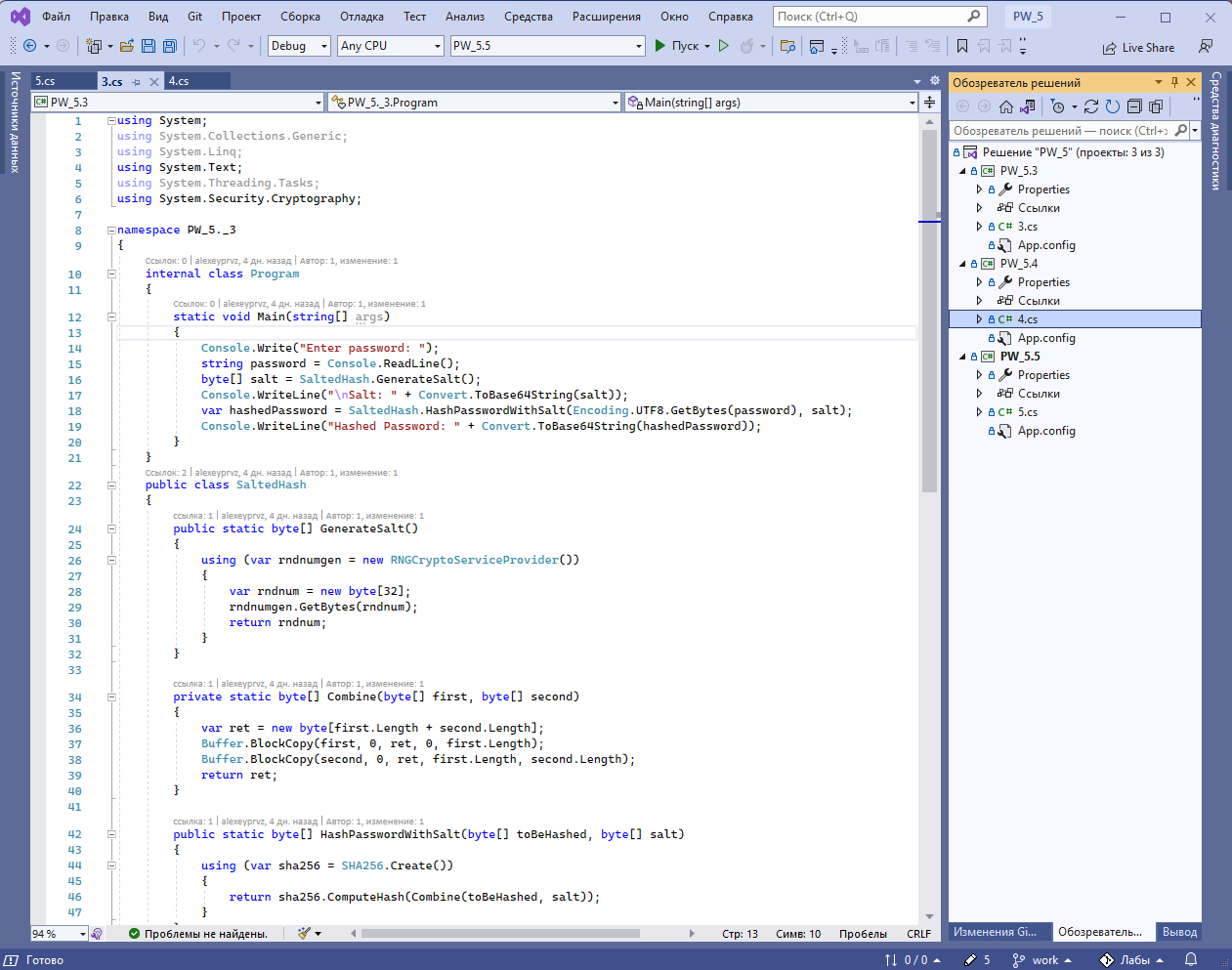
Група МІТ-21



**Практичне заняття №5**

**Тема:** Безпечне зберігання паролів

1. Розробити клас SaltedHash, що реалізує хешування паролів із додаванням додаткової ентропії.



Перший метод, використовуючи об’єкт RNGCryptoServiceProvider, створює додаткову ентропію, тобто сіль, для хешування:

public static byte[] GenerateSalt()

{

using (var rndnumgen = new RNGCryptoServiceProvider())

{

var rndnum = new byte[32];

rndnumgen.GetBytes(rndnum);

return rndnum;

}

}

Наступний метод в цьому класі називається Combine. Мета цього методу - об'єднати два байтових масиву в однобайтовий масив, який використовується для додавання нашої попередньо згенерованої солі до пароля, який ми хочемо хешувати:

private static byte[] Combine(byte[] first, byte[] second)

{

var ret = new byte[first.Length + second.Length];

Buffer.BlockCopy(first, 0, ret, 0, first.Length);

Buffer.BlockCopy(second, 0, ret, first.Length, second.Length);

return ret;

}

Ну і третій метод, викликаючи другий метод Combine, обчислює хеш функцію за алгоритмом SHA256 та повертає її:

public static byte[] HashPasswordWithSalt(byte[] toBeHashed, byte[] salt)

{

using (var sha256 = SHA256.Create())

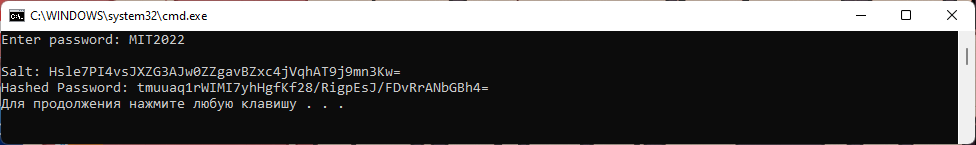
{

return sha256.ComputeHash(Combine(toBeHashed, salt));

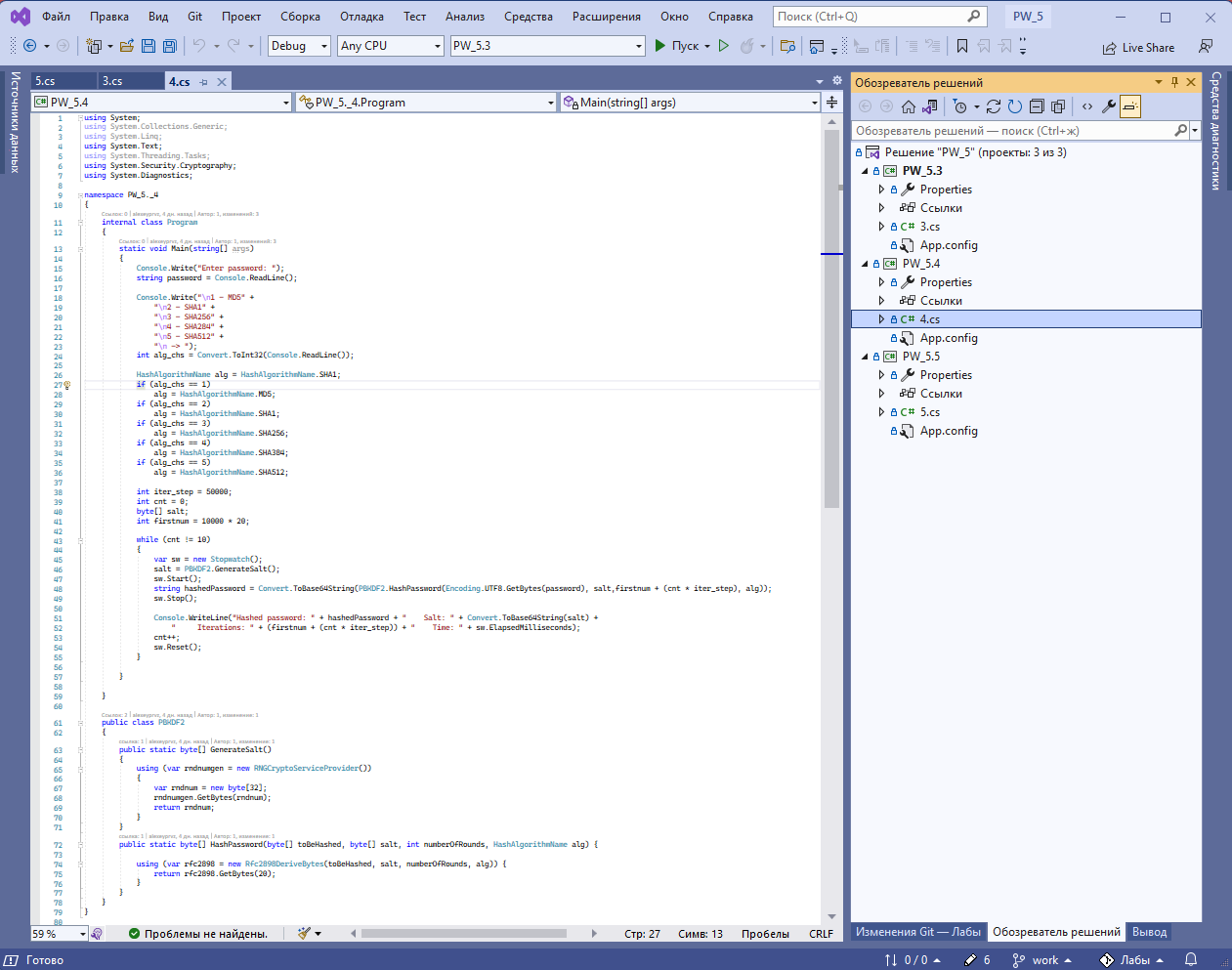
}

}

Виконання програми:



1. Розробити клас PBKDF2, що має наступну функціональність: генерує "сіль", задає алгоритм хешування (MD5, SHA1, SHA256, SHA384, SHA512) та обчислює хеш для заданого числа ітерацій. Створити програму, що обчислює час, витрачений на обчислення хешу для різного числа ітерацій (10 значень із кроком 50'000; перше значення = номер варіанта \* 10'000 ). Побудувати графік залежності витраченого часу від числа ітерацій.



Для початку розберемо клас PBKDF2:

Перший метод відповідно генерує сіль:

public static byte[] GenerateSalt()

{

using (var rndnumgen = new RNGCryptoServiceProvider())

{

var rndnum = new byte[32];

rndnumgen.GetBytes(rndnum);

return rndnum;

}

}

А ось другий метод вже ініціює новий екземпляр классу Rfc2898DeriveBytes , в якому можна передати кількість ітерацій для та назву хеш-алгоритму для формування ключа:

public static byte[] HashPassword(byte[] toBeHashed, byte[] salt, int numberOfRounds, HashAlgorithmName alg) {

using (var rfc2898 = new Rfc2898DeriveBytes(toBeHashed, salt, numberOfRounds, alg)) {

return rfc2898.GetBytes(20);

}

}

В кінці повертаємо або 160, або 20 байт.

Відповідно вказуємо користувачу яка цифра відповідає якому алгоритму хешування:

Console.Write("Enter password: ");

string password = Console.ReadLine();

Console.Write("\n1 - MD5" +

"\n2 - SHA1" +

"\n3 - SHA256" +

"\n4 - SHA284" +

"\n5 - SHA512" +

"\n -> ");

int alg\_chs = Convert.ToInt32(Console.ReadLine());

В залежності від введеної цифри такий алгоритм і будемо використовувати:

HashAlgorithmName alg = HashAlgorithmName.SHA1;

if (alg\_chs == 1)

alg = HashAlgorithmName.MD5;

if (alg\_chs == 2)

alg = HashAlgorithmName.SHA1;

if (alg\_chs == 3)

alg = HashAlgorithmName.SHA256;

if (alg\_chs == 4)

alg = HashAlgorithmName.SHA384;

if (alg\_chs == 5)

alg = HashAlgorithmName.SHA512;

Далі потрібні змінні:

int iter\_step = 50000; // крок в 50000 ітерацій

int cnt = 0; // лічильник

byte[] salt; // для зберігання солі

int firstnum = 10000 \* 20; // перший крок: 10000 на номер варіанту

Далі відповідно викликаємо нашу функцію для генерування солі та хешування паролю. Також ініціюємо секундомір для обчислення часу хешування. Все це відбувається у циклі 10 разів зі збільшенням кількості ітерацій на 50000 кожного разу:

while (cnt != 10)

{

var sw = new Stopwatch();

salt = PBKDF2.GenerateSalt();

sw.Start();

string hashedPassword = Convert.ToBase64String(PBKDF2.HashPassword(Encoding.UTF8.GetBytes(password), salt,firstnum + (cnt \* iter\_step), alg));

sw.Stop();

Console.WriteLine("Hashed password: " + hashedPassword + " Salt: " + Convert.ToBase64String(salt) +

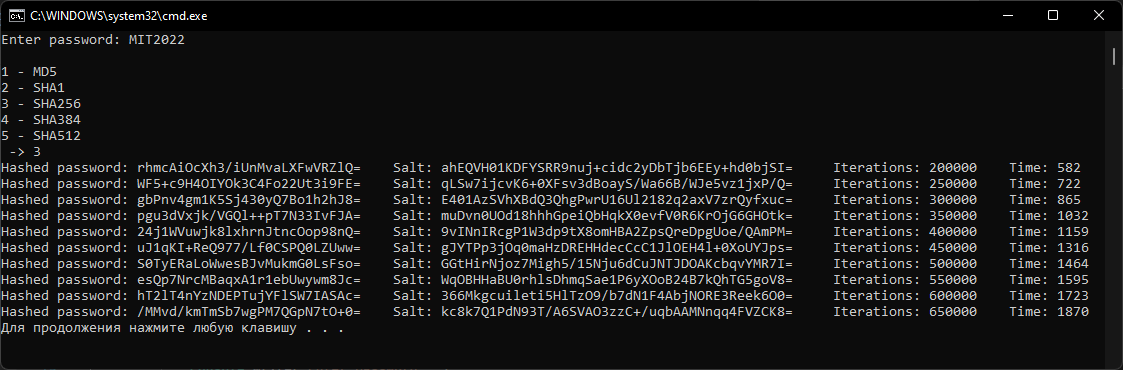
" Iterations: " + (firstnum + (cnt \* iter\_step)) + " Time: " + sw.ElapsedMilliseconds);

cnt++;

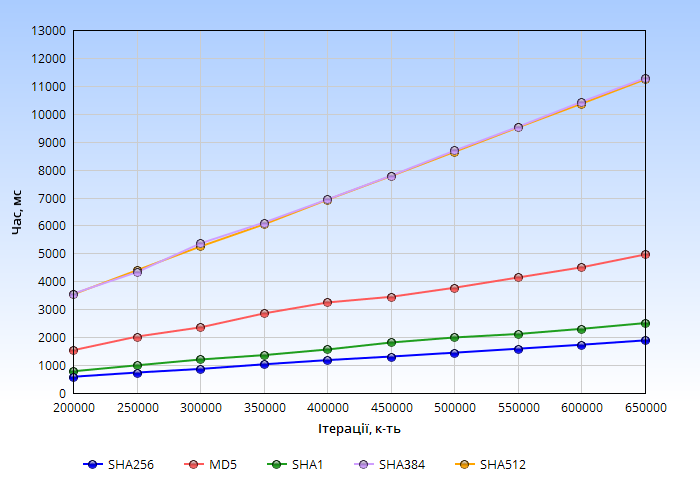
sw.Reset();

}

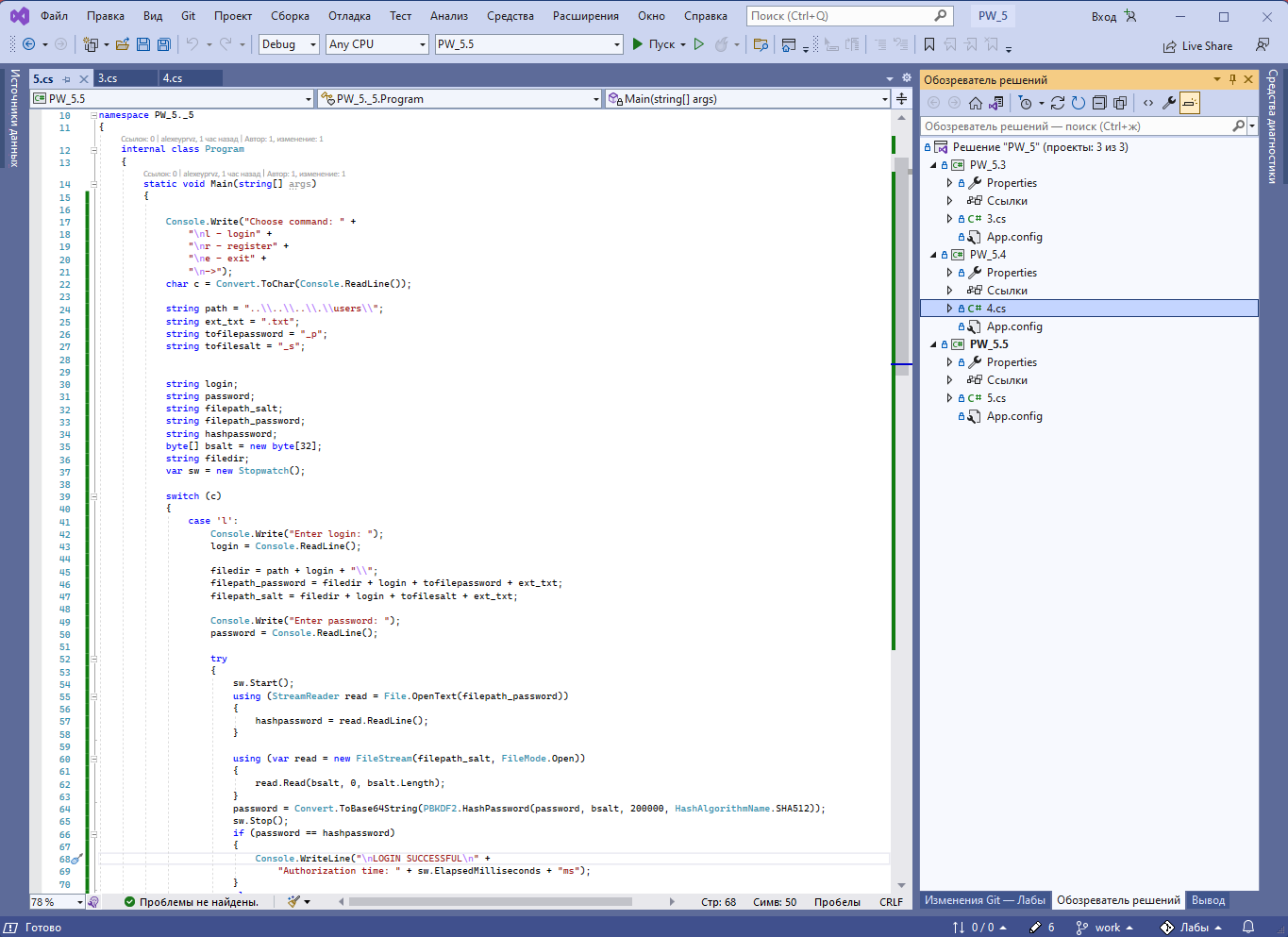
Виконання програми на прикладі одного алгоритму:

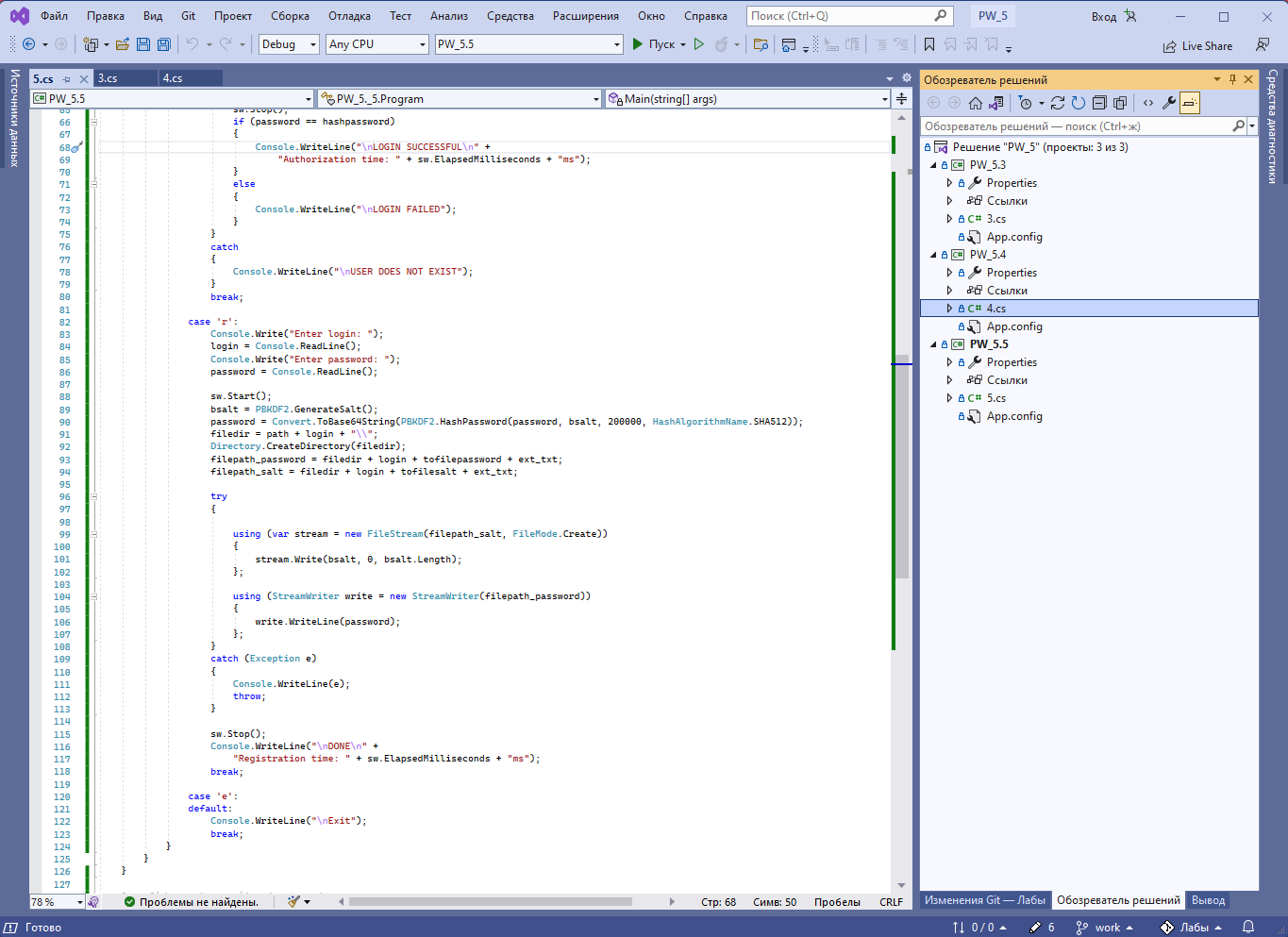


Графік:



1. Написати програму, що реалізує хешування введеного пароля під час реєстрації користувача та зберігає логін, пароль та "сіль" у пам'яті. Реалізувати можливість автентифікації за логіном і паролем. Число ітерацій = номер варіанта \* 10'000.





Класс PBKDF2 такий самий, як у минулому завданні, тому не зупиняюсь знову. Єдина різниця тільки, що інформацію для хешування передаю у вигляді рядка.

На початку програми пропонуємо користувачу або зареєструватися, або залогінитись:

Console.Write("Choose command: " +

"\nl - login" +

"\nr - register" +

"\ne - exit" +

"\n->");

char c = Convert.ToChar(Console.ReadLine());

Також вказуємо деякі змінні, які знадобляться нам пізніше:

string path = "..\\..\\..\\.\\users\\";

string ext\_txt = ".txt";

string tofilepassword = "\_p";

string tofilesalt = "\_s";

Перша змінна це шлях до папки з користувачами, друга – це розширення наших файлів, третя – це додаток до назви файлу, який зберігає хешований пароль, а четверта – це відповідно додаток до назви файлу, який зберігає сіль.

Розглянемо спочатку реєстрацію користувача.

Запит на введення логіну та паролю:

Console.Write("Enter login: ");

login = Console.ReadLine();

Console.Write("Enter password: ");

password = Console.ReadLine();

Далі генеруємо сіль:

bsalt = PBKDF2.GenerateSalt();

Далі хешуємо пароль:

password = Convert.ToBase64String(PBKDF2.HashPassword(password, bsalt, 200000, HashAlgorithmName.SHA512));

Далі створюємо папку в нашій директорії відповідно до логіну:

filedir = path + login + "\\";

Directory.CreateDirectory(filedir);

Також, використовуючи наші змінні, створюємо за допомогою конкатенації шляхи знаходження файлів:

filepath\_password = filedir + login + tofilepassword + ext\_txt;

filepath\_salt = filedir + login + tofilesalt + ext\_txt;

Далі створюємо файли у блоці try{ } catch{ }.

Записуємо сіль в один файл:

using (var stream = new FileStream(filepath\_salt, FileMode.Create))

{

stream.Write(bsalt, 0, bsalt.Length);

};

А хешований пароль в інший файл:

using (StreamWriter write = new StreamWriter(filepath\_password))

{

write.WriteLine(password);

};

В кінці виводимо повідомлення про завершення реєстрації та час, за який ми це виконали:

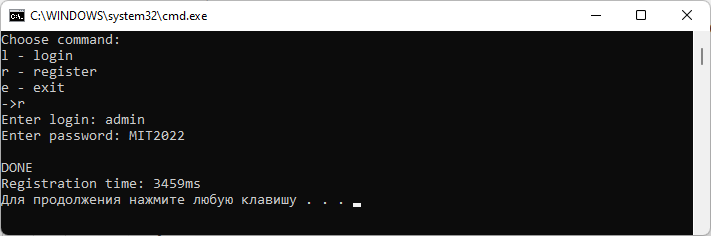
sw.Stop();

Console.WriteLine("\nDONE\n" +

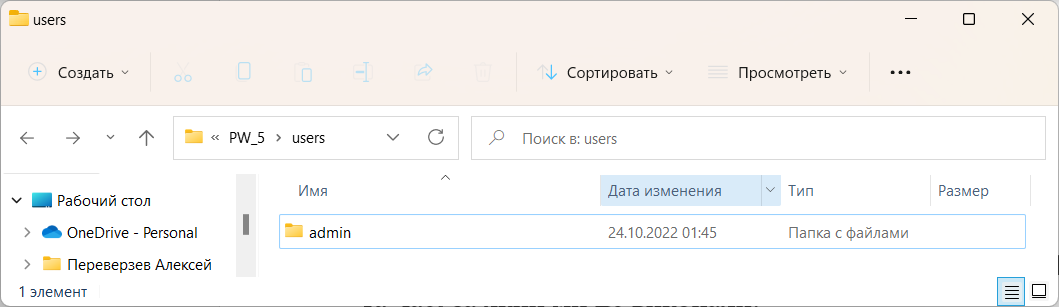
"Registration time: " + sw.ElapsedMilliseconds + "ms");

break;

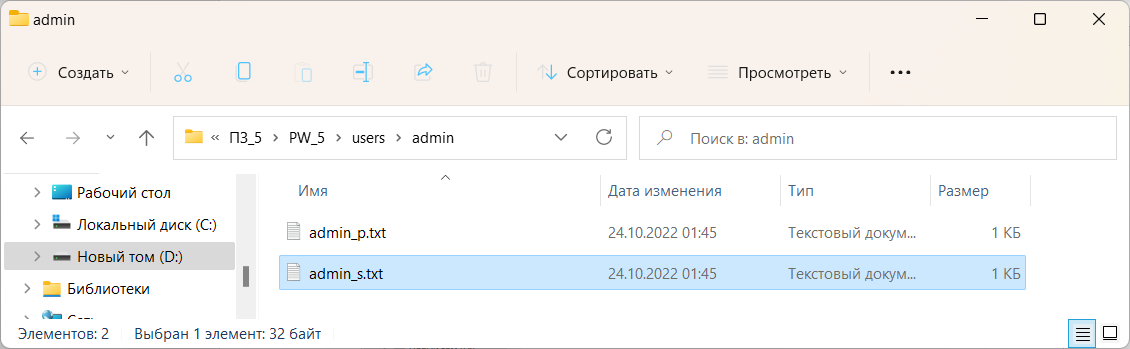
Виконання частини коду:

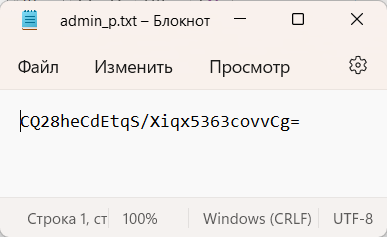
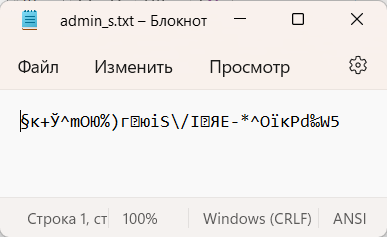


Створена папка:



Створені файли:



Розглянемо частину коду з авторизацією:

Щоб не акцентувати увагу на тому самому коді, розглянемо відмінності.

Відкриваємо файл з хешованим паролем та зчитуємо рядок, записавши це у змінну:

using (StreamReader read = File.OpenText(filepath\_password))

{

hashpassword = read.ReadLine();

}

Так само зчитуємо масив у файлі із сіллю:

using (var read = new FileStream(filepath\_salt, FileMode.Open))

{

read.Read(bsalt, 0, bsalt.Length);

}

Відповідно, використовуючи введений пароль та зчитану з файлу сіль, ствроюємо хеш функцію:

password = Convert.ToBase64String(PBKDF2.HashPassword(password, bsalt, 200000, HashAlgorithmName.SHA512));

Далі порівнюємо хеш функцію з файлу та тільки що створену. Якщо збігаються:

if (password == hashpassword)

{

Console.WriteLine("\nLOGIN SUCCESSFUL\n" + "Authorization time: " + sw.ElapsedMilliseconds + "ms");

}

Інакше:

else

{

Console.WriteLine("\nLOGIN FAILED");

}

Також, якщо було введено неправильний логін, то отримаємо помилку:

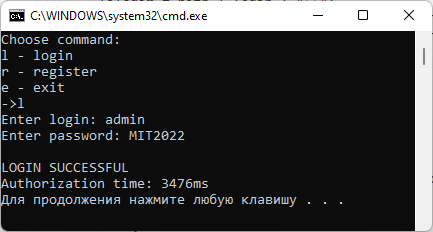
catch

{

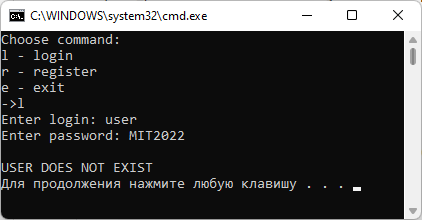
Console.WriteLine("\nUSER DOES NOT EXIST");

}

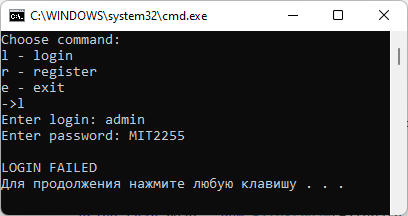
Виконання частини коду:



Невірний логін:



Невірний пароль:



**Висновки**: в ході практичної роботи ми дізналися як зберігати паролі різними методами, переваги та недоліки кожного з методів, дізнались що таке додаткова ентропія і для чого вона потрібна.