МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
(МОСКОВСКИЙ ПОЛИТЕХ)

Факультет информационных технологий
Кафедра «Инфокогнитивные технологии»

ЛАБОРАТОРНАЯ РАБОТА №1

на тему: *«Изучение свойств криптографических функций хеширования»*

Направление подготовки 09.03.03 «Прикладная информатика»
Профиль «Корпоративные информационные системы»
Дисциплина «Защита информации»

Выполнил:

студентка группы 201-361

Саблина Анна Викторовна

Проверил:

Харченко Елена Алексеевна

Москва 2023

Теоретическая часть

Secure Hash Algorithm 1 – это алгоритм криптографического хеширования. Для входного сообщения произвольной длины (максимум $2^{64} - 1$ бит, что примерно равно 2 эксабайта) алгоритм генерирует 160-битное (20 байт) хеш-значение, называемое также дайджестом сообщения, которое обычно отображается как шестнадцатеричное число длиной в 40 цифр. Используется во многих криптографических приложениях и протоколах.

Основываясь на данной информации, можно сделать вывод о том, что при $2^{160} + 1$ файлов минимум у 2-х будет одинаковый хеш. Таким образом, задача сводится к написанию программы, которая создаст 2^{160} файлов и проверит их хэши на совпадение.

Однако также есть информация, что для появления коллизии SHA-1 необходимо сгенерировать 2^{69} файлов.

Это число было вычислено в 2017 году в результате исследований уязвимостей SHA-1, которые показали, что существуют атаки на этот алгоритм, позволяющие создавать коллизии. Количество файлов, необходимых для создания коллизии, было оценено исходя из сложности атак на алгоритм.

Исследователи из Google и CWI Amsterdam использовали метод называемый "алгоритмом Шаттена" (англ. "the shattered algorithm"), который позволяет находить коллизии для SHA-1 с использованием более слабых вычислительных мощностей, чем ранее известные методы.

Они вычислили, что для создания коллизии SHA-1 необходимо сгенерировать 2^{69} файлов, что означает огромное количество вычислительных мощностей и времени. Однако, даже такое огромное число файлов не гарантирует, что коллизия будет найдена, это лишь верхняя оценка сложности атаки на алгоритм.

Несмотря на отсутствие подобных вычислительных мощностей и времени, можно написать довольно простую программу с заранее определенным количеством файлов, которая будет в перспективе находить коллизию для заданного файла. Для ее реализации рассмотрим метод замены схожих по начертанию букв русского и английского алфавитов, таких как «р», «о», «е», «у», «а», «х», «с». Проанализировав исходный файл, было выявлено, что в него входит более 800 символов «о», что при комбинировании подстановок создает сложность 2^{800}, которая является достаточной для создания коллизии. Таким образом, будем осуществлять замену русскоязычной «о» на английскую.

Для поставленной задачи будет достаточно заменять только букву «о» и использовать 64 бита (тип long), чтобы продемонстрировать принцип работы программы. Однако при увеличении мощностей возможны замена типа данных с long на класс BigInteger, обертку примитивного типа данных int, а также добавление в код программы подстановок других букв, перечисленных выше, замена пробелов нечитаемыми символами.

Практическая часть

Для реализации программы, генерирующей из файла leasing.txt эквивалентные по смыслу текстовые документы в количестве, достаточном (условно) для возникновения коллизии функции хеширования SHA-1, был выбран метод замены схожих по начертанию букв русского и английского алфавитов. Проанализировав исходный файл, было выявлено, что в него входит более 800 символов «о», что при комбинировании подстановок создает сложность 2^{800}, которая является достаточной для создания коллизии.

Далее был создан проект на языке Java.

Общий принцип работы программы:

1. В строковую переменную `fileContent` передается содержимое файла.

2. Для данного файла генерируется хэш SHA-1.
3. В *fileContent* производится поиск первых *elems* символов «o» и для каждого запоминается его позиция в документе.
4. Для каждой модификации `leasing.txt` создается новый файл.
5. Все созданные файлы в цикле сравниваются по хэшу с `leasing.txt`.

```
public class Main {
    static int elems = 64;
    static long[] oPositions = new long[elems];
    static String fileContent;
    static ArrayList<String> compareList = new ArrayList<>();

    public static void main(String[] args) {
        try {
            fileContent = Files.readString(Paths.get("leasing.txt"));
            generateSHA1("leasing.txt");
            int count = 0;
            int k = 0;
            for (int i = 0; i < fileContent.length(); i++) {
                if (fileContent.charAt(i) == 'o') {
                    count++;
                    if (count <= elems) {
                        oPositions[k] = i;
                        k++;
                    } else break;
                }
            }

            for (int i = 1; i <= Math.pow(2, 10); i++) {
                createFile(i);
            }

            String firstElement = compareList.get(0); // получить первый элемент массива
            for (int i = 1; i < compareList.size(); i++) {
                String currentElement = compareList.get(i);
                if (currentElement.equals(firstElement))
                    System.out.println("Найдена коллизия с файлом №" + i
                            + ":" + currentElement);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Листинг 1 – Декларация глобальных переменных и метод main

Метод `createFile(int fileNumber)` получает на вход порядковый номер модификации, которую нужно совершить с исходником. С помощью побитовых операций для переменной данного порядкового номера вычисляются позиции подстановок и затем выполняются сами подстановки. После чего создается новый файл на основе полученной измененной строки. Затем для данного файла вызывается метод *generateSHA1*.

```
protected static void createFile(int fileNumber) {
    StringBuilder temp = new StringBuilder(fileContent);

    // замена русскоязычной буквы "о" на "o" английскую
    for (int i = elems - 1; i >= 0; i--) {
        // получаем бит по индексу i для проверки необходимости замены буквы
        // на данной позиции
        int bit = (fileNumber >> i) & 1;
        if (bit == 1)
            temp.setCharAt(Math.toIntExact(oPositions[i]), 'o');
    }

    try {
        // создание нового файла
        String path = "src/txt/" + fileNumber + ".txt";
        Files.write(Paths.get(path), temp.toString().getBytes(),
             StandardOpenOption.CREATE);

        generateSHA1(path);
    } catch (IOException e) {
        System.out.println("An error occurred: " + e.getMessage());
    }
}
```

Листинг 2 – Метод createFile(int fileNumber)

Метод `generateSHA1(String path)` получает на вход путь до файла, для которого необходимо сгенерировать хэш-код. Формируется и выполняется команда для cmd со следующими аргументами: openssl, dgst, -sha1, `path`. Результат выполнения команды для каждого файла выводится в командную строку и добавляется к списку всех сгенерированных хэш-кодов с помощью метода *`addToCompare`*.

```
protected static void generateSHA1(String path) {
    try {
        ProcessBuilder builder = new ProcessBuilder("openssl", "dgst", "-sha1", path);

        // сообщает процессу объединять стандартный вывод
        // и стандартный поток ошибок в один поток,
        // который мы можем прочитать в Java
        builder.redirectErrorStream(true);
        Process process = builder.start();

        // для чтения вывода процесса
        BufferedReader reader =
             new BufferedReader(new InputStreamReader(process.getInputStream()));
        String line;

        // читаем каждую строку вывода процесса и выводим ее в консоль
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
            addToCompare(line);
        }

        // чтобы дождаться завершения процесса и получить код ошибки,
        // если таковой имеется
        int exitCode = process.waitFor();
        System.out.println("Exited with error code " + exitCode);
    } catch (InterruptedException | IOException e) {
        throw new RuntimeException(e);
```

```
        }
    }
```

Листинг 3 – Метод generateSHA1(String path)

Метод `addToCompare(String output)` получает на вход результат работы cmd в виде строки, обрезает ее до значения хэша и добавляет в список сравнения.

```
    private static void addToCompare(String output) {
        compareList.add(output.substring(output.indexOf("= ") + 2));
    }
}
```

Листинг 4 – Метод addToCompare(String output)

Ссылка на проект в репозитории GitHub:

– https://github.com/LazyShAman/dp/tree/main/1.