Assignment 1
Brute Force Attack Estimation

L. Towell, L.M.Towell@liverpool.ac.uk

November 6, 2019

Code repository: https://github.com/luketowell/COMP522-Assignment1

1. Password List

Below is the password list that I have decided to use for this exercise.

N	Password
1	abc
2	P@ssW0rD
3	Th!$IsAV3ryL0n9pA$$w0rd

2. Salt and Iteration Count

For the purpose of this coursework I have hardcoded the salt used within this program, I have also decided to use an iteration count of 1024. The below table shows the time per run and the average time taken in milliseconds over 5 iterations to encrypt and decrypt the string "This is an example string" using the defined salt and iteration counts. *Appendix A* details the code used for executing the encryption and decryption program. *Appendix B* Is the output produced by the code.

The below timings have been recorded running the program on a Macbook air with an Intel core i5 processor and 16GB of RAM. Timings on other machines are likely to differ.

Iteration time in milliseconds (ms)			
N	abc	P@ssW0rD	Th!$IsAV3ryL0n9pA$$w0rd
1	155.77	2.45	1.46
2	8.55	1.66	1.19
3	6.27	1.65	1.24
4	12.94	4.0	1.25
5	1.57	2.01	2.69
Average	7.33	2.35	1.57
The first iteration of abc has been omitted from the average			

3. Brute Force Attack Estimation

Given that it is specified that the attacker knows the salt, iteration count, encryption type, input string and cipher text used to encrypt the input string then the only piece of information the attacker would need to find is the password used to encrypt the string. In order to work out the password using a Brute force attach the attacker is going to have to iterate through each possible character that could be used in each different combination.

If we presume that passwords are made of uppercase, lowercase, numbers and special characters and they are limited to traditional ASCII encoded characters then this gives the attacker a possible 95 characters for each character in the password. The equation for working out a password via brute force attack is therefore 95^n where n is the number of characters within the password. The table below shows the amount of possible combinations for N character lengths.

Number of characters	Possible password combinations
1	95
2	9025
3	857375
4	81450625
5	7737809375
6	735091890625
7	69833729609375
8	6634204312890620
9	630249409724609000
10	59873693923837900000
15	463291230159753000000000000000
20	358485922408542000000000000000000000000
23	307356867725024000000000000000000000000000000
25	277389573121834000000000000000000000000000000000

In order to estimate the time taken to brute force an attack we need to look at the time taken to iterate through every combination possible considering that in the worse case the last combination will be the password we are attempting to discover. The calculation for estimating time taken to discover a password is therefore the time taken to try one password (one iteration) mutliplied by the number of possible combinations. For example if we say the time taken for 1 iteration is 1ms then we can assume that it would take in the worst case to crack a 3 character password (abc) which would be 95^3 (857375 combinations) * the taken to complete one iteration e.g. 1ms per attempt would take 857375ms or 857.375 seconds or 14.3 minutes.

Figure. 1 is a table of estimated time taken to discover passwords of varying character lengths using the brute force method. You can see that the longer the character length the longer the time taken growing to time periods which will never be broken within our lifetime after just 5 characters.

Taking *Figure. 1* into account I have estimated the time taken to break my passwords using my average encryption time of 1.56ms per iteration in the table below.

Password	characterLength	Estimation to discover
abc	3	21mins
P@ssW0rD	8	315,554 years
Th!$IsAV3ryL0n9pA$$w0rd	23	105 Septillion Years

N of Characters	Time in milliseconds	Time in seconds	Minutes
1	142.5	0.14	0
2	13537.5	14	0
3	1286062.5	1286	21
4	122175937.5	122176	2036
5	11606714063	11606714	193445
6	1.10264E+12	1102637836	18377297
7	1.04751E+14	104750594414	1745843240
8	9.95131E+15	9951306469336	165855107822
9	9.45374E+17	945374114586914	15756235243115
10	8.98105E+19	89810540885756800	1496842348095950
15	6.94937E+29	694936845239630000000000000	1158228075399380000000000
20	5.37729E+39	537728883612813000000000000000000000000	896214806021356000000000000000000000
23	4.61035E+45	461035301587536000000000000000000000000000	768392169312560000000000000000000000000
25	4.16084E+49	416084359682751000000000000000000000000000000	693473932804585000000000000000000000000000

N of Characters	Hours	Days	Years
1	0	0.00	0.00
2	0	0.00	0.00
3	0	0.01	0.00
4	34	1.41	0.00
5	3224	1.00	0.00
6	306288	12762.01	34.96
7	29097387	1212391.14	3322
8	2764251797	115177158	315554
9	262603920719	10941830030	29977617
10	24947372468266	1039473852844	2847873569
15	193038012566564000000000	8043250523606830000000	22036302804402300000
20	14936913433689300000000000000000000	6223713930703860000000000000000000	17051271043024300000000000000000
23	12806536155209300000000000000000000000000	5336056731337220000000000000000000000000	14619333510512900000000000000000000000
25	11557898880076400000000000000000000000000000	4815791200031840000000000000000000000000	13193948493237900000000000000000000000000

Figure 1: Table of time taken to discover characters assuming 1 iteration takes 1 millisecond.

4. How Does Iteration Count Affect Brute Force Timing?

An iteration count is the number of times the password is hashed when generating the key used for encryption. There for without the correct iteration count the hacker is going to be unable to obtain the correct key and therefore will be unable to correctly tell if they've reached the correct password.

5. Brute Force Attack Estimation Without Known Iteration Count

If a hacker does not know the iteration count of an application then they have to try the same brute force attack multiple times for multiple iteration counts which essentially means that the time taken grows exponentially. E.g. If the iteration count was 300 and the hacker started their attempt with an iteration count of 1 they would have to work out the characters in the password $95^n * i$ where i is the number of iterations needed to reach the one initially used when hashing the password e.g. $95^n * 300$ would be the maximum number of possible combinations to discover the correct password and correct iteration count.

6. Comparison with Online Services

The table below is the estimation of how long I believe it would take to discover my passwords compared to what an online website has estimated.

Password	My Estimation	Online Estimation
abc	21mins	400 ns
P@ssW0rD	315,554 years	9 hrs
Th!$IsAV3ryL0n9pA$$w0rd	105 Septillion Years	19 Septillion Years

There are a number of potential reasons for the difference between my service(*Appendix A*) and the online service(*https://howsecureismypassword.net/*). I have listed some of them below:

1. Different Attack Methods It is not specified on the web application which methods are being used to crack the password that are provided to the service. The first password 'abc' is estimated to be discovered in 400ns. One of the potential reasons that this password is so easy to crack could be that the service is also considering time taken using something like a dictionary attack which looks for commonly used phrases or passwords. In this case a password like 'abc' is very easily discovered because of how common it is.

2. Larger Computational Power The online web application is likely running on a more powerful server / multiple servers which will produce a faster iteration speed which will mean that the user of the machine would be able to try more iterations in a shorter time period hence why the times generated by the web application are considerably faster than the times generated by my machine.

   Another variation of more computational power would be multithreaded / multicore machines which would then mean that the program could attempt multiple passwords at a time which would mean that the iteration count could be divided across the multiple threads or cores. This would mean that the attacker can attempt to crack the password in a shorter period of time than my single core machine.

Appendices

A Password Encryption & Decryption Program

```
import javax.crypto.Cipher;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.PBEParameterSpec;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.security.Key;

/**
 * Example of using Password-based encryption
 */

public class PasswordBasedEncryption {
    public static void main(String[] args) throws Exception {
        // setup iteration count
        int iterationCount = 5;

        // Setup passwords used to encrypt values

        String[] passwords = new String[] { "abc", "P@ssW0rD", "Th!$IsAV3ryL0n9pA$$w0rd" };
        System.out.println("Password based encryption timings");

        for (int i = 0; i < passwords.length; i++) {
            System.out.println("------------------------------");
            // initialise time array
            double[] time = new double[6];
            System.out.println("Password used: " + passwords[i]);

            for (int j = 0; j < iterationCount; j++) {
                PBEKeySpec pbeKeySpec;
                PBEParameterSpec pbeParamSpec;
                SecretKeyFactory keyFac;
                // start timing
                long startTime = System.nanoTime();
                // Salt
                byte[] salt = { (byte) 0xc7, (byte) 0x73, (byte) 0x21, (byte) 0x8c, (byte) 0x7e,
                    (byte) 0xc8,
                        (byte) 0xee, (byte) 0x99 };

                // Iteration count
                int count = 1024;

                // Create PBE parameter set
                pbeParamSpec = new PBEParameterSpec(salt, count);

                // Initialization of the password
                char[] password = passwords[i].toCharArray();

                // Create parameter for key generation
                pbeKeySpec = new PBEKeySpec(password);

                // Create instance of SecretKeyFactory for password-based encryption
                // using DES and MD5
                keyFac = SecretKeyFactory.getInstance("PBEWithMD5AndDES");

                // Generate a key
                Key pbeKey = keyFac.generateSecret(pbeKeySpec);
```

```
            // Create PBE Cipher
            Cipher pbeCipher = Cipher.getInstance("PBEWithMD5AndDES");

            // Initialize PBE Cipher with key and parameters
            pbeCipher.init(Cipher.ENCRYPT_MODE, pbeKey, pbeParamSpec);

            // Our plaintext
            byte[] cleartext = "This is an example string".getBytes();

            // Encrypt the plaintext
            byte[] ciphertext = pbeCipher.doFinal(cleartext);
            System.out.println("Cipher text: " + Utils.toHex(ciphertext));

            pbeCipher.init(Cipher.DECRYPT_MODE, pbeKey, pbeParamSpec);

            // Decrypt the plaintext
            byte[] ciphertext2 = pbeCipher.doFinal(ciphertext);
            String plainText = new String(ciphertext2);

            // end time
            long endTime = System.nanoTime();

            // calculate total time and add to the time array
            long totalTime = endTime - startTime;

            double totalTimeMs = totalTime / 1000000.0;
            BigDecimal totalTimeMsRounded = new BigDecimal(totalTimeMs).setScale(2,
                RoundingMode.HALF_EVEN);
            double roundedTotalTime = totalTimeMsRounded.doubleValue();
            time[j] = roundedTotalTime;

            // output of all times and key components of the encryption algorithm
            System.out.println("Decrypted text: " + plainText);
            System.out.println("loop " + (j + 1) + ": " + roundedTotalTime + "ms");
        }
        // summed time calculation to output average over the iteration counts.
        double summedTime = 0;
        for (var k = 0; k < time.length; k++) {
            summedTime += time[k];
        }

        // Work out and print the average time for each password

        System.out.println("Total time:" + new BigDecimal(summedTime).setScale(2,
            RoundingMode.HALF_EVEN));
        double avgTime = summedTime / iterationCount;
        System.out.println("average time:" + new BigDecimal(avgTime).setScale(2,
            RoundingMode.HALF_EVEN));

    }
  }
}
```

B Output of Appendix A when ran in terminal

```
Password based encryption timings
-----------------------------
Password used: abc
Cipher text: c4d03d30be1dfdd3c5ace92ebe5bacb959c80a9e9213a21dd5615cf275e8688f
Decrypted text: This is an example string
loop 1: 117.31ms
Cipher text: c4d03d30be1dfdd3c5ace92ebe5bacb959c80a9e9213a21dd5615cf275e8688f
Decrypted text: This is an example string
loop 2: 1.56ms
Cipher text: c4d03d30be1dfdd3c5ace92ebe5bacb959c80a9e9213a21dd5615cf275e8688f
Decrypted text: This is an example string
loop 3: 2.09ms
Cipher text: c4d03d30be1dfdd3c5ace92ebe5bacb959c80a9e9213a21dd5615cf275e8688f
Decrypted text: This is an example string
loop 4: 1.96ms
Cipher text: c4d03d30be1dfdd3c5ace92ebe5bacb959c80a9e9213a21dd5615cf275e8688f
Decrypted text: This is an example string
loop 5: 2.32ms
Total time:125.24
average time:25.05
-----------------------------
Password used: P@ssW0rD
Cipher text: b639839be3610610478c09998f8ede508799d641f60110840dd59b2e0790b125
Decrypted text: This is an example string
loop 1: 2.11ms
Cipher text: b639839be3610610478c09998f8ede508799d641f60110840dd59b2e0790b125
Decrypted text: This is an example string
loop 2: 2.14ms
Cipher text: b639839be3610610478c09998f8ede508799d641f60110840dd59b2e0790b125
Decrypted text: This is an example string
loop 3: 1.74ms
Cipher text: b639839be3610610478c09998f8ede508799d641f60110840dd59b2e0790b125
Decrypted text: This is an example string
loop 4: 1.67ms
Cipher text: b639839be3610610478c09998f8ede508799d641f60110840dd59b2e0790b125
Decrypted text: This is an example string
loop 5: 1.7ms
Total time:9.36
average time:1.87
-----------------------------
Password used: Th!$IsAV3ryL0n9pA$$w0rd
Cipher text: fffecd4d43d0e33255d4c82f86a71f235257be392215d5f1aa9a33a14b884e51
Decrypted text: This is an example string
loop 1: 1.7ms
Cipher text: fffecd4d43d0e33255d4c82f86a71f235257be392215d5f1aa9a33a14b884e51
Decrypted text: This is an example string
loop 2: 1.67ms
Cipher text: fffecd4d43d0e33255d4c82f86a71f235257be392215d5f1aa9a33a14b884e51
Decrypted text: This is an example string
loop 3: 1.64ms
Cipher text: fffecd4d43d0e33255d4c82f86a71f235257be392215d5f1aa9a33a14b884e51
Decrypted text: This is an example string
loop 4: 1.62ms
Cipher text: fffecd4d43d0e33255d4c82f86a71f235257be392215d5f1aa9a33a14b884e51
Decrypted text: This is an example string
loop 5: 1.6ms
Total time:8.23
average time:1.65
```