Brute Force

Authors

- Reuben Rappaport (rbr76)
- Alex Clark (ayc55)

Part 1 - Salted Myspace Passwords

Formulas

There are 37144 passwords in the Myspace dataset. We started by computing some statistics on these - the median password length is 8 characters but the standard deviation is 43.7, due to the fat tail with quite a few outliers. The hashing algorithm used - SHA-512 - breaks input messages into 1024 bit blocks before processing. Of the passwords in myspace.txt only five are long enough when salted that they require multiple blocks. Two of these are long passphrases which take up 2 blocks. Another one of these is the string "40RI304TJJH" repeated a large number of times which requires 13 blocks, a fourth consists entirely of the character "#" and requires 42 blocks while the largest is the string "abcdefgh" repeated many times and requires 50 blocks. Hashing all of these passwords with a single salt would require $37144 + 1 + 1 + 12 + 41 + 49 = 37248$ SHA-512 iterations.

There are 2^{32} possible 4 byte salts. To calculate the total number of SHA-512 iterations we multiply this number by the number of iterations required to hash each password with a single salt. This gives us

$$2^{32} * 37248 \approx 1.6 \times 10^{14}\ SHA - 512\ Iterations$$

Our dictionary consists of $((Password, Salt), Hash)$ tuples. Each hash is 64 bytes, and each salt is 4 bytes. The median password only requires 8 bytes but we decided that since we already had the dataset and could do so we should calculate the cost exactly. To store all the passwords requires

$$\begin{aligned} Storage\ Cost &= \sum_{i=1}^{|Passwords|} (Length(Password_i) + Salt\ Size + Hash\ Size) * Possible\ Salts \\ &= \sum_{i=1}^{37144} (Length(Password_i) + 4\ Bytes + 64\ Bytes) * 2^{32} \\ &\approx 1.22 \times 10^{16}\ Bytes \end{aligned}$$

Empirical Observations

To empirically test how long it would take to hash all possible strings we wrote java code which hashed all of the passwords in the dataset with a salt and printed the resulting dictionary to a file and then ran it with 100 randomly generated 4 byte salts.

```
public static byte[] genSalt() {
    byte[] newsalt = new byte[4];
    rand.nextBytes(newsalt);
    return newsalt;
}

public static void main(String[] args) {
  .
  .
  .
  for (int i = 0; i < start; i++) {
        reader.readLine();
  }
  long startTime = System.currentTimeMillis();
  System.out.println("Start time:" + startTime);
  for (int i = start; i < end; i++) {
    read = reader.readLine();
    for (int j = 0; j < numSalts; j++) {
        salt = genSalt();
        saltstring = new String(salt, StandardCharsets.UTF_8);
        hashed = SHA_512_Hash(read, saltstring);
        writer.write(read + ", " + saltstring + ", " + hashed);
        writer.newLine();
    }
  }
  long endTime = System.currentTimeMillis();
  System.out.println("End time:" + endTime);
  .
  .
  .
}
```

We ran this code on a MacBook Air with the following specs

Specification	Value
Processor	Intel Core i5
Number of Cores	2

Specification	Value
Processor Speed	1.6 Gigahertz
L2 Cache (per core)	256 Kilobytes
L3 Cache	3 Megabytes
RAM	4 Gigabytes
Operating System	MacOS Sierra Version 10.12.2

We got the following results:

Metric	Value
Salts Per Password Used	100 Salts
Total Elapsed Time	27047 Milliseconds
Average Time to Hash a Salted Password	0.00728 Milliseconds
Average Time to Hash a Salted Dataset	270.47 Milliseconds
Total Space Used By Dictionary	553.6 Megabytes
Average Space to Store a Salted Dataset	5.536 Megabytes

We extrapolated from these averages to compute the total time and space required for the dictionary

$$\begin{aligned}
Total\ Time &= Time\ to\ Hash\ a\ Salted\ Dataset * Number\ of\ Salted\ Datasets\\
&= 270.47 \times 10^{-3}\ Seconds * 2^{32}\\
&\approx 1.16 \times 10^{9}\ Seconds\\
&\approx 36.81\ Years\\
Total\ Space &= Space\ to\ Store\ a\ Salted\ Dataset * Number\ of\ Salted\ Datasets\\
&= 5.536 \times 10^{6}\ Bytes * 2^{32}\\
&\approx 2.38 \times 10^{16} Bytes
\end{aligned}$$

Conclusions

Our theoretical calculations told us that computing the entire dictionary would require about 1.6×10^{14} SHA-512 calculations and take on the order of 10^{16}

bytes to store. Running an experiment on a MacBook air suggested that in terms of actual time expended these calculations would take approximately 36.81 years and gave us a storage estimate on the same order of magnitude as our theoretical calculations. The difference is explained by the fact that the hexadecimal encoded representation of a 64 byte hash takes 128 characters, doubling the space to store hashes in our experiment.

Part 2 - Random Alphanumeric Passwords

Formulas

The character set `a-z`, `0-9` contains 36 different characters. This means that there are 36^n possible length n strings drawn from this alphabet. In this problem we are considering all strings in this alphabet of size ≤ 8. The total possible number of such strings is

$$\sum_{i=1}^{8} 36^i \approx 2.90 \times 10^{12} \textit{ Possible Strings}$$

Since the longest strings we're considering are only 8 bytes and we're not using salts, we don't need to worry about SHA-512 breaking them up into multiple blocks. To build the dictionary we will have to go through a single hashing iteration for each possible string. That is, we will need approximately 2.90×10^{12} SHA-512 Iterations.

Our dictionary consists of $(Password, Hash)$ tuples. Each hash consists of 64 bytes while each password takes up somewhere between 1 and 8 bytes. We can calculate the total cost as follows

$$\begin{aligned} Storage\ Cost &= \sum_{i=1}^{8} 36^i * (Hash\ Size + i) \\ &= \sum_{i=1}^{8} 36^i * (64 + i) \\ &\approx 2.09 \times 10^{14}\ Bytes \end{aligned}$$

Empirical Observations

To empirically test how long it would take to compute the whole dictionary we wrote java code which randomly generated 2^{16} strings from the alphabet `a-z`, `0-9` with their lengths distributed in the same fashion as the possibility space. We hashed each string and stored it along with said hash in a dictionary file.

```java
public static String genAlphanumeric() {
    final String alpha = "0123456789abcdefghijklmnopqrstuvwxyz";

    //Segment probability space based on combinatoric distribution
    double onethreshold = 36.0 / 2821109907456.0;
    double twothreshold = onethreshold + 1296.0 / 2821109907456.0;
    double threethreshold = twothreshold + 46656.0 / 2821109907456.0;
    double fourthreshold = threethreshold + 1679616.0 / 2821109907456.0;
    double fivethreshold = fourthreshold + 60466176.0 / 2821109907456.0;
    double sixthreshold = fivethreshold + 2176782336.0 / 2821109907456.0;
    double seventhreshold = sixthreshold + 78364164096.0 / 2821109907456.0;

    int len;
    double seed = Math.random();
    if (seed < onethreshold) {
        len = 1;
    }
    else if (seed < twothreshold) {
        len = 2;
    }
    else if (seed < threethreshold) {
        len = 3;
    }
    else if (seed < fourthreshold) {
        len = 4;
    }
    else if (seed < fivethreshold) {
        len = 5;
    }
    else if (seed < sixthreshold) {
        len = 6;
    }
    else if (seed < seventhreshold) {
        len = 7;
    }
    else len = 8;

    StringBuilder build = new StringBuilder(len);
    for (int i = 0; i < len; i++) {
        build.append(alpha.charAt(rand.nextInt(alpha.length())));
    }

    return build.toString();
}

public static void main(String[] args) {
  .
  .
  .
  int total = end - start;
    String[] passwords = new String[total];
    for (int i = 0; i < total; i++) {
        passwords[i] = genAlphanumeric();
    }
    long startTime = System.currentTimeMillis();
    System.out.println("Start time:" + startTime);
    for (int i = start; i < end; i++) {
        read = passwords[i];
        hashed = SHA_512_Hash(read, "");
        writer.write(read + ", " + hashed);
        writer.newLine();
    }
    long endTime = System.currentTimeMillis();
    System.out.println("End time:" + endTime);
  .
  .
  .
}
```

We ran this code on the same machine as for part 1 - to see its specs reference the previous table. We obtained the following results

Metric	Value
Total Number of Passwords	2^{16} Strings
Total Elapsed Time	924 Milliseconds
Average Time to Hash a Password	0.0141 Milliseconds
Total Space Used By Dictionary	9.1 Megabytes
Average Space to Store a Password and Hash	139 Bytes

We extrapolated from these averages to compute the total time and space required to build the full dictionary

$$
\begin{aligned}
Total\ Time &= Time\ to\ Hash\ a\ Password * Number\ of\ Possible\ Passwords \\
&= 0.0141\ \times 10^{-3}\ Seconds * 2.90 \times 10^{12} \\
&\approx 4.06 \times 10^{7}\ Seconds \\
&\approx 67.13\ Weeks \\
Total\ Space &= Space\ To\ Store\ a\ Password\ and\ Hash * Number\ of\ Possible\ Passwords \\
&= 139 Bytes * 2.90 \times 10^{12} \\
&\approx 4.03 \times 10^{14} Bytes
\end{aligned}
$$

Conclusions

Our theoretical calculations told us that computing the whole dictionary would take 2.90×10^{12} SHA-512 calculations and take on the order of 10^{14} bytes to store. Running an experiment on a Macbook air determined that this would take about 67.13 weeks to compute. Again our experimental result for storage was a little less than double our theoretical result due to the hexadecimal encoding of hashes.

Part 3 - Comparison

Dictionary (1) based on the salted MySpace passwords takes more resources to compute, requiring approximately 1.6×10^{14} SHA-512 iterations. We experimentally determined that running on a Macbook Air this would take approximately 37 years. This dictionary would also use on the order of 10^{16} bytes of memory.

Dictionary (2), in contrast, would require approximately 2.90×10^{12} SHA-512 iterations. Experimentally, this would require about 67 weeks running on the same laptop. This dictionary would use on the order of 10^{14} bytes of memory.

Taken together, this shows that dictionary (1) requires significantly more computing resources.

Part 4 - Stolen Database

In contrast, if the attacker stole a hashed password database, including salts, the situation in dictionary (1) would require far fewer computational resources in order to crack a single user's password.

This is because dictionary (1) has only 37144 possible passwords, and with a database there is no need to compute the hash for these passwords plus all possible salts, assuming that a salt is stored with its corresponding hashed password.

Dictionary (2), on the other hand, has 2.90×10^{12} possible passwords, and cracking a single password would require computing the hash for each of these, in a worst-case scenario. In fact, this requires no less resources than without a database, since there are no salts with which to gain an advantage by not needing to compute all possible salts.

Part 5 - System Administration

Generally speaking, the MySpace password system (1) is superior in situations where the password database is not stolen, and the alphanumeric password system (2) is superior when a password database is stolen. On the whole, we would judge system (2) to be superior because in the event that a database is stolen, the security of the MySpace password system drops to effectively nil, since the computing resources needed to check 37144 passwords is negligible. The alphanumeric password system, meanwhile, still maintains a reasonable level of security which is not compromised by the database being exposed. This is an example of defense in depth, where users can continue to be relatively safe even if one security mechanism (whatever defenses are used to protect the password database) is compromised.

There are other criteria worth considering, however, one of which is ease of use. With the MySpace password set, it is possible that a user would be assigned a password which is effectively impossible to remember, whether due to length or lack of patterns to remember. That said, the vast majority of MySpace passwords would be easier to remember than the alphanumeric passwords might be, many of the former simply being words or words with single numbers/characters appended. In contrast, some users might find it difficult to remember a randomly generated alphanumeric password which would most frequently be 8 characters long, although one could use mnemonics for this similar to remembering a phone number.

Psychological acceptability for the user is related to the above, however, and is far more likely to be a concern with the MySpace passwords, since a user is more likely to take issue with a forcibly assigned and potentially unchangeable password of “fuckyou” than one like “abxy3792”.

Although neither design’s security depends on secrecy, in an open design situation the MySpace passwords would be advantageous. This is because knowing the details of their generation would still require one to compute salts for every listed password in building a dictionary, which as mentioned before would require approximately 36 years on a comparable computer to that which we tested on. In contrast, knowing the details of the alphanumeric passwords would let one know that salts are not used, and that generating a dictionary is thereby much easier than it could conceivably have been otherwise.

Considering entropy, the average entropy of an 8-character alphanumeric password is $8 \log_2 36$, with 8-character being overwhelmingly most likely, which equates to approximately 41 bits of Shannon entropy. The MySpace passwords, while their median length is identical to the median length of the alphanumeric passwords, are primarily composed of English words, which decreases randomness. However, they have a larger alphabet space in terms of which characters are allowed to be used (symbols and capitals being available, unlike with the alphanumeric passwords), which leads to a potentially greater degree of entropy.

Concerning deployability, there is no real difference between these two password configurations.

Overall, though both configurations have many flaws, we would choose to use configuration (2) of the two provided choices.