Analysis of Hashing used in Appu[1]

Nikita Mundhada, Yucheng Liu
M.S. in Information Security, GaTech
December 9, 2013

***Abstract*- Appu is a chrome extension built by researchers at Georgia Tech. It allows a user to reduce his personal information footprint across the web [1]. Appu needs to detect password patterns of users in order to be able to warn him about password reuse. It tweaks the SHA-256 scheme to store the hashes of passwords using the new scheme on the user's machine. We have analyzed` the security of this tweaked scheme and proved that it is insecure.**

I. Introduction

With so many web applications and sites, it's hard for a user to keep track of where his personal information resides. Appu makes this job easy by keeping track of personal information such as passwords, username, birthdate, address, credit card numbers, and social security number so that a user can find out all sites that store a particular bit of his personal information. Appu also downloads personal information from sites where you own an account and tries to prevent password reuse [1]. To do this, on the user's machine it stores the website name, long hash and short hash of password. The long hash is the millionth time hash value of the password i.e. the password is hashed a million times. The short hash is the last 32 bits of the hash when done the first time. Appu uses SHA-256 to compute both the long hash and the short hash. We will be analyzing the security of this tweaked hashing scheme - that works by computing a long hash in combination a short hash from point of view of collision resistance and resistance to dictionary attacks.

II. Analysis of Collision resistance of hash used by Appu

We will prove that long hash is collision resistant and short hash is not collision resistant. Long hash collision resistance is needed by Appu because the server decides two passwords are different if their long hashes do not match and if two different passwords hash to the same value under long hash, then this would confuse the server. Appu warns users about password reuse using the short hash (i.e. by comparing just the last 32 bits of SHA-256 hash). So there is a possibility that a user may get a false warning even when the passwords used are different (if the SHA-256 hash does not match but the last 32 bits match). We calculate the possibility of this happening below.

2.1 Collision Resistance of Long Hash

Let SHA-256' denote the new scheme which involves hashing a million times using SHA-256. We will prove by reduction, however, that this does not happen because if hashing done using SHA-256 is collision resistant then hashing done with SHA-256' is also collision resistant [2].

We proved that:

SHA-256 is CR2-KK secure => SHA-256' is CR2-KK secure

Taking contrapositive, we need to prove:

There is an efficient adversary 'A' attacking SHA-256' in CR2-KK sense => There is an efficient adversary 'B' attacking SHA-256 in CR2-KK sense.

We construct B using A.

Adversary B:

Run A.

A will output a pair of messages (m0, m1) that hash to the same value under SHA-256'.

m2 = m0 and m3 = m1

for i = 1 to 100,000:

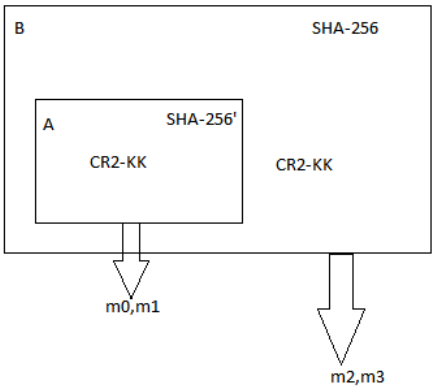
if SHA-256(m2) = SHA-256(m3):

break;

else:

m2 = SHA-256(m2) and m3 = SHA-256(m3)

return (m2, m3)



Advantage:

$$\mathrm{Adv}^{CR2-KK}_{sha-256}(B) = \Pr[\mathrm{Exp}^{CR2-KK}_{sha-256}(B)=1] = \mathrm{Adv}^{CR2-KK}_{sha-256'}(A)$$

This is because B is right whenever A is right and B simulates a perfect CR2-KK game for A. B works by the logic that if there is a collision in the one-millionth hash, there was a collision somewhere in the hash chain i.e. some intermediate values must have hashed to the same value.

Resources:

B makes no queries and A makes no queries.

*O(Time taken by B) = 100,000 * O(Time taken by A) = O(Time taken by A)*

Note: Thus, B is an efficient CR2-KK adversary.

`

2.2 Collision Resistance of Short Hash

We calculate the possibility of no full hash collision but short hash collision. This is the same as the number of trials needed by a birthday attack on the short hash to find a collision i.e. N = √2|R| = √(2 * 2^32) = 92,681 trials should suffice to get a collision with a probability close to one. So, it's easy to find two passwords whose short hash is the same but not the long hash. So, the possibility that the user gets a warning about password re-use even when he has used different passwords would be close to one if the number of entries in the password-hash table increases beyond 92,681.

III. Analysis of Dictionary attack against Appu

Appu stores hashes of passwords of users on his machine. The most common type of attacks on password hashes are dictionary attacks where an attacker has a table of possible passwords and corresponding hash values of those passwords. When given a hash value of a password, the attacker simply scans this table till he gets the hash value matching the hash value that he wants to invert and if he finds it in the table, he simply returns the corresponding password entry. The passwords Appu stores are often of important websites like Bank of America etc. So, it is important to take care that an attacker getting hold of the table where short and long hashes are stored from the user's machine, should not be able to get back the password after seeing its long hash and short hash. We prove that long hash is secure against dictionary attack but the short hash is not. So, the scheme as a whole, is not secure against dictionary attacks.

3.1 Security against Dictionary Attack

We design the following experiment to define Password recovery against chosen dictionary (PR-CD) attack of a hashing scheme S.

For an adversary A, consider the following experiment $\text{Exp}^{pr-cd}_{S}(A)$ that involves:

(1) Computing an offline table of passwords and corresponding hash value for all likely password values in a dictionary.

(2) Guessing possible passwords and verifying their correctness using a Verification Oracle (VFO). The VFO returns 1 if the guessed password is indeed the correct one else it returns 0. The VFO cannot be queried more than five times.

If the VFO returns 1 for any one of the queries, return 1, else return 0. We define advantage of adversary A as follows:

$$\text{Adv}^{pr-cd}_{S}(A) = \Pr[\text{Exp}^{pr-cd}_{S}(A)=1]$$

The advantage of the adversary in a dictionary attack is a dependent on the quality of the dictionary. The higher the quality, the more are the chances of the adversary being able to guess the password. We define **Quality of Dictionary (QoD)** to be the probability that the given dictionary actually containing user's password.

Note: We limit the number of queries to the VFO to be 5 because in our case if the adversary tries too many passwords at a website such as Bank of America, the website might detect some unusual activity and notify the user. So, it is important that the adversary guesses the correct password in the first few trials itself.

3.2 Security of Long Hash against Dictionary Attack

Let SHA-256' denote the long hash scheme. We define an adversary A who works as follows: Firstly, the adversary is given a dictionary of possible passwords and the long hash value for which he has to find the corresponding password. A first computes a HTable which has 2 columns, password and corresponding long hash value for each password in the dictionary. Then, he systematically scans this table for his challenge long hash. If he finds a match, he checks to see if the corresponding password value is correct by querying it to the VFO. If the VFO returns 1, then A returns 1 else A returns 0. [For details of proof, see Appendix II]

It is easy to see that the advantage of A is the probability that A finds a preimage of given challenge hash in the dictionary which in effect is

$$\mathrm{Adv}^{pr-cd}_{SHA-256'}(A) = \Pr\left[\mathrm{Exp}^{pr-cd}_{SHA-256'}(A) = 1\right] = \mathrm{QoD}$$

Time complexity = Time taken to generate HTable + Time taken to compare challenge hash with each entry in HTable= O(time to do one SHA-256 hash * 1,000,000 * size of dictionary) + O(Time taken to do 2^20 comparisons)

Although, this does not seem much theoretically, in practice, on a Q6600 quad-core processor, 65,139 hashes can be computed in a second [3]. So, the time taken to compute hashes for the entire dictionary which typically has 2^20 entries would be (2^20/65,139) which is 16 seconds. Now, if we hash each password in the dictionary a million times, the adversary would take (1,000,000 * 16seconds) which is 185 days to compute the same dictionary. So, although for a high quality dictionary the attack succeeds in just one query to the VFO, we conclude that the million time hash is secure against dictionary attack because too much time is taken to compute the password-hash table.

3.3 Security of Short Hash against Dictionary Attack

In the short hash case, the attacker B wins when he can find at least one and no more than five passwords (one of them is the real one) which hash to the same 32-bits value as the real password. The adversary works in the same way as the long hash adversary except this time the HTable will have entries corresponding to the last 32 bits of SHA-256 done on each dictionary password. [For details, see Appendix II]

Similar to the one-million time hash case, the advantage of B is the probability that it successfully finds the correct preimage of the 32-bit hash value (i.e. the password) in the HTable. However, as the length of hash is decreased significantly as compared to the full length SHA-256 case, we may find multiple passwords in HTable having the same short hash. [For details of proof, see Appendix III]

$$Adv^{pr-cd}_{SH}(B) = \Pr[Exp^{pr-cd}_{SH}(B) = 1] = (1 - 2^{-39}) \cdot \Pr[QoD] \blacksquare$$

3.4 Security of Short Hash combined with Long Hash against Dictionary Attack

The composite scheme is just as secure as the weaker of long hash and short hash. As we have proved that the short hash is insecure against dictionary attacks, the composite scheme is also insecure against dictionary attacks.

IV. Conclusion

We conclude that the entire point of the designer of the scheme in hashing a million times was to improve resistance to dictionary attacks. However, as he wanted to achieve some computational benefit [For details, see appendix IV], he had to store the short hash. He hoped that storing just the last 32 bits of the one-time hash would increase the number of possible passwords that an attacker might come up with and he will not be able to decide which the correct one is. We find that although the number of passwords that an attacker can compute does increase, as far as a dictionary attack is concerned, it is still not enough to stop the attacker from achieving his objective.

REFERENCES

[1] Appu: Personal Information Manager.http://appu.gtnoise.net/appu.html
[2] Encrypting (MD5) multiple times can improve security.
http://www.facebook.com/l.php?u=http%3A%2F%2Fstackoverflow.com%2Fquestions%2F6869129%2Fencryptingmd5-multiple-times-can-improve-security&h=cAQFwZQ4i
[3] How many SHA256 hashes can a modern computer compute.
http://stackoverflow.com/questions/4764026/how-many-sha256-hashes-can-a-modern-computer-compute

APPENDIX

Appendix I.

$\mathbf{Exp}_{SHA-256'}^{pr-cd}(A)$

pwd $\xleftarrow{\$}$ {All possible passwords}

y $\xleftarrow{\$}$ SHA-256'(pwd)

```
Adversary A(pwd, y, D):
    Step 1: Compute the hash table
    index = 0
    for pwd' in D:
        HTable[index][0] = pwd'
        HTable[index][1] = SHA-256'(pwd')
        index ++;
    Step 2: Look for y in pre-computed hash table
    computed_pwd = ''
    for entry in HTable:
        if(HTable[entry][1] == y)
            computed_pwd = HTable[entry][0]
            break;
    Step 3: Verify the correctness of password from Verification oracle
    if(computed_pwd) and VFO(computed_pwd):
        return 1;
    else:
        return 0;
```

Appendix II.

$\mathbf{Exp}_{SH}^{pr-cd}(B)$

pwd $\xleftarrow{\$}$ {All possible passwords}

y $\xleftarrow{\$}$ SH(pwd)

```
Adversary B(pwd, y, D):
    Step 1:Compute the hash table
    index = 0
    for pwd' in D:
        HTable[index][0] = pwd'
        HTable[index][1] = SH(pwd')
        index ++;
    Step 2: Look for y in pre-computed hash table
    For entry in HTable
        if(HTable[entry][1] == y)
            list-pos-pwd.append(HTable[entry][0] )
    Step 3: Verify the correctness of password from Verification oracle
    trials = len(list-pos-pwd) OR 5 (whichever is less)
    For each entry e in list-pos-pwd:
        If trials > 0:
            Flag = VFO(e)
                If flag =1
                    Return pwd;
                    Break;
                Trials--;
            Return 0;
```

Appendix III.

Assumption -

$$Adv_{SH}^{pr-cd}(B) = \Pr[Exp_{SH}^{pr-cd}(B) = 1]$$

= Pr[At least one and no more than 5 passwords hash to the same short hash value]

= Pr[no more than 4 different passwords hash to same value ∩ corrent password is in the dictionary]

$$= \Pr[Case_0 \cup Case_1 \cup Case_2 \cup Case_3 \cup Case_4 \mid QoD] \cdot \Pr[QoD]$$

$$= \sum_{i=0}^{4} \Pr[Case_i \mid QoD] \cdot \Pr[QoD] \blacksquare$$

In which,

$Case_i$ - implies the case that exactly *i* password(s) will hash to the same value as the real password.

$$\Pr[Case_i \mid QoD] = \binom{2^{20}}{i} \cdot (\frac{1}{2^{32}})^i \cdot (1 - \frac{1}{2^{32}})^{(2^{20}-i)}$$

After calculation, $\Pr[Exp_{SH}^{pr-cd}(B) = 1] = (1 - 2^{-39}) \cdot \Pr[QoD]$

Appendix IV.

Appu needs to warn user about password reuse. When a new entry is made for a password in the table, it simultaneously computes long hash and short hash. Computing the long hash takes about 6 minutes, as Appu wants to give user a quick warning about password reuse, it does not wait for the user's machine to finish computing the long hash. So it quickly computes the short hash, and compares it with short hash entries of all passwords in the table, if it finds the same value somewhere in the table, it concludes that the password has been used previously. So, as just the last 32 bits are taken into account, it's possible that two different passwords may have the same short hash value and the user may get a false warning.