hw02

Daven Farnham

February 2017

1. Public Key Encryption

   (a) It was said in a piazza post that Trudy knows a complete list of the buildings Alice and Bob might meet at. If this is the case, and since Trudy knows the form of the messages, she can simply use the public keys she has and encrypt all possible messages then compare her encrypted messages to those sent by Alice and Bob. Once she finds a match, she'll know which plaintext she used to form that particular encrypted message.

   So, for example, she can encrypt 'YES' and 'NO' using both Alice and Bob's encrypted keys (since this is a common message sent between the two). Then, whenever she sees an encrypted message she can compare it with the ones she encrypted and determine which plaintext resulted in a match. This can be expanded to include all buildings and possible times s.t. Trudy will have a complete dictionary (of the form of message sent between Alice and Bob) mapping plaintexts to encryptions and can just query the table to determine the plaintexts of the messages sent between the two.

   (b) A very simple modification would be to vary the form of the message sent between Alice and Bob. For example, in the two example messages sent by Bob he uses 11 characters. If you didn't know the form of the message, assuming Bob uses only alphanumeric characters and common punctuation, you're looking at roughly 66^{11} possible combinations. If the message could be even longer, the feasibility of compiling a list of all possible plaintexts decreases substantially as long as the form of the message isn't predetermined.

2. Passwords

   (a) So let's compute this piece by piece. If the size of the table is 'm' and the time to hash is 'h', it'll take $(h * m)$ time to precompute the table. Once you have the table, you need to search it. If you search it 't' times with each search taking a constant amount of time and a success proportion in terms of 't' as 's', then $(t * s)$ will give the number of successes:

$$t + (h * m) : \text{equals the total amount of work}$$

$$(t * s) : \text{number of successes}$$

$$\frac{t+(h*m)}{(t*s)}$$

As t increases, you'll see that it dominates. (h * m) as a constant will effectively go to zero as t increases indefinitely s.t. you end up with:

$$\frac{t}{(t*s)} = \frac{1}{s}$$

(b) I think the difference here is that since each password has a user salt, you'd need to compute the table for each user. However, some of the passwords you're trying to crack will be in the table while others won't.

$$(t * s) * \frac{h*m}{2} : \text{work to find password for hash in table}$$

$$(t * (1 - s)) * (h * m) : \text{work to find password for hash not in table}$$

$$\frac{(t*s)*\frac{h*m}{2}+(t*(1-s))*(h*m)}{ts}$$

't' cancels out from this equation simply leaving:

$$\frac{s*\frac{h*m}{2}+(1-s)*(h*m)}{s}$$

3. (a) I think method (1) would actually be harder to crack. So similar to above, let's assume an attacker wants to construct a rainbow table. Assuming passwords are alphanumeric (upper and lowercase and numbers) that means that each character can be up 62 different values. If a password is of length 9 then this produces 62^9 different possible values. With an included salt, however, an attacker would need to construct tables for each possible salt. Just to illustrate, even if the salt could only be 4 possible values, an attacker would now need to hash $4 * 62^9$ values to successfully crack the password system.

In the case where the user needs to input 2 passwords each of length 8, again, assuming similar numbers of possible characters this means there are roughly 62^8 possible values to attempt to hash. If the salts are different but of the same length, then taking the example from above of a small salt of only 4 possible values, a full table would now require $(4 * 62^8 + 4 * 62^8)$ total hashes, which is still less than the system in (1).

I think the point that shines through here is that the 62^9 term is the dominant factor regardless of the additional salting or the fact you need to hash 62^8 passwords twice.

4. (a) I think it's not the case since passwords chosen by humans aren't going to be nearly as random as passwords chosen by a computer. More clearly, even if a computer doesn't have a completely random algorithm for determining a value $[0, 2^{60})$, the distribution of the numbers within that range will be much more evenly distributed than for a human trying to pick an alphanumeric password.

   Humans like to pick words as passwords, which would immediately rule out passwords of random characters. Furthermore, certain words are more common than others. These words are much more likely to occur as passwords than others, making it easier to crack these kinds of passwords with something like a dictionary attack or a hybrid dictionary attack where you might add on additional numbers to words before trying them as inputs.

   Overall, the inability of humans to come anywhere close to a random distribution between the 62^{10} possible alphanumeric characters in a password is why these passwords aren't nearly as secure as a computer generated key of 2^{60} possible values.

5. (a) I think the biggest problem comes from the fact that since JS is client side, you can essentially edit or disable this code. So, for example, you could disable or edit the JS running on your client to not check the validity of the inputs or send incorrect information to the website, rendering any checks on a website useless.