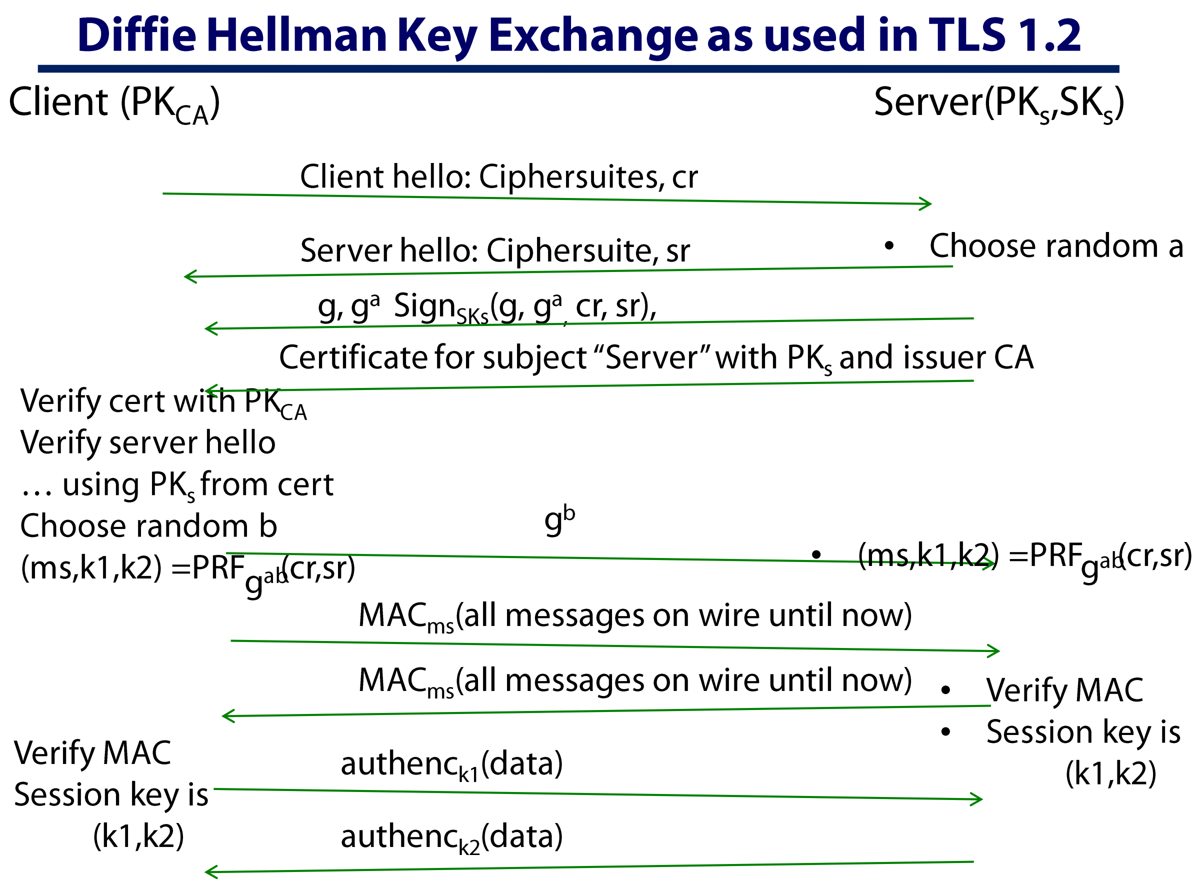
Practice Problem Set 3: Applied Crypto Potpourri

February 28, 2017

1 TLS

Exercise 1. The following figure shows Diffie Hellman Key Exchange for one-sided authentication as used in TLS 1.2 that we discussed in class today. This protocol is used to encrypt much of the web's communications. The client is typically a web brower and the server is typical a webserver, e.g. Facebook's server. The session keys $(k1, k2)$ and secret material a, b and g^{ab} is deleted by the client and server at the end of the session.



1. The protocol shows in the figure supports "one sided authentication". That is, the client knows that she is talking to the server. However, the server does not know which client she is

talking to. To make this point crystal clear, consider a man-in-the-middle attacker that wishes to impersonate the client to the server. (That is, the man-in-the-middle attacker sits between the client and server. Write down the protocol that attacker would use to impersonate the client to server (i.e. convince the server that the server is talking to the client, instead of to the attacker).

2. Now consider a man-in-the-middle attacker that sits between the client and server, and wishes to impersonate the server to the client. Assuming that the client knows the correct public key for the CA. Why does the attacker fail to impersonate the server to the client?

3. Consider a flawed TLS implementation where the client "forgets" to check the signature on the server's certificate. Write down exactly how a man-in-the-middle attacker that intercepts the communications between client and server can establish one pair of session keys $(k1, k2)$ between itself and the client, and another pair of session keys $(k1', k2')$ between itself and the server. Explain why, by doing this, the attacker can silently intercept, read, and pass on any data sent from client to server, and vice versa, without the client or server ever realizing that their communications have been read.

4. Consider a man-in-the-middle attacker that steals the secret key of the CA, SK_{CA}. Explain why, by doing this, the attacker can silently intercept, read, and pass on any data sent from client to server, and vice versa, without the client or server ever realizing that their communications have been read. (Hint, once again the attacker establishes one pair of session keys $(k1, k2)$ between itself and the client, and another pair of session keys $(k1', k2')$ between itself and the server. Write down exactly how it does this.)

5. If this attacker becomes a man-in-the-middle for another client and the same server $Server$, can it carry out the same attack?

6. If this attacker becomes a man-in-the-middle for another client and a different server $Server2$, can it carry out the same attack?

7. Now consider a passive attacker that has collected all the communication sent between the client and server that have been done *in the past*. Suppose this passive attacker has now stolen the secret key of the CA, SK_{CA}. Can it use this secret key to decrypt the past communications that it has collected?

8. Now consider a passive attacker that has collected all the communication sent between the client and server *in the past*. Suppose this passive attacker has now stolen the secret key of the server, SK_S. Can it use this secret key to decrypt the past communications that it has collected?

9. If you were an attacker, which key would you most want to steal? Your choices are $SK_{CA}, SK_S, ms, k1, k2$. Justify your response.

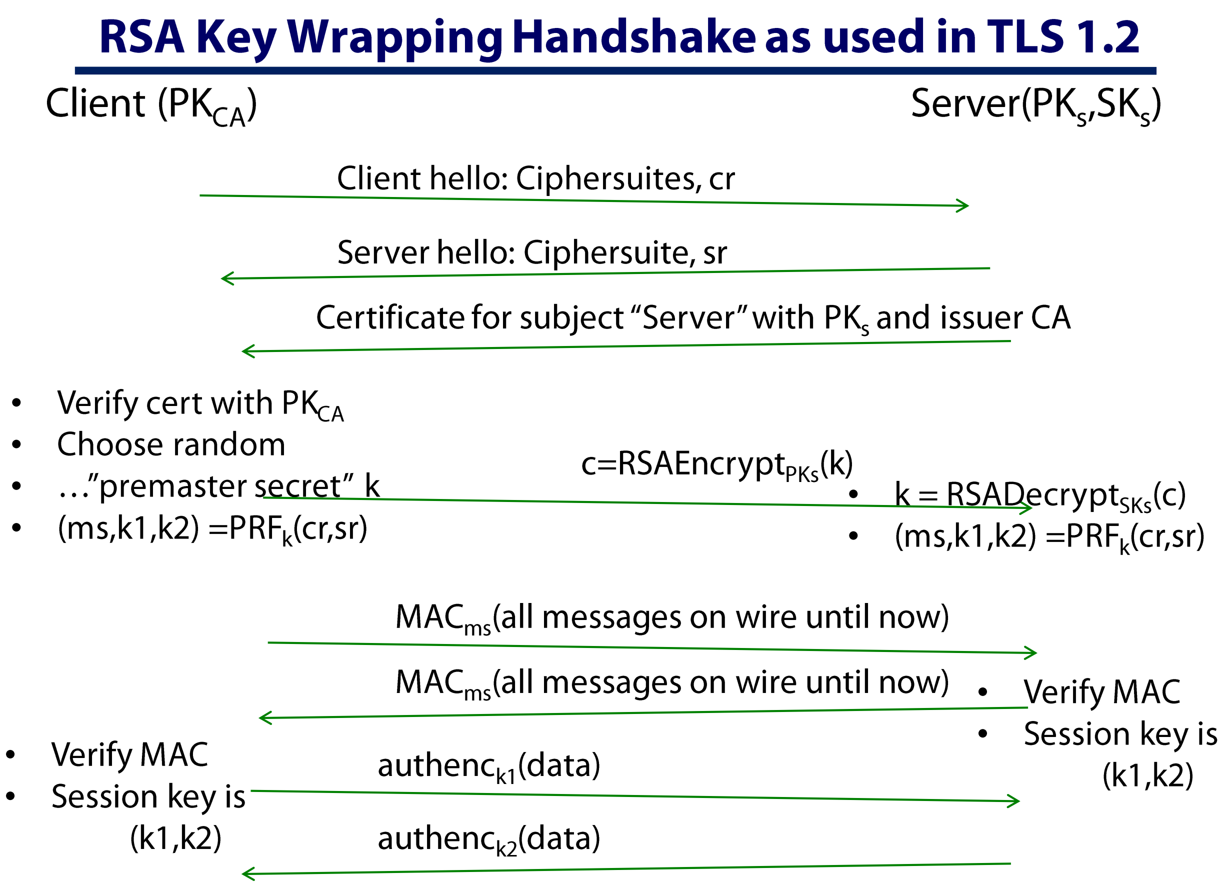
10. Consider the Diginotar incident that was in the news in 2012. (Look on the Internet!) Explain which of the above keys were stolen in the attack on Diginotar. Also explain how these keys were used, by the attacker, to impersonate Google to users in Iran.

11. The SuperFish malvertising software was shipped as part of Lenovo laptops in 2015. Researchers discovered that the Superfish software modified the laptop's browser to so that the SuperFish Public Key was installed as trusted CA public key. The SuperFish software also

made itself a man-in-the-middle between the browser and the laptop's network connection. Explain exactly how this allowed SuperFish to decrypt any communication sent from the user to any webserver, without the user knowing, and modify this communication to inject SuperFish ads. (Hint, consider how SuperFish might forge a certificate for the Server.)

12. Here is an alternative key exchange mechanism: RSA-Key-Wrapping as the mechanism for Key Exchange with one-sided authentication as used in TLS 1.2. This mechanism is generally considered outdated today. This is because, unlike Diffie-Helman Key Exchange, it does not provide *forward secrecy.*

    Specifically, this mechanism fails to prevent the following attack, that is prevented by the Diffie Hellman Key exchange. Consider a passive attacker that has collected all the communication sent between the client and server that have been done *in the past*. Suppose this passive attacker has now stolen the secret key of the Server, SK_s. Show how it can use this secret key to decrypt the past communications that it has collected.



13. (Optional.) Repeat all the questions above for the RSA-based key exchange.

2 Password Hashing

Exercise 2. Consider the following tweet, which quotes from the user manual of LastPass, a password manager.

Why does the author consider this to be "crazy talk"? To answer this question,

1. Explain what a password manager is. You can look online for sources. Why is a password manager useful?
2. Describe how an attacker might be able to break the security of the password manager, given what was tweeted above. That is, explain how the attacker learns all of the user's passwords.
3. Make sure to clearly describe your threat model – that is, explain exactly what sort of access the attacker has to user's Andriod device, and what sort of cryptanalysis capabilities she might have.

Exercise 3. Password cracking. Suppose you are in charge of security for a major web site, and you are considering what would happen if an attacker stole your database of usernames and passwords. You have already implemented a basic defense: instead of storing the plaintext passwords, you store their SHA-256 hashes [1].

Part A:

[1] You shouldn't actually use raw SHA-256 for this task, in actual practice you should use a library designed

Your threat model assumes that the attacker can carry out 4 million SHA-256 hashes per second. His goal is to recover as many plaintext passwords as possible from the information in the stolen database. Valid passwords for your site may contain only characters a–z, A–Z, and 0–9, and are exactly 8 characters long. For the purposes of this homework, assume that each user selects a random password.

1. Given the hash of a single password, how many hours would it take for the attacker to crack a single password by brute force, on average?

2. How large a botnet would he need to crack individual hashes at an average rate of one per hour, assuming each bot can compute 4 million hashes per second?

Part B:

Based on your answer to part (a), the attacker would probably want to adopt more sophisticated techniques. You consider whether he could compute the SHA-256 hash of every valid password and create a table of (*hash*, *password*) pairs sorted by hash. With this table, he would be able to take a hash and find the corresponding password very quickly.

1. How many bytes would the table occupy?

Part C:

It appears that the attacker probably won't have enough disk space to store the exhaustive table from part (b). You consider another possibility: he could use a *rainbow table*, a space-efficient data structure for storing precomputed hash values.

A rainbow table is computed with respect to a specific set of N passwords and a hash function H (in this case, SHA-256). We construct a table by computing m *chains*, each of fixed length k and representing k passwords and their hashes. The table is constructed in such a way that only the first and last passwords in each chain need to be stored: the last password (or *endpoint*) is sufficient to recognize whether a hash value is likely to be part of the chain, and the first password is sufficient to reconstruct the rest of the chain. When long chains are used, this arrangement saves an enormous amount of space at the cost of some additional computation.

Chains are constructed using a family of *reduction functions* $R_1, R_2, \ldots, R_k$ that deterministically but pseudorandomly map every possible hash value to one of the N passwords. (We can think of each R_i as a PRF keyed with a key that the attacker chose uniformly and independently at random; that is, the key is known to the attacker.) Each chain begins with a different password p_0. To extend the chain by one step, we compute $h_i := H(p_{i-1})$ then apply the ith reduction function to arrive at the next password, $p_i = R_i(h_i)$. Thus, a chain of length 3 starting with the password `hax0r123` would consist of

$$(\,\texttt{hax0r123},\ R_1(H(\texttt{hax0r123})),\ R_2(H(R_1(H(\texttt{hax0r123}))))\,)$$

After building the table, we can use it to quickly find a password p_* that hashes to a particular value h_*. The first step is to locate a chain containing h_* in the table; this requires, at most, about $k^2/2$ hash operations. Since h_* could fall in any of $k-1$ positions in a chain, we compute the password that would *end up* in the final chain position for each case. If we start by assuming h_* is right before the end of the chain and work backwards, the possible endpoints will be

specifically for password hashing that uses a function such as scrypt or bcrypt (see `http://yorickpeterse.com/articles/use-bcrypt-fool/`). Today, GPU-based hashing is so fast that an attacker can often just compute hashes on the fly. See the link for more details.

$R_k(h_*)$, $R_k(H(R_{k-1}(h_*)))$, $\ldots$. We then check if any of these values is the endpoint of a chain in the table. If we find a matching endpoint, we proceed to the second step, reconstructing this chain based on its initial value. This chain is very likely to contain a password that hashes to h, though collisions in the reduction functions cause occasional false positives.

[You can read more about rainbow tables here `http://en.wikipedia.org/wiki/Rainbow_table`]

1. For simplicity, make the optimistic assumption that the attacker's rainbow table contains no collisions and each valid password is represented exactly once. Assuming each password occupies 8 bytes, give an equation for the number of bytes in the table in terms of the chain length k and the size of the password set N.
2. If $k = 5000$, how many bytes will the attacker's table occupy to represent the same passwords as in (c)?
3. Roughly how long would it take to construct the table if the attacker can add 2 million chain elements per second?
4. Compare these size and time estimates to your results from (a), (b), and (c).

Part D:

You consider making the following change to the site: instead of storing SHA-256(*password*) it will store SHA-256(*server_secret* || *password*), where *server_secret* is a randomly generated 32-bit secret stored on the server. (The same secret is used for all passwords.)

1. How does this design partially defend against rainbow table attacks?
2. Briefly, how could you adjust the design to provide even stronger protection? (Your answer should be no more than 3 sentences long.)