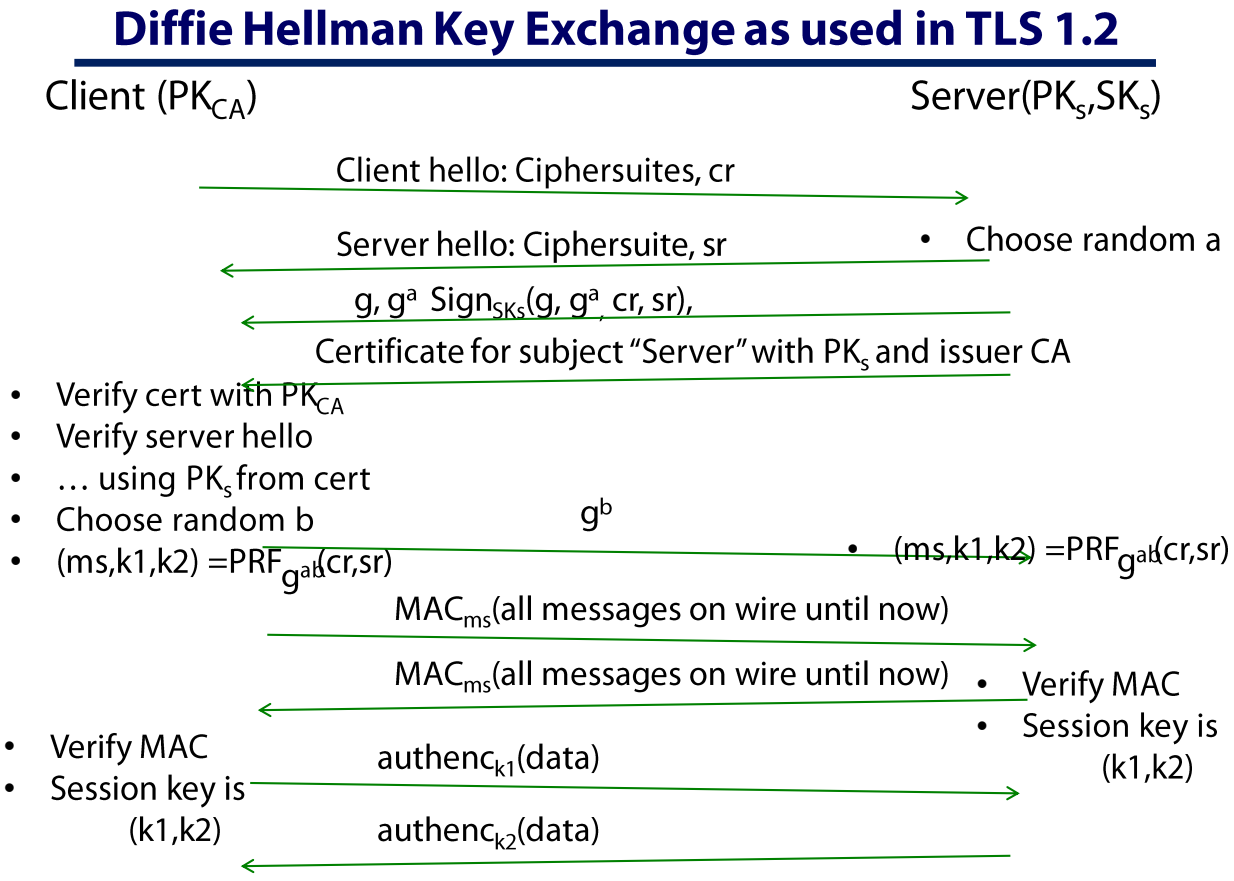
Practice Problem Set 3: Applied Crypto Potpourri Solutions

March 19, 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).

Solution: The attacker just does exactly what the client does in this picture.

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?

Solution: because the attacker cannot forge the server's signature, it has to use g^a as chosen by the server. but by the hardness of descrete logs, the attacker cannot obtain a from g^a, and thus cannot compute g^{ab} and thus cannot learn the master secret ms and thus cannot provide a correct MAC. so the client will learn that something is wrong.

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.

Solution: Attacker generates random x, y and sends g, g^x to Client and g, g^y to the server. Now the attacker has two keys g^{xb} which is shared with the Client and g^{ay} which is shared with the Server. Attacker then generates two session keys $(m_s, k_1, k_2) = PRF_{g^{xb}}(cr, sr)$ for the client and $(m_s, k_1^{'}, k_2^{'}) = PRF_{g^{ay}}(cr, sr)$ for the server.

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.)

Solution: Attacker can do the same attack as in the last part. However, when talking to the client, the attacker needs to know how to produce a signature on g, g^x so that the client will accept these values. However, the attacker does not know the server's secret keys. So, the attacker will choose a new key pair $(PK*, SK*)$ for the signature scheme, and then use $SK*$ to sign (g, g^x, cr, sr). Now, the attacker needs to convince the client that $PK*$ is really the public key of the server. Because the attacker has the CA's secret key, he is able to sign certificates with the SK_{CA} such that it can be successfully verified by the PK_{CA} in the browser's root store. So, the attacker uses SK_{CA} to sign a certificate that says that $PK*$ is the server's public key. Done!

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?

Solution: Yes

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?

Solution: Yes; the only difference is that the certificate will be for $PK*$ and the $Server2$.

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?

Solution: no, because knowledge of SK_{CA} does not allow the attacker to determine a or b that were used in the previous sessions. These are ephemeral values that are deleted.

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?

Solution: no, because knowledge of SK_{CA} does not allow the attacker to determine a or b that were used in the previous sessions

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.

Solution: SK_{CA} because I could use it to issue forged certificates and then impersonate any server on the internet.

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.

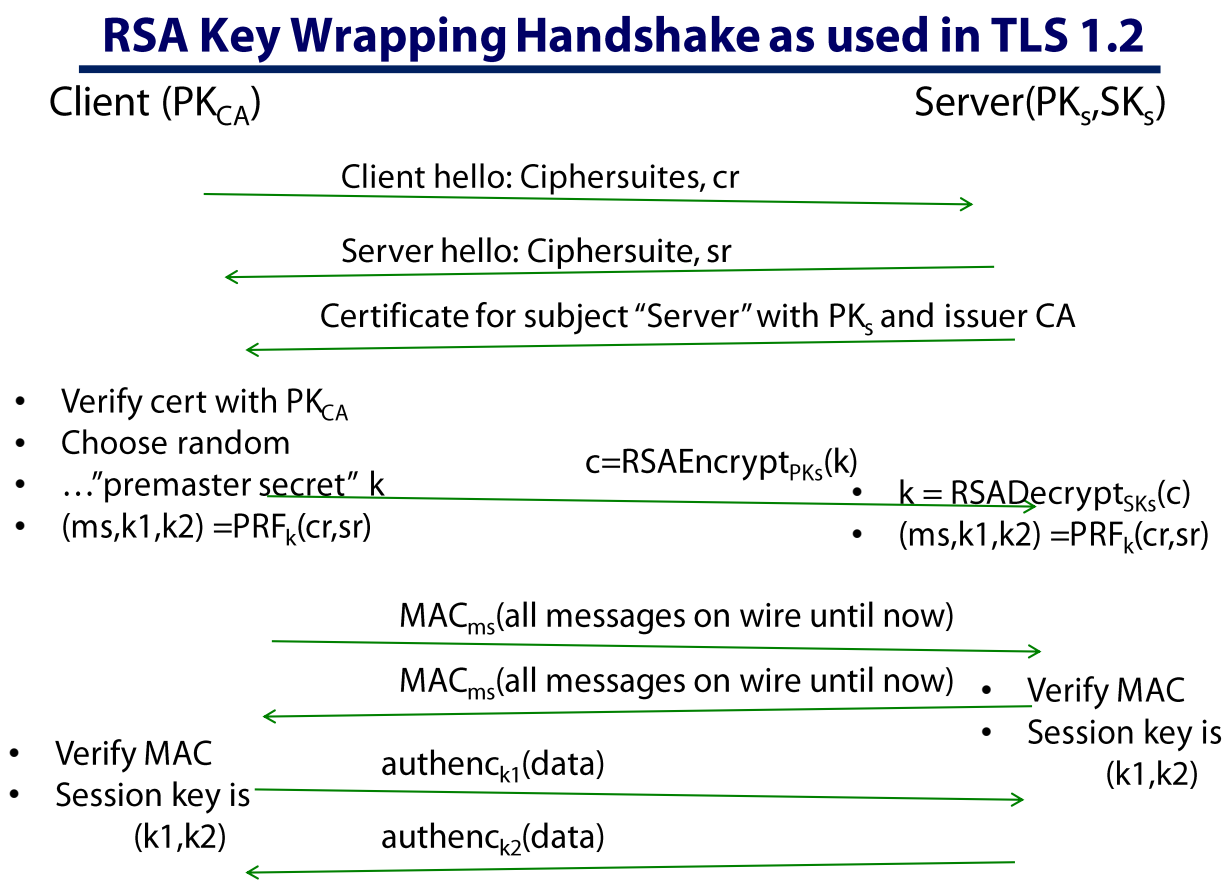
    Solution: Diginotar used to be a trusted CA, whose key was trusted by browsers. The Diginotar CA key was stolen. Certificate was created claiming that the attacker's public key $PK*$ was the public key of google.com. When users went to google.com they saw the familiar https lock. The crafted certificate is accepted by user's browser because it's signed by diginotar.

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.)

    Solution: Superfish software had access to laptops before they shipped, and so it placed it own superfish public key as a trusted CA public key in the browser root store. The superfish secret key was hardcoded into the superfish software. When the user requests a website, SuperFish gives them a fake certificate and signs it with its superfish secret key, and uses this as we have seen in the "stolen CA key" attack to intercept the encrypted communications. The certificate validates because the superfish public key is trusted by the user's browser. SuperFish now can decrypt traffic sent to them (they're on-path) and re-encrypt it and send it on, then similarly decrypt traffic they receive insert ads and re-encrypt it.

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?

   Solution: Password managers store all your passwords for convenience, they often encrypt passwords with one "root" password so that users only have to remember one password.

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.

   Solution: First of all a 4 digit pin only has 10^4 combinations or $10,000$ combinations. This can easily be brute forced. Secondly the key to decrypt the credentials is store as plaintext in the source. This means you can decrypt the master key/pin and then decrypt 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.

Solution: The attacker will need access to the encrypted passwords, the encrypted credentials and the source code. They need to have physical access to the device, or to have hacked into the device somehow.

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:

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?

   Solution: There are $n = (26 + 26 + 10)^8 = 218,340,105,584,896$ possible passwords. Since we assume passwords are randomly selected, an unknown password has an equal probability of having any of the n values, and the expected number of guesses in a brute-force attack is $n/2$. Therefore, the attack will take: $t = n/2(4\text{million})\text{s} = 27,292,513\text{s} = 7581\text{hours}$.

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?

   Solution: Since it takes an average of 7581 hours to crack a single hash, the attacker needs a 7581 node botnet to crack one hash per hour on average.

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?

   Solution: The table would have n entries (see above), each occupying 8 bytes for the password plus 32 bytes for the hash. Thus, the table would occupy $n(40\text{bytes}) = 8,733,604,223,395,840\text{bytes} = 7.76\text{pebibytes}$.

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

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 $R_k(h_*)$, $R_k(H(R_{k-1}(h_*)))$, ... 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.

   Solution: Each chain has a start and an end, for 16 bytes. The table would occupy $16\lceil N/k \rceil$ bytes.

2. If $k = 5000$, how many bytes will the attacker's table occupy to represent the same passwords as in (c)?

   Solution: $16\lceil n/5000 \rceil = 698,688,337,872\text{bytes} = 651\text{GiB}$.

3. Roughly how long would it take to construct the table if the attacker can add 2 million chain elements per second?

> **Solution:** This is similar to (a), but the attacker needs to compute the full table and takes twice as long per password. Thus, it takes $4 \cdot 7581$hours $= 30,324(30,325 without rounding error)$hours to compute the full table.

4. Compare these size and time estimates to your results from (a), (b), and (c).

> **Solution:** It takes 4 times as long (or as big a botnet) to build the table using rainbow tables, but it occupies only about 1/12500 as much space.

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?

> **Solution:** Instead of using a standard precomputed table that could work against any site, the attacker needs to compute a separate table for each site.

2. Briefly, how could you adjust the design to provide even stronger protection? (Your answer should be no more than 3 sentences long.)

> **Solution:** Strategy A: Use a per-user salt rather than a per-server secret value.
> Strategy B: Slow down the computation, say by using many iterations of the hash.