Week of March 14, 2016

Question 1  Password Hashing  (10 min)
When storing a password $p$ for user $u$, a website randomly generates a string (called a salt) $s$, and saves the tuple $(u, s, r = H(p||s))$, where $H$ is a cryptographic hash function.

(a) Say user $u$ tries to log in submitting a password $p'$ (which may or may not be the same as $p$). How does the site check if the user should be allowed to log in?

(b) Why use a hash function $H$ rather than just store $(u, p)$? Isn’t that just so much simpler? Keep it simple stupid, right?

(c) What is the purpose of the salt $s$?

(d) Suppose the site has three candidate hash function to choose from, $H_1$, $H_2$, and $H_3$. They each satisfy the following properties as displayed in the table.

<table>
<thead>
<tr>
<th>Functions</th>
<th>One-Way</th>
<th>Second Pre-Image Resistant</th>
<th>Collision Resistant</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_1$</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$H_2$</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$H_3$</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Which of these hash functions are suitable choices for the website’s password hashing scheme, in that they provide a significant gain in security over just storing passwords?

Solution:

(a) The site finds the tuple with user $u$, and compute $H(p'||s)$ using the $s$ in the user’s tuple. If this hash output is equivalent to the stored $r$ value, then the user submitted a valid password and is allowed to log into the account.

(b) If the hash function is secure (discussed further in part (d)), then even if an attack is able to hack into the site and obtain all user tuples, it still doesn’t know their passwords and cannot log into their accounts.

(c) One attack on hashed password uses dictionary attacks, where an attacker builds a table mapping common passwords to their computed hashes. Then if the attack get access to the user tuples, this table can be used to map back many hashes to their passwords.

By inject enough randomness for each password, it is no longer possible to effectively do dictionary attacks, since even common passwords will map to
many different hash values.

(d) All of them actually are secure, since one-way is the main property needed. Essentially, if the attacker gets a hold of a user’s hash value, it needs to find a pre-image to be able to impersonate that user. This is prevented by the hash functions’ one-wayness.

Second pre-image resistance’s definition is based on already knowing one password, and finding another that hashes to the same output. This is not an issue that is a threat to the website’s password validation process. Similarly, collision resistance involves finding any two passwords that hash to the same output, again an unnecessary property.
**Question 2  PKI and TLS**

Your web browser’s traffic is secured when you use HTTPS, which relies on the TLS protocol. In this problem, we will construct the basic framework of TLS (though the actual TLS protocol is more complex and detailed).

Our browser B wants to exchange messages with Apache web server A. The Apache server A has an RSA key pair \((PK, SK)\). Whenever a browser B initiates communication with A, A first sends a certificate essentially saying “I’m server A and my public key is PK”.

(a) How can B be sure that the certificate it supposedly got is really from A? Describe what is done by the browser to authenticate the certificate. Whom is the browser trusting?

(b) Why is it okay for B to get the certificate directly from A, rather than querying a certificate authority (CA)? What are some advantages of getting the certificate directly?

(c) Now B has authenticated the certificate, so it knows A’s public key. RSA is slower than symmetric-key crypto, so ideally we want to use AES instead of RSA to encrypt messages. B generates a random AES key \(k\). Describe a simple protocol such that A and only A obtains the true value of \(k\), and both B and A know they each have the same \(k\).

(d) Oh noes! The server admin has made a terrible mistake and accidentally leaked \(SK\), the RSA secret key. Evil Eve, who happens to be between B and A in the communication path, has gotten her hands on a copy of \(SK\). What sort of badness could she do from that point onward?

(e) Eve, always a data squirrel, has also been recording the encrypted traffic between B and A even before the secret key leak. Now that she has the secret key, she’s glad she did all that work to save the old traffic data. How is she able to access the encrypted data?

(f) How might our protocol be changed to protect against this attack on recorded old traffic?

(g) What should A do now that its secret key has been leaked?

**Solution:**

(a) B relies on the certificate chain of trust. The browser comes with a few trusted root certificates, which contain a public key for the associated trusted organizations (called certificate authorities (CA)). Each CA can sign other certificates for other organization, which serve as intermediate CAs. This can happen recursively for several levels, down to A’s certificate. Thus, A’s certificate will also contain a signature, created with the keys of a CA.
To authenticate A’s certificate, B validates the certificate’s signature with the public key of the signing CA (obtained from the CA’s certificate). The CA’s certificate can be recursively authenticated likewise up a certificate signed by a root CA, which B inherently trust.

The browser is trusting each CA in this chain of trust to provide signatures only for the correct parties (e.g. a certificate saying “I’m server A with public key PK” is only signed by a CA if it is really the server A and A indeed has the key pair with public key PK). Also, the CA needs to keep its secret signing key secure!

If a CA is compromised or its secret key is leaked, or issues rogue/erroneous certificates, then all bets are off! The number of CAs in the PKI system makes this a very real threat. This has happened in several notable cases (DigiNotar and Comodo in 2011, Turktrust in 2012, Symantec and CNNIC in 2015, to name just a few...).

(b) The certificate chain is built on signatures using the private keys of signing CAs, which we can check with the CAs’ public keys. These signatures allow us to validate that the entire chain is correct as indicated by the CAs, so it is fine to get the certificate directly from B rather than a CA.

Getting the certificate directly from B helps avoid a potential privacy concern. If we had to always query the CAs to get a site’s certificate, the CA is able to observe what sites are being visited. Additionally, the protocol for establishing a secure communication channel between B and A is simplified and can be more performant (due to fewer message round-trips) by excluding an additional party (the CA).

(c) B sends A the encryption of k using PK, e.g. EncPK(k). Since only A has the associated secret key SK, only A can access k correctly. For A to demonstrate that it obtained the correct key k, it can send back the encryption of some value that B expects, such that B can decrypt with k and confirm that the plaintext value was as expected. B can likewise demonstrate it has the same key by echoing back a random nonce chosen by A. So the protocol could look as follows:

- B → A : RSA.EncPK((k, NB)) for random nonce NB.
- A → B : AES.Enck((NB, NA)) for random nonce NA.
- B → A : AES.Enck(NA)

(d) Since Eve is an on-path attacker, she can see all messages sent between B and A. Since Eve also has the secret key, she can decrypt the message containing the shared AES key, and use that same key to decrypt all further traffic. So Eve can snoop entirely on the traffic. What’s more, Eve can also modify the messages. Note that even if the messages have MACs or signatures attached (as
they are in TLS), since the secret key and session key are revealed, these can be recomputed by Eve to match her modified messages.

(e) Since Eve has all of the old data, she can do the same attack as in the previous part to uncover the session key for each session, revealing all the messages Eve has squirreled away.

(f) Recall that the Diffie-Hellman (DH) key exchange allows two parties to establish a shared key without revealing the key in any exchanged messages. So instead of sending the encryption of $k$ directly, we can send DH key exchange messages, with $A$ signing its messages. $B$ and $A$ can confirm they each have the same key using nonces or timestamps, as before. The protocol could look as follows:

- Let the public DH parameters be generator $g$ and prime $p$. $B$ and $A$ each generate their secret DH key, $b$ and $a$, respectively.
- $B \rightarrow A : RSA.Enc_{PK}((DH_B = g^b \mod p, N_B))$ for random nonce $N_B$.
- $A \rightarrow B : A$ computes $k = DH_B^a \mod p$, and sends $AES.Enc_k((DH_A = g^a \mod p, N_B, N_A)), RSA.Sign_{SK}((DH_A, N_B, N_A))$ for random nonce $N_A$.
- $B \rightarrow A : A$ computes $k = DH_A^b \mod p$ and sends $AES.Enc_k(N_A)$

Note that though Diffie-Hellman itself is not secure against MITM, the use of signatures on the server’s messages will ensure that an attacker cannot tamper with $A$’s half of the key exchange. If the attacker tampers with $B$’s messages (necessary otherwise the attacker will not have any control over the session key), $B$ and $A$ will end up with different keys and the attacker will not be able to convince $B$ of otherwise. So $B$ will terminate the connection, preventing an insecure communication channel from $B$ to $A$.

Use of Diffie-Hellman in TLS provides a property we call perfect forward secrecy, as the session key is safe in the future even if the secret key is revealed and previous messages were recorded.

(g) $A$ should generate a new RSA key pair, and revoke its old certificate, since it is no longer valid to use the old keys associated with the old certificate. This revocation is done with the CA who originally issued $A$’s certificate, using some CA-specific out-of-band scheme.

Clients are made aware of the revocation using either the Online Certificate Status Protocol (OCSP) or looking up the certificate in a Certificate Revocation List (CRL). OCSP lets a browser query an issuing CA about the status of an individual certificate, whereas the CRLs are lists periodically received by the browser containing revoked certificates.

Note though that OCSP requires additional network communication when validating a certificate, and reveals the site being visited to the CA, a potential privacy issue. On the other hand, CRLs can be large, requiring space and...
bandwidth to download, and newly revoked certificates may not be immediately received by all clients.
Question 3  Crypto Protocol Errors

Alice (A) and Bob (B) want a secure communication channel between them, and will rely on a trusted server (S). Assume the following notation:

- A and B are the identities of Alice and Bob respectively.
- $K_{AS}$ is a symmetric key only known to A and S.
- $K_{BS}$ is a symmetric key only known to B and S.
- $N_A$ and $N_B$ are random nonces generated by A and B, respectively.
- $K_{AB}$ is a symmetric key generated for this communication session between A and B.

They use the following protocol (known as the Needham-Schroeder protocol):

- $A \rightarrow S : A, B, N_A$
- $S \rightarrow A : Enc_{K_{AS}}(N_A, K_{AB}, B, Enc_{K_{BS}}(K_{AB}, A))$
- $A \rightarrow B : Enc_{K_{BS}}(K_{AB}, A)$
- $B \rightarrow A : Enc_{K_{AB}}(N_B)$
- $A \rightarrow B : Enc_{K_{AB}}(N_B - 1)$

(a) Eve has been monitoring the Needham-Schroeder protocol messages exchanged between Alice and Bob. One day Eve uncovers an old session key (an old $K_{AB}$ value) from a previous protocol exchange. How can Eve leverage this old key?

(b) How do you fix the protocol vulnerability?

Solution:

(a) Eve can replay the message $Enc_{K_{BS}}(K'_{AB}, A)$ to Bob, who will accept it without being able to tell the key is not fresh. Because Eve knows the old session key $K'_{AB}$, she can successfully complete the rest of the protocol with Bob. Now Bob will start talking to Eve thinking he is talking securely to Alice. This is a replay attack. Note that Needham-Schroeder was a real protocol, and the vulnerability was discovered several years later. Often cryptographic protocol errors are quite subtle.

(b) The solution is to ensure the freshness of the session key using timestamps or nonces. At the beginning of the protocol:

- $A \rightarrow B : A$
- $B \rightarrow A : Enc_{K_{BS}}(A, N'_B)$
- $A \rightarrow S : A, B, N_A, Enc_{K_{BS}}(A, N'_B)$
- $S \rightarrow A : Enc_{K_{AS}}(N_A, K_{AB}, B, Enc_{K_{BS}}(K_{AB}, A, N'_B))$
Then the protocol completes as before. By including the fresh nonce $N'_B$, Bob can ensure that the message he is receiving from Alice includes a session key freshly generated by $S$. 