NETWORK SECURITY HW4

GUR TELEM 206631848

1. RSA/MAC/Signatures

1.1. **RSA.**

1.1.1. *Math Incorrect. (Secured - no one will ever know the message except A).* d is part of the private key of A and not known to B.

Unless securely passed to B, it would simply be impossible for B to decrypt the message, if it was passed to B in an insecure way (for example, without being encrypted using a pubkey of B), it would be possible for B to decrypt but other people would be able to decrypt too as they would have seen d passed plainly to B (from A).

1.1.2. *Math&Sec Correct.* (e, n) is the pubkey that is published to all so B would be able to encrypt as described. And A is the only one that has d. Meaning that he and only he will be able to do $C^d = M^{ed} = M^1 = M$ and get the plaintext message.

We proved in class that it's mathematically correct and secure.

1.1.3. *Math Incorrect (security irrelevant).* Mathematically it's the same as previous section, except using $H(M)$ instead of M and switching between e and d roles (ok since $ed = de$).

The only problem is that using e to decrypt the signature will result with the hash of M, not M itself.

So the described will always fail.

1.1.4. *Math Incorrect. Secure.* By definition, a hash function is not reversible (the result of the hash is usually much much smaller than the original message, there could be countless messages that hash to the same value. Although it would be hard, usually, to find a collision).

Thus it's incorrect mathematically. It's secure because no one, will be able to reverse the hash back into a message, with or without the encryption keys.

1.1.5. *Math correct and secure.* This is how a signature is calculate. $\left(H(M)^d\right)^e = H(M)^{ed} = H(M)$. Since it's usually a difficult task to find hash collision, no one will be able to send messages and calculate signatures for them (as d is unknown to all but A).

1.1.6. *Math correct but insecure.* It mathematically works the same as section e. So it works.

Since e is known to all, anyone would be able to replicate what B is doing (B doesn't have any information that the rest don't as B didn't generate a priv/pub keys).

1.1.7. *Math incorrect but secure.* As we said, B does't even have d so they can't calculate the signature they want.

The message itself won't be possible to decrypt because how RSA works and what we explained in section b.

So it can't be done as described.

1.2. **Which of Hash/MAC/DigSig is the best fit.**

(1) Answers

(a) Digital signature is best suited here. Assuming everyone has Bob's pubkey which the corresponding privkey was used to sign the files, then Bob would upload the file as well as $H(f_{content})^d = sig$ and match it with the file. No one would be able to create a signature that would be verified with Bob's pubkey that is known to all (described in previous sections)

(b) Hashes is the tool for the job. A client would first hash the file and send only the hash (smaller than the entire file). The server would search if the hash exists, if it does, it would tell the client that they don't need to re-upload. If the hash doesn't exist, the client would upload the file and the server would save both the hash and the file in storage.

(c) Again, digital signature is our tool here. Using long term private/public keys for each store, the store will be required to digitally sign the confirmation that they bought an item. Since no one can replicate signing (the **secret** priv key is used on the data), then the store wouldn't be able to deny (everyone saw that the store published its pubkey so it can't be anyone else that got the matching private key).

1.3. **RSA signatures.**

(1) Answers

(a) 3 disadvantages of using RSA signature without hashing

(i) Signatures would be significantly longer (linear to the size of the data)

(ii) A series of signed messages can be signed as a single message by an attacker without knowing the private key (see slide 476 from the lectures' slides)

(iii) Existential forgery - an attacker can randomly generate a signature and then do $sig^e \mod n = m$. Meaning that and attacker can generate random messages that seem like they've been signed by someone else.

(b) Let's look at a bank's server which receives signed messages from clients containing only amount in ILS and returns single use token used to withdraw that amount (valid say for 1 minute). Using regular RSA signature we can randomize a signed message. The message will be interpreted as a number and the bank will return us with a one time token to withdraw that amount. Using hashes, we can't use existential forgery to generate a random signed message so we wouldn't be able to send a signed message that the bank will approve.

1.4. **The chosen bit Amit chose is 0. Nir sends $010101\ldots01$ and $n = 26$.** Meaning that Nir chose 1 as the bit.

Amit chose 0 so to win he needs Nir's bit to be 0.

So he needs to modify the message such that n would be odd.

Using a few tries, he needs to generate a random signed message such that n is odd. The chance for that is 0.5 (pretty high).

So generating random sig message and checking if $sig^e \mod 2 = 1$. When it does, modify Nir's message to $sig\,(M)\,||sig$.

Thus 0 will be chosen bit for Nir and the XOR will be 0 (Amit wins).

Profit!

1.5. **Amit chose 1. Nir chose $M = 000\ldots0011\ldots111$ and $n = 5$. Meaning that Nir's bit is 0.** Amit can win. Again by using generating random signed messages to send as $sig\,(n)$.

When Amit manages to generate a signed sig s.t. $sig^e > 64$, he will send $sig\,(M)\,||sig$ (with sig the random signed msg he found).

Thus 1 will be chosen bit for Nir and the XOR will be 0 (Amit wins).

2. PKI

2.1. AES Symmetric key sent with the product on DoK. Two problems.

(1) Since all of the branches get the product, they all get the symmetric key, so any branch would be able to MITM every other branch and encrypt a new FW file for the product (or SW, I don't really know what kind of company it is). A symmetric key means that both sender and receiver are able to both encrypt and decrypt using the same key.
(2) Since the key is symmetric, a signature on the file isn't reliable as everyone has the key and everyone can use it to sign FW/SW that they send.

2.2. Does sending pubkey with the update help? This still doesn't help. Since the branches don't know the pubkeys of other branches in advance, a malicious branch can send their own malicious FW+pubkey (they will sign the malicious FW with their own private key).

The receiving branch will have no way to know that the sender is a malicious branch and not legit since they trust the pubkey they receive with the FW for verification.

2.3. What's the sent message and how it's verified? Assume the certificate of S is $S_{cert} = S_{name}||S_{pub}||CA_{name}||\left(SHA256\left(S_{name}||\right.\right.$

The sent message is $M||\left(SHA256\left(M\right)\right)^{S_{priv}}||S_{cert}$.

(1) Verify that S_{cert} matches the sender they expect.
(2) Decrypt S_{cert}'s last component (using the trusted pubkey of the CA that the branch saved before) and check that it matches the hash of the first two components.
(3) Decrypt $\left(\left(SHA256\left(M\right)\right)^{S_{priv}}\right)^{S_{pub}}$ using the pubkey in S_{cert} and get $SHA256\left(M\right)$
(4) Calculate $SHA256\left(M\right)$ and compare the result to what they got in step 3.

If any of the steps fail. Don't trust the message.

Note(s): depending on implementation, there might be another check against the revoke list to make sure that the S_{cert} isn't revoked (due to leaked private key of the owner or something like that).

It's also possible for the receiver to verify the certificate using an $OCSP$ but I assume that's not part of the question as it was hinted that that isn't related to the sections prior to section 6.

2.4. Does the CA need to be online while verifying the digital signature. No/yes depending on implementation.

No because:

The CA's public key is saved by each trusting entity. Meaning that the pubkey can be used offline to verify the signature on the certificates other branches sent to identify. The CA signed each branch's cert in advance and after verify that signature of the CA, the branch can trust the pubkey which is also located in that cert.

No need for the CA to do anything.

Again note: Since we saved the CA's pubkey and trusted it, unless we want to update the revoke list, we don't need to ever communicate with the CA at all.

Yes because:

If we use the implementation with OCSP, the CA needs to be online to verify that the S_{cert} is still a valid certificate and that the private key didn't leak.

2.5. Advantages of using a secondary CA.

(1) A leaked private key of a secondary CA is much less bad than one of a root CA because it can be revoked using the revoke list rather than physically and manually trusting a new root CA (which is self signed so we wouldn't trust the internet to get it).
(2) A leaked private key for CA_{update} (and thus adding the CA_{update} to the revoke list) would cause only that certificate to not be trusted and the rest of the CA_* will not cause messages to be refused as not trusted. Other network needs would stay unaffected.

2.6. How does it protect user's privacy. Without OCSP Stapling, if users want to validate the authenticity of a certificate, they need to initiate a connection to the CA itself (a third party), letting them know that they want to access the owner's certificate (I wonder what website users don't want any third party to know they even visit.....).

With OCSP Stapling, the certificate is freshly signed by the CA thus, eliminating the chance that the certificate was revoked by the CA. The owner is frequently asking the CA for freshly signed temporary certificates. If an owner is no longer trusted, the CA will simply refuse to provide fresh temporary certs.

The owner of the cert is the one responsible for getting freshly signed certs thus making the OCSP anonymous for the clients.

2.7. **How does it improve performance?** I'm assuming "השרת" refers to the CA's server.

Since the website keeps a his certificate (very very) fresh, using the temporary certificates described above, the website can serve the same temporary certificate to all of its clients, thus eliminating the need for each client to verify the status of the certificate itself. Instead of `num_of_clients` requests to the CA's server per the time frame of the temporary validity, only one is being made (per the time frame of the temporary validity). That's a stacks up to a big load reduction.

3. Auth Control

3.1. **Protected against context hijacking?** Yes.

The protocol is protected against context hijacking because by the time the server and the client established a connection (context), they already exchanged pub/priv keys and the communication is encrypted so an attacker wouldn't be able to inspect, let alone send messages as either the client or the server.

3.2. **Server impersonation.** No.

An attacker can pretend to be the server, replying with their own g^y and a random string as the challenge.

The attacker would accept any response from the client as a valid challenge solution and now the client communicates with an attacker, thinking it's the server.

3.3. **MITM.** No.

If the attacker is in a position to intercept communication between the client and the server. The attacker can get requests from the client, replace the pubkey with their own, and the same for the response from the server. Except switching the keys, the attacker will leave the authentication protocol the same.

After that, the attacker can inspect all the traffic going between the server and the client without encryption by simply decrypting and re-encrypting the messages using its own keys (the attacker has symmetric keys g^{xa}, g^{ya} with the client and the server respectively).

3.4. **Offline Dictionary (including hybrid).** No (kinda).

The attacker can take the encrypted challenge that the server sends unencrypted by the temporary g^x, and the response challenge.

Now with a dictionary of hashed passwords, the attacker will test (offline) test all the hashes to encrypt the challenge until the attacker finds a hash with the encrypted challenge looking like what the attacker sent. Then the attacker knows the password of the client, and along with the ID_c that was also sent unencrypted, the attacker can now authenticate as this client.

The kinda is because such a dictionary attack is very expensive as the challenge needs to be encrypted using each of the hashes in the dictionary and it isn't nearly as fast as merely searching a plain hash in a dictionary of hashes.

3.5. **Perfect forward secrecy.** Yes.

A leaked long term encryption key (the hashed password) wouldn't expose the past (or future) exchange pub/priv keys that the client/server exchange in the beginning of the exchange.

It would however allow the attacker to authenticate as the client (and possibly event request a password change) if the attacker manages it before the real client changes the password.

3.6. **(attacker with MITM). How does the client verify the server identity? How does it prevent impersonation?** The client sends their pubkey encrypted using the hash of the password. Meaning that only the server (who knows the hash of the password) will be able to decrypt the message and send back the pubkey back.

The client would decrypt, using the DH key, the response the server sent and compare g^x.

An attacker would need to know both the hashed password and y or x to be able to send g^x encrypted with the symmetric Diffie Hellman key so the client would know to trust the attacker.

Since the attacker doesn't have the hashed password, they wouldn't be able to sent a response that the client would trust.

3.7. **(attacker with MITM). How does the server verify the client identity? How does it prevent impersonation?** The server authenticate the client is really the client by the fact that the client was able to complete the challenge (meaning that the client was able to decrypt g^y using the hashed password).

An attacker would need to be able to send a solved challenge encrypted under the DH key but since the attacker doesn't have the hashed password, it would need to rely on the real client to solve the challenge. But the client would first verify that they're talking to the real server (as explained above) so the attacker wouldn't be able to get a solved challenge (that is not encrypted by keys the attacker doesn't know).

So the attacker wouldn't be able to get a solved challenge to send so the server would refuse communication with it.

3.8. **How does it prevent MITM?.** For MITM attack the attacker needs read knows the public keys of both the server and the client. But since the only time the server sends their pubkey, it's encrypted under the hash of the password, the attacker wouldn't be able to decrypt it and thus wouldn't be able to establish a MITM connection.

3.9. **How does it protect from dictionary attack?** For a dictionary attack on a parameter, an attacker needs that only a single parameter would be unknown. In described protocol, each step has two parameters that are unknown to the attacker.

In step 1, it's the password and the g^x.

By the time we got to step 2 the server would have already gotten g^x. So the first argument has two parameters unknown and the second has 3 unknown parameters ($K, g^x, challenge$).

In step 3 we have K and $challenge$ encrypted.