Nick Weaver
Fall 2018

CS 161
Computer Security

Discussion 4

Week of September 17, 2018: Cryptography II

Question 1 ***Public Key Basics*** **(10 min)**

Assume Alice and Bob are trying to communicate over an insecure network with public key encryption. Both Alice and Bob have published their public keys on their websites. (You can assume they already know each other's correct public key)

(a) Alice receives a message: *Hey Alice, it's Bob. You owe me 100 bucks. Plz send ASAP*. The message is encrypted with Alice's public key. Can she trust it?

Solution: No, she cannot. As the name implies, a public key is generally publicly known. Anyone can go to Alice's website and use her public key to encrypt whatever they want.

(b) Bob receives a message: *Hey Bob, that last message you sent me was sketchy. I don't think I owe you 100 bucks. You owe me.* The message is digitally signed using Alice's private key. Can he trust it was from Alice? How does he verify this message?

Solution: Yes. As long as Alice has kept her private key secure, she is the only one who could construct such a message. Bob (and anyone else) can verify it using Alice's public key.

Note that because they operate in a public key setting, signatures provide three properties: integrity, authenticity, and *non-repudiation*; the last of which is not provided by symmetric key MACs. Since Alice has published her public key to the world, everyone knows that the public key belongs to Alice; and anyone can use the public key to verify if a signature was generated using the corresponding private key. Only Alice can generate signatures that correctly verify under her public key because she's the only one who knows the private signing key. Thus, if anyone sees a message with a signature that verifies under Alice's public key, it must've been signed by Alice.

Note that when we say "the message is digitally signed", Alice will need to send both the message and the signature.

(c) Alice receives a message: *Hey Alice, I know things have been rough these last two messages. But I trust you now. Here's the password to my Bitcoin wallet which has over $100.* The message is encrypted with Alice's public key. Alice decrypted this and tested the password, and it was in fact Bob's! Can an eavesdropper figure out the password?

Solution: No. An eavesdropper would not have Alice's private key, which is needed to decrypt the message.

Question 2 *Confidentiality and Integrity* (25 min)

Alice and Bob want to communicate with both confidentiality and integrity. They have:

- A symmetric key encryption function $E(K, M)$ and corresponding decryption function $D(K, M)$. They have already securely shared the key K.
- A cryptographic hash function H (recall that H is not keyed, so anyone can compute $H(x)$ given x).
- A secure MAC generation function $MAC(K, M)$.
- RSA Signature Algorithm $\mathit{SIGN}_d(M)$ and $\mathit{VERIFY}_n(M, S)$, with d being Alice's private key and n being her public key. (For convenience, we often leave out mention of e as part of the public key.)

You may assume that H, E, MAC, and SIGN_d do not interfere with each other when used in combination—for example, if you compute $H(E(K, M))$, the message M will be confidential because E guarantees it, and the computation of H makes no difference. To send message M to Bob, Alice has the option to use any of the six schemes listed below.

Alice Sends to Bob
1. $C = H(E(K, M))$
2. $C = C_1, C_2$: where $C_1 = E(K, M)$ and $C_2 = H(E(K, M))$
3. $C = C_1, C_2$: where $C_1 = E(K, M)$ and $C_2 = MAC(K, M)$
4. $C = C_1, C_2$: where $C_1 = E(K, M)$ and $C_2 = MAC(K, E(K, M))$
5. $C = \mathit{SIGN}_d(E(K, M))$
6. $C = C_1, C_2$: where $C_1 = E(K, M)$ and $C_2 = E(K, \mathit{SIGN}_d(M))$

(a) Consider the threat model where Eve is only able to eavesdrop and Alice and Bob are using the key K for the first time and will use it once and only once. Furthermore, Eve has no additional information about the plaintext messages that will be sent. For each scheme above, determine whether or not the scheme (1) provides *confidentiality* and (2) if Bob is able to *correctly decrypt* the message (if he is able to, write down the decryption procedure).

Solution: Only option 3 does not provide confidentiality, because the MAC is sent in plaintext, and MACs are not guaranteed to provide confidentiality (they could leak information about message M). For example, consider the secure MAC: LeakyMAC(m) = HMAC$(m)||m$. LeakyMAC is a secure MAC (any attack that managed to forge a MAC for LeakyMAC could be used to forge a MAC for the standard HMAC), but LeakyMAC clearly leaks the full message m!

A less contrived way to see this is that if $M_1 = M_2$, then MAC(M_1) = MAC(M_2), i.e., if Eve sees the MAC for two different messages that happen to have the same contents, then she'll be able to tell that the messages were the same, which violates our confidentiality goals.

Bob cannot decrypt Option 1 because he cannot invert H. Similarly, he cannot extract the original message from the signature sent in Option 5.

A side note: When sending a signature $S = \mathit{SIGN}_d(M)$, we also need to send the message M too. The recipient will need the message M to be able to verify the signature. Notice that the signature S doesn't include/reveal the message. For example, for RSA, the signature S is defined as $S = H(M)^d \bmod n$, so it depends on the hash of the message; given $H(m)$, you can't recover M (since cryptographic hashes are one-way). Also, a signature is a short fixed-length value: e.g., with a 2048-bit RSA modulus (if n is 2048 bits), the signature is 2048 bits long. Obviously that's not long enough to carry a long message M. This is why Option 5 is not decryptable.

(b) Now consider the threat model where Mallory is able to eavesdrop and actively alter/forge the message Alice sends to Bob. For each decryptable scheme, determine whether or not the scheme provides *integrity*; that is, will Bob be able to detect any tampering with the message? If the scheme provides integrity, describe how Bob checks the integrity of the message. If not, describe how Mallory could alter the message without Bob detecting.

Solution: Option 2 does not provide integrity, as Mallory can forge a message by sending Bob C_1', $H(C_1')$

Option 3 and 4 provide integrity because if Mallory alters C_1 into C_1', she cannot alter C_2 into a valid C_2' because she cannot find the correct MAC tag.

Option 6 provides integrity, because if Mallory alters C_1 into C_1', she cannot alter C_2 into a valid C_2' because she does not have the private key to forge the digital signature

Note: Using the same key K in Option 4 for encryption and MAC generation is generally a bad idea in practice. Depending on what encryption scheme E is used and what MAC algorithm is used, using the same key K could cause vulnerabilities/exploits. It's better to use 2 independent keys, K_1 for encryption and K_2 for MAC generation.

(c) If Alice and Bob use these schemes to send many messages, the schemes become vulnerable to a *replay attack*. In a replay attack, Mallory remembers a message that Alice sent to Bob, and some time later sends the exact same message to Bob, and Bob will believe that Alice sent the message. How might Alice and Bob redesign the scheme to prevent or detect replay attacks?

Solution: One possible way is for Alice to include nonces in her message. A nonce is an arbitrary number that can only be used once. Alice can, along with her message, send a nonce as well as either a MAC or Digital Signature of this nonce to Bob. This prevents replay attacks because Bob can realize if he gets the same message twice. The MAC/Digital Signature of the nonce is important because it protects the integrity of the nonce.

Summary Chart:

Alice Sends to Bob	Conf	Integ	Decryption + Integ-Check (if applicable)
1. $C = H(E(K, M))$	yes	N/A (can't decrypt)	can't
2. $C_1 = E(K, M)$, $C_2 = H(E(K, M))$	yes	no	$M = D(K, C_1)$
3. $C_1 = E(K, M)$, $C_2 = MAC(K, M)$	no	yes	$M = D(K, C_1)$ and $C_2 \overset{?}{=} MAC(K, M)$
4. $C_1 = E(K, M)$, $C_2 = MAC(K, E(K, M))$	yes	yes	$M = D(K, C_1)$ and $C_2 \overset{?}{=} MAC(K, C_1)$
5. $C_1 = SIGN_d(E(K, M))$	yes	N/A (can't decrypt)	can't
6. $C_1 = E(K, M)$, $C_2 = E(K, SIGN_d(M))$	yes	yes	$M = D(K, C_1)$ and $VERIFY_n(M, D(K, C_2))$

Question 3 *Password Hashing* (10 min)

When storing a password p for user u, a website randomly generates a string s (called a "salt"), 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.

Functions	One-Way	Second Pre-Image Resistant	Collision Resistant
H_1	Yes	No	No
H_2	Yes	Yes	No
H_3	Yes	Yes	Yes

A function H is *second pre-image resistant* if given m_1, it is difficult to find m_2 such that $m_1 \neq m_2$ but $H(m_1) = H(m_2)$. (Sanity check: how does this differ from being collision resistant?)

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 attacker hacks into the site and obtains all of the user tuples, that still won't enable recovery of their passwords.

(c) A general method for attackers to recover passwords from their hashes is to employ *dictionary attacks*: the attacker has a (large) list of possible/common passwords and tries each one to see if it yields an observed hash. If so, then the attacker has identified a working password for the associated account.

One countermeasure for defenders is to employ an *expensive* hash function: one that requires significant computation to evaluate. Doing so slows down the rate at which attackers can try different passwords from their dictionary. An easy way to generate an expensive hash function is to use N iterations of a standard hash function such as SHA-256, where N might be 1,000, increasing the time required to try a given password from microseconds to milliseconds.

In response to this countermeasure, attackers can *precompute* the hashes corresponding to their password dictionary, so that once in possession of a set of hashed passwords, they still can quickly locate ones that correspond to entries in their dictionary.

Using a salt is a counter-countermeasure to this technique. If precomputing, the attacker must now derive hashes for each possible salt. If the salt has s bits, then this increases the dictionary computation (and size) by a factor of 2^s. Thus, for a sufficiently large salt, the attacker gains no benefit from precomputation: the best they can do is run through their dictionary anew after acquiring the tuples (which, recall, include the associated salts).

(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, they need 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.