CS162 Discussion #14: Security

Announcements

- Phase 4 code due Tuesday, 12/07 @ 11:59pm (`submit proj4-code`)
- Final exam will be Thursday, 12/16 @ 8:00am – 11:00am in 10 Evans
- This is our last discussion. ;_; Thanks for a great semester! Happy holidays ☺
    - There will be additional office hours during dead week (TBA)

H(browns) = yum :9

Last week we discussed how we could implement networking between computers. This opens up a lot of possibilities in how we can use our systems, but it also introduces more security issues. If we aren't careful, other users on the network can intercept and read messages meant to be private, make other users think they're talking to someone they're not, or any number of other malicious activities. We want to take security measures to prevent these things from happening.

The basic idea for keeping messages private from people who might be looking at your packets is to encrypt the messages with a key, using a cryptographic algorithm, before sending them. The recipient can then decode the encrypted message to obtain the original text. Unencrypted messages are called *plaintext*; encrypted messages are called *ciphertext*. When using encryption, there are some important properties we want to hold:

- It shouldn't be possible to figure out plaintext from ciphertext without having the correct key.
- Having both the plaintext and ciphertext for a message shouldn't allow you to figure out what the key is.

The simplest protocol we can use is *symmetric key encryption*. This means that both the sender and the recipient have the same key, which can be used to both encrypt and decrypt the message. This method works just fine as long as the key stays private to the sender and the recipient. However, this introduces the problem of how to securely establish the symmetric key between the two sides, especially if they have never met before.

One way we can alleviate this problem is by using a trusted third party *authentication server*. In this system, both the sender and receiver register with an authentication server, each establishing their own password (shared with the server). When Alice wants to communicate with Bob, she requests a key from the server. The server sends back a message encrypted with Alice's shared key (based on a hash of her password) with the server. Alice then attempts to decrypt this message using her password. If her password is correct, the decrypted message will contain the a temporary key for communicating with Bob, as well as a message containing the same key but encrypted with Bob's shared key with the server. Alice can then send this message to Bob, who will then decrypt the message with his password and obtain the session key. Session keys are temporary to avoid *replay attacks*, in which a malicious user can take encrypted messages sent from Alice to Bob and send them again later to get Bob to think Alice is communicating with him again.

Another way we can establish secure communication between Alice and Bob without relying on a central key server or worrying about a way to establish a shared key is by using *public-key encryption*, also called *asymmetric key encryption*. In this scheme, Alice and Bob each generate their own public/private key pairs. Any message encrypted with the public key can be decrypted with the corresponding private key, and vice versa – but it should be virtually impossible to derive one key from the other. Alice and Bob each make the public keys freely available. Then, when Alice wants to send a message to Bob, she encrypts her message with Bob's public key, which then only Bob can decrypt with his private key. In this scenario, Alice can also assure Bob that the message is really coming from her by also including a *signature* in the message. Alice's signature is encrypted by her private key; if Bob can decrypt it using Alice's public key, he knows that it was really signed by Alice.

There's another problem with this, though: how does Alice know that Bob's public key is really from Bob? She can verify Bob's public key by turning to a *certificate authority*. A certificate authority is a trusted organization that verifies public keys by signing them with its own private key. In this case, Alice has a public key for the certificate authority that she trusts. She can then request a certificate for Bob's public key, which contains Bob's public key along with a hashed version of the key signed by the certificate authority. After decrypting the signed hash, Alice can compare the key with the hash of the key to verify that Bob's key is correct.

Up until this point, we've mentioned hashing messages or signatures. Cryptographic hashing is an important security measure that allows us to verify the accuracy of messages received. This is to prevent someone from intercepting a message and altering the contents so that the receiver gets the wrong message. If Alice sends a signed hash of her message along with the actual message, Bob can check the hashed message against the signed hash. If they don't match up, he knows the message has been tampered with and disregards it. In order for hash functions to be cryptographically secure, they must have the following properties:

Let $H(m) = h$ be a hash function.

- Infeasible to find a message that has the same hash as another message. (Given h_1 where $H(m_1) = h_1$, it's hard to find m_2 such that $H(m_2) = h_1$)
- Infeasible to find two messages with the same hash. (Hard to find m_1, m_2 such that $H(m_1) = H(m_2)$)
- A change in a message causes a seemingly random change in its hash; knowing a message's hash doesn't reveal anything about the message itself.

A Final Word

Thanks again for being an awesome class that I can be proud of! :D You guys are some of the best students/friends I've ever had. I've learned a lot from being your TA this semester – thanks also for putting up with my mistakes and all those times I didn't have the answers. Good luck on the final!