Alin Tomescu, CSE408
Tuesday, April 5th, Lecture #18

Network protocols II (nonces and timestamps)

Replay protection using nonces

As always, Alice wants to send a message to Bob, and we don't want her messages being replayed.

- Alice requests a nonce: "Hi, I'm Alice, can I has nonce? Kthxbye."
- The server sends her an r (random and big), this is the nonce.
- Alice sends m and r **signed** with $Sig_{ska}(m, r)$.
  - The server needs to remember r, Alice's identity and her public key.
- Bob validates the nonce and if it's good, executes the command, and discards the nonce to avoid replay.

Attacks

If an attacker **replays** the message, Bob will know that the message has already been sent (r would've been crossed out the first time).

Man in the middle attack: Attacker sends an r to Alice, getting a message from her that he can replay later.

- So we'd better sign r with Bob's secret key, when we send it to Alice

Anyone can request a nonce from the server, which could be bad since nonces are not bound to the two parties communicating. So the request message should be more specific:

- Alice requests nonce sending $m = \text{alice, bob}, request\ nonce, Sig_{ska}(\ldots)$
  - But this is replayable
  - Therefore, if the server already sent Alice an r, he shouldn't send another one until he receives the message for the first one.
- Bob responds with $m = \text{bob, alice}, r, Sig_{skb}(\ldots)$
- Alice then responds with $m = \text{alice, bob}, Enc_{pkb}(cmd), Sig_{ska}(\ldots + r)$
  - Encryption is used to protect the command and the nonce is not part of the message

DoS attack:

- Alice wants to send two commands m_1 and m_2
- Alice requests a nonce for the first command m_1
- Server sends $r_1, Sig_{skb}(r_1)$ for the first command,
- Alice sends the command m_1, attacker discards m_1 but remembers $r_1, Sig_{skb}(r_1)$
- Alice requests a nonce for the second command m_2
- Server sends $r_2, Sig_{skb}(r_2)$ for the second command, attacker discards it and replays the previous r_1 to Alice
- Alice gets r_1 and sends the command m_2 but doesn't realize m_1 was never executed
  - This can be prevented if you only allow one "active" nonce, so the second nonce would never be granted to Alice until she sends the message for the first one
  - ACKs and more nonces can also be used to solve this problem

Fix using **ACKs**:

- Send an ACK each time a command is executed and an r is crossed out.
- Each message exchange (request r, get r, send m) is ACKed by Bob: $m = \text{alice, bob}, r, ACK, Sig_{skb}(\ldots)$

Fix using extra nonces:

- Alice can send a nonce of hers when she requests the nonce from Bob
  - $m = \text{alice, bob}, request\ nonce, r_a, Sig_{ska}(\dots)$
- Bob replies as before but includes her nonce too
  - $m = \text{bob, alice}, r_a, r_b, Sig_{skb}(\dots)$
- Alice sends her message with the nonce r_b which was bound to it using the r_a value
  - $m = \text{alice, bob}, Enc_{pkb}(cmd), Sig_{ska}(\dots + r_b)$
- Alice is now "a little" stateful (she has to remember her nonce for a while)
- Alice still never knows whether the copy command got to the server
- Attacker can still drop the last message since there's no ACK

Advantages over the counter mode

- Alice is stateless, she does not need to maintain any state.
- No synchronization, synchronization is free.
- If the server crashes, no problem other that there could be an r out there, so a message for that r could be replayed when that r happens to be selected again later. The probability is low though.

Conclusion

- The client is stateless
- Synchronization is free
- Pipelining issues

Timers or timestamps

- Alice sends $m = \text{alice, server}, t, cmd, Sig_{ska}(\dots)$
- Server gets the time t' and checks if $t \geq t' - \varepsilon \Leftrightarrow t' - t \leq \varepsilon$
  - ε is the time-window within which messages are accepted by the server
  - If $t' - t \leq \varepsilon$ then accept the message, otherwise reject it

A major design decision: how big should the time-window ε be?

- a lower bound for ε is half the RTT for TCP (let's say 30 ms)
- there's also the issue of clock sync/drift
- $\varepsilon \geq 100ms$ usually

Within ε, the attacker might be able to do a lot of stuff on a fast network.

- drop, reorder and replay messages

Fixes:

- server should not allow two messages to have the same timestamp (solves replay)
- one suggested fix: is to buffer up all the incoming messages
  - m enters queue when it's fresh (within the time window)
  - m exists queue when it's rotten (when it's not in the time window anymore)
    - that's when you execute m

By the time we fix all the deficiencies, timer mode becomes counter mode.

ACKs:

- Alice sends $m = \text{alice, server}, t, cmd, Sig_{ska}(\dots)$

- Server sends $m = \text{server, alice}, ACK, t, t', Sig_{skb}(\ldots)$ as soon as it receives it, if it is accepted.

Key agreement protocols

- we have to rely on a third-party to do it
- we need some secret to separate the good guys from the bad guys

Needham Schroder

- out there for a while until people realized it was broken
- two version, public and symmetric key versions
- we have Trent, the trusted party (key distribution center)
- Trent has a secret that he shares with Alice and another secret that he shares with Bob