Alin Tomescu, CSE408
Thursday, March 31st, Lecture #17

Network protocols I (counters)

We will discuss network protocols in a more abstract setting, to understand some of the issues associated with them and how to fix them.

Suppose you want to build a file-server. You have a **client Alice** and she wants to authenticate to a **file server Bob** because she wants to issue the server a command. The file server wants to know who is executing this command.

Trial and error protocols

Let's come up with a few authentication protocols in a public-key setting (Alice has a secret key sk and Bob has the public key pk associated with it).

Protocol 1

Alice – the client	Bob – the server
"Hi, I'm Alice" →	
	← Prove it!
$m = \langle r, Sig_{sk}(r) \rangle$ →	
	Bob checks the signature on r using $Vrfy_{pk}(r, sig_{from\ Alice})$
"delete $file.txt$" →	

Replay attack: Once a bad guy has seen such a signature pair $m = \langle r, Sig_{sk}(r) \rangle$, he can store it and authenticate later as Alice.

Man in the middle attack: We can let Alice authenticate. After that, we intercept her messages, change them to whatever we want and then have the server execute them (as if Alice was executing them).

Protocol 2

Alice – the client	Bob – the server
"Hi, I'm Alice" →	
	← Here's r (random and big), sign it and prove it!
$m = Sig_{sk}(r)$ →	
	Bob checks the signature on r using $Vrfy_{pk}(r, m)$
"delete $file.txt$" →	

Man in the middle attack as before.

Protocol 3

Alice – the client	Bob – the server
"Hi, I'm Alice" →	
	← Here's $Enc_{pk}(r)$ (r is random and big), sign r it and prove it!
$m = Sig_{sk}(r)$ →	
	Bob checks the signature on r using $Vrfy_{pk}(r, m)$
"delete $file.txt$" →	

Encrypting helps to add some secrecy but still doesn't solve the overall problem.

Alin Tomescu, CSE408
Thursday, March 31st, Lecture #17

The problems:

- After authentication the rest of the conversation is not signed (or authenticated). An attacker can therefore jump in and send commands in the name of Alice.
    - Having Alice sign her commands would solve this, but…
- There is nothing stopping an attacker to reissue a command once he sees one. Some state has to be maintained on the server side to prevent this.
    - Using counters, timestamps or nonces can help with this

Security principles

These principles will ensure that your protocol will not be susceptible to replay or MITM attacks:

- bind the messages together (previous schemes are vulnerable because commands are not bound to authentication)
- explicit messages (each message should be self contained and interpretable on its own)
- replay protection (use nonces)
    - counters
    - timestamps
    - nonces

A better protocol using counters

The file server Bob has the public-key pk of Alice and a **counter** ctr_b. Alice has her secret key sk and a counter ctr_a. We will see how ctr_b and ctr_a are related to each other. Initially, both counters are set to 0. As messages get exchanged, the counters get incremented and are kept "in-sync."

When Alice issues a command cmd, she sends:

$$m = \text{alice}, \text{bob}, cmd, ctr_a, Sig_{sk_{alice}}(\ldots)$$

So now, she is including her name and signing everything in the message. Therefore, the message is **self contained**, and **commands are bound to authentication**. The counter ctr will be used for **replay protection**.

Now, how should we handle these counters? Suppose authentication is done securely, as we've seen in the previous examples and a message m is sent to Bob from Alice.

$$m = \text{alice}, \text{bob}, cmd, ctr_a, Sig_{sk_{alice}}(\ldots)$$

First way to handle counters: send and wait for an ACK

If $ctr_a = ctr_b$, then:

- Bob replies with an ACK message $m = \text{alice}, \text{bob}, ACK, ctr_a + 1, Sig_{sk_{bob}}(\ldots)$
- Bob increments his counter $ctr_b = ctr_b + 1$
- Alice gets the ACK and increments her counter $ctr_a = ctr_a + 1$
- The two counters are kept in sync, as long as Alice gets Bob's ACK.
- Messages cannot be pipelined. Alice can only send one message, wait for its ACK and then send the next one.

Otherwise, if $ctr_a > ctr_b$ then Alice's counter is too high. Since messages cannot be pipelined, Bob should discard this message and maybe send an ACK to Alice letting her know which message he is expecting.

Otherwise, if $ctr_a < ctr_b$, then it's probably because a message is being replayed. Bob should discard this message and maybe send an ACK to Alice letting her know which message he is expecting.

Second way to handle counters: cumulative ACKs, TCP style

If $ctr_a \geq ctr_b$, then:

- Bob stores the message, but doesn't execute the command until all the message from $[ctr_b$ to $ctr_a]$ have arrived.
    - This prevents attackers from discarding messages in a pipelined set
- If this message with ctr_a fills a gap from $[ctr_b$ to $ctr_b + n]$ then set $ctr_b = ctr_b + n$ and send an ACK back to Alice with this new counter.
    - Also execute the commands in these messages, now that they are all here in order.
    - Note that this "filling the gap" condition covers the basic case where $ctr_a = ctr_b$
- When Alice gets this ACK she will set her counter to the counter in the ACK message.

Otherwise, if $ctr_a < ctr_b$, then:

- discard the message because this is a replay attack

Possible attacks to consider

What happens if Alice has 2 connections to the file server?

If there's more than one file server then Alice has a counter for server A and a counter for server B, then the attacker can wait for the counters to sync and can replay messages from one file server to another. Therefore, the file server's name needs to be included in the messages Alice sends, and in the ACKs from the server too, for friendliness.

What happens if the server crashes?

- server loses state
- a resynchronization protocol has to be executed

Session IDs

To enable Alice to have two connections (or sessions) open to Bob, another piece of information is added to the messages: a Session ID number that identifies a communication session between Alice and Bob.