Network Security

Professor Gaurav S. Kasbekar
Department of Electrical Engineering
Indian Institute of Technology, Bombay

Week - 04
Lecture - 20
Authentication: Part 1

Hello, in the previous few lectures, we discussed the principles of cryptography and message integrity. Now, we start our discussion of another fundamental building block of network security, namely authentication. We will discuss authentication in this lecture and the next few lectures. What is authentication? Sometimes it is also called "end-point authentication".

It is a process of one entity proving its identity to another entity over a computer network. So, suppose, for example, that Alice and Bob are two users who are connected over a computer network. Alice needs to prove to Bob that she is indeed Alice, and Bob needs to prove to Alice that he is indeed Bob. This process is known as authentication or end-point authentication. Here are some examples.

A user who wants to access email proves his/her identity to the email server. Conversely, the email server also has to prove to the user that it is a legitimate email server. Another example is a mobile device that wants to connect to the internet wirelessly via a base station; it proves its identity to the base station and vice versa. So, only users who are authorized to connect to the internet should be allowed to connect. So, a mobile device must need to prove its identity to the base station, and the base station must also prove its identity to the mobile device, which will ensure that the mobile device does not connect to fraudulent base stations.

An authentication protocol typically runs before the two entities perform some other task. In this example of the user accessing email, the authentication protocol runs before the user checks their inbox or sends an email. In this second example of a mobile device that wants to connect to the internet wirelessly, an authentication protocol runs before the mobile device exchanges information over the internet via the base station. So, authentication is a process that is performed before the two entities who are authenticating, before they perform some other task. An analogy is human authentication.

We humans authenticate all the time. For example, humans who meet in person recognize each other's faces. If Alice and Bob meet in person, then Alice recognizes Bob by his face and vice versa. Humans who talk over the telephone recognize each other's voices. And when a person visits a foreign country, the immigration official authenticates him/her by checking the passport.

So, these are means of human authentication. But we clearly need different means to authenticate over a computer network. So, we'll discuss protocols used to authenticate in a computer network in this lecture and the next few lectures. So, we should make one distinction. Here, we focus on the authentication of a live party.

That is, we assume that the entity being authenticated is communicating with the authenticating entity. So, this is different from message integrity, which we discussed in the previous few lectures. Message integrity is also called message authentication. In message integrity, recall that a receiver of a message verifies that the message was indeed sent by the claimed sender. So, in the case of message integrity, Alice may send a message to Bob, and then Alice may even log off, and Bob needs to check that the message received is indeed from Alice and was not modified during transit.

In contrast, in end-point authentication, which we are discussing now, Alice and Bob are live at the two ends of a computer network, and they need to check whether the entity on the other side is indeed who they claim to be. So, that's the difference between authentication and message integrity. So, we now design an authentication protocol. Suppose Alice needs to authenticate herself to Bob. First, we discuss the case of one-way authentication.

That is, Alice needs to authenticate herself to Bob. But Bob doesn't have to authenticate himself to Alice. Later on, we'll discuss the case of mutual authentication, where Alice authenticates herself to Bob and vice versa. Bob also authenticates himself to Alice. So, we start with one-way authentication.

Alice needs to authenticate herself to Bob. Let 'ap' denote the authentication protocol. We'll design a working protocol in several steps. First, we'll consider some protocols, which have some flaws. Later on, we'll correct those flaws and design an authentication protocol that works correctly.

So, let's start with a simple protocol. Alice just sends a message to Bob saying that she's Alice. So, this illustrates a protocol to authenticate herself. Alice just sends a message to Bob saying that 'I am Alice'. But this has a flaw.

An intruder, say Trudy, can also send such a message. That's illustrated here. An intruder, Trudy, can also send a message claiming, 'I am Alice'. So, this is clearly a flawed protocol. Just saying that 'I am Alice' doesn't prove that the sender is indeed Alice.

Here's another better authentication protocol. Suppose Alice has a well-known network address, for example, an IP address from which she always communicates. So, for example, Alice may be using a desktop computer, which has a certain IP address, and she always communicates from that IP address. Alice sends a message to Bob saying that she's Alice, and Bob checks whether the source IP address of the message matches Alice's address. This illustrates the protocol.

Alice sends the message, 'I am Alice', and the source IP address from which this message is sent is Alice's IP address. Bob verifies whether the source IP address is that of Alice. If yes, then Bob believes that it is Alice on the other side of the connection. So, does this protocol achieve authentication? This is better than the previous protocol.

There is one entity that Bob can check whether the IP address is correct, that of Alice or not. But unfortunately, this protocol also does not achieve authentication. It has a flaw. An intruder, Trudy, can create a message saying that she's Alice and put Alice's IP address in the source IP address field and send it to Bob. This process is known as IP spoofing, which is illustrated in this figure.

Trudy sends a message to Bob saying, 'I am Alice', and the source IP address in that message is Alice's IP address. On the internet, it's quite easy to create a message and put the source IP address equal to that of some other user. So, using this IP spoofing process, Trudy can send a message to Bob claiming that Trudy is Alice, and the source IP address can be Alice's IP address. So, hence, this protocol also fails. The reason is that it's easy to spoof IP addresses on the internet.

Here's another authentication protocol, ap3.0. Suppose a shared secret, such as a password, exists between Alice and Bob. So, Alice and Bob know a secret, which is a password, and no one else knows that secret. To authenticate herself, Alice sends her password to Bob. Does this protocol achieve authentication?

So, at first, it looks like a secure protocol. No one other than Alice knows the password. So, no one else can authenticate themselves to Bob. But this protocol also has a flaw. An intruder, say Trudy, can sniff the password and later on authenticate herself to Bob.

So, Trudy can tap the communication channel between Alice and Bob, and sniff the password, and later on Trudy can send the password to Bob claiming to be Alice. This shows the protocol. Alice sends the message 'I am Alice' along with the password to Bob. This password is a secret shared between Alice and Bob. But then intruder Trudy taps the communication channel between Alice and Bob and records the password.

Later, Trudy sends a message saying 'I am Alice' to Bob along with the recorded password. Bob doesn't know whether this password is indeed sent by Alice or was sniffed by some intruder who is now sending it. So, because of this, this protocol also fails. Let's try to fix this protocol. If we can encrypt the password, then if Alice encrypts the password before sending it, then Trudy won't know the password, even if he/she records the password.

So, even if Trudy traps the communication channel between Alice and Bob, Trudy won't come to know the password if it is encrypted. So, let's try encrypting the password. So, in the authentication protocol ap3.1, a shared secret or password between Alice and Bob exists again. To authenticate herself, Alice sends her encrypted password to Bob. So, even if Trudy taps the communication channel between Alice and Bob, Trudy is not able to find out the password.

Bob decrypts the message and checks whether the password is correct. So, does this protocol achieve authentication? Here it is true that Trudy does not get to know the password of Alice, but Trudy does not need to know the password. So, notice that just the encrypted password needs to be sent to Bob to authenticate Alice. So, here, an intruder, Trudy, can sniff and store the encrypted password and later on just play back the encrypted password to authenticate himself/herself.

So, here, the password is encrypted, but then Trudy just sniffs the encrypted password and plays it back. And then, Bob may think that it is Alice on the other side. So, for this reason, this protocol also fails. So, to design a secure protocol, we introduce a concept known as a nonce. Protocols ap3.0 and ap3.1, which are the password-based protocols we discussed, failed because Bob could not tell whether Alice was live, that is, currently at the other end of the connection, or the messages he was receiving were a recorded playback of an earlier authentication by Alice.

So, this property of liveness, whether Alice is actually there on the other side of the connection or Alice was there previously and her messages were recorded and then are being currently laid out. So, this liveness or absence of liveness cannot be checked in protocols ap3.0 and ap3.1. That's the reason they failed. So, this liveness can be found out using a nonce. A nonce is a number that a protocol uses only once.

That number is never used again by the protocol during its operation. So, in the context of authentication, Alice may authenticate herself to Bob many times, for example, on several days or once every week. So, over a long duration, Alice may authenticate herself many times to Bob. But when a nonce is used, a different nonce is used for every authentication. The nonce never repeats.

So, the reason for calling it a nonce is it stands for n-once. It's a number n, which is used only once. We want to design an authentication protocol using the concept of a nonce. So, this is a protocol, authentication protocol ap4.0, which uses a nonce. Suppose a shared symmetric key K_{A-B} between Alice and Bob exists.

This protocol shown in this figure is used for authentication. So, Alice sends the message 'I am Alice' to Bob to initiate the protocol. Bob selects a nonce, say R, and sends it to Alice. So, this R is a number which will not be used again if Alice wants to authenticate again to Bob. So, this number R is sent to Alice.

So, this nonce R is a challenge sent by Bob to Alice to see whether Alice can respond to the challenge correctly. To respond to the challenge, Alice encrypts the nonce R using the secret shared key K_{A-B} to get $K_{A-B}(R)$ and sends it to Bob. So, $K_{A-B}(R)$ is the encrypted version of the nonce R. Then Bob decrypts the received message $K_{A-B}(R)$ and checks whether it equals R. If yes, then Bob assumes that it is indeed Alice on the other side of the connection because Alice is the only one other than Bob who knows the secret K_{A-B}. So, this proves that it is indeed Alice on the other side. So, why does this R have to be a nonce?

- Alice encrypts R using K_{A-B} to get $K_{A-B}(R)$; sends it to Bob

Why should it never be repeated in our subsequent authentication by Alice? So, the reason is that this R is a nonce, so later on, suppose some intruder Trudy records this entire conversation between Alice and Bob. Later on, if Trudy tries to authenticate as Alice, Bob

will send some other number, say R'. So, Bob won't send the same number R'. So, this recorded $K_{A-B}(R)$ won't be useful to Trudy to authenticate himself/herself to Bob.

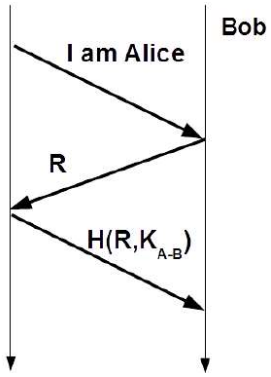
So, in this first authentication, some number R is sent, then in a subsequent authentication, some other number will be sent as a challenge R. Hence, recording these messages does not help an intruder to authenticate themselves to Bob. That's the reason that a nonce is useful. This protocol correctly achieves authentication. Now, ap4.0 correctly achieves authentication, but it uses encryption.

So, again suppose a shared symmetric key K_{A-B} between Alice and Bob exists and Alice needs to authenticate herself to Bob. Authentication protocol ap4.0, which we discussed on the previous slide, has the shortcoming that it uses encryption, and we know that encryption is time-consuming. Can we modify protocol ap4.0 such that no encryption is used? So, we can try using a cryptographic hash function. We know that computing a cryptographic hash function is much faster than encryption, so let's try that to create an authentication protocol.

Here's the protocol, which does not use encryption. Alice sends a message, 'I am Alice', to Bob. Bob selects a nonce, say R, and sends it to Alice. So far, it's the same as the previous protocol, ap4.0. But now Alice does not compute $K_{A-B}(R)$; instead, Alice computes $H(R, K_{A-B})$ and sends it to Bob, where H(.) is a cryptographic hash function.

So, notice that this is like a message authentication code; this $H(R, K_{A-B})$ is like a message authentication code. So, Alice computes this $H(R, K_{A-B})$ and sends it to Bob. Now, how does Bob verify whether this is correct? Bob knows K_{A-B}, and Bob also knows the nonce R. Bob also independently computes $H(R, K_{A-B})$ and checks whether it equals the value that was sent by Alice. If yes, then the authentication is successful.

- Alice computes $H(R, K_{A-B})$ and sends it to Bob
  - where $H(.)$ is a crypt. hash function
- Bob also computes $H(R, K_{A-B})$, and checks whether it equals the value sent by Alice
- In above protocol, if $H(.)$ is replaced with $c(.)$, a checksum function, is resulting protocol secure?
  - No, since $c(R, K_{A-B})$ may reveal some information about K_{A-B}



So, this protocol also achieves authentication correctly. So, again, the fact that this is a nonce is useful because if Trudy records all these messages, then next time Trudy tries to authenticate as Alice, a different number will be sent—a challenge R. So, these recorded messages will not help Trudy to authenticate themselves as Alice. And the fact that this is the hash function is crucial because, from this value $H(R, K_{A-B})$, an intruder could not be able to find out any information about the secret K_{A-B}. So, in this protocol, if this H(.) is replaced with c(.), which is a checksum function, then is the resulting protocol secure? It is not secure because a checksum function may reveal some information about its input, that is K_{A-B}, whereas the cryptographic hash function has the property that the output of the hash function does not reveal any information about its input.

So, from $H(R, K_{A-B})$, one is not able to recover any information about the secret K_{A-B}. For this reason, this H(.) has to be a hash function. It cannot be a checksum function. We now discuss a variant of protocol ap4.0. As before, suppose a shared secret symmetric key K_{A-B} exists between Alice and Bob.

This shows a variant of ap4.0. Alice sends a message, 'I am Alice', to Bob. Then Bob selects a nonce, say R, and then encrypts the nonce to get $K_{A-B}(R)$ and then sends $K_{A-B}(R)$ to Alice. Alice then decrypts the message to get the nonce R and sends it to Bob. So, here the encryption and decryption are done by different entities.

So, in the previous protocol, R just sent the nonce, and Alice encrypted it. Here, Bob encrypts the nonce and sends it to Alice, and Alice decrypts the nonce. Alice decrypts the encrypted nonce and sends the recovered nonce to Bob. So, this protocol also achieves authentication. So, the fact that Alice is able to decrypt this message shows that she's in possession of the secret K_{A-B}, which only Alice knows other than Bob.

Hence, this protocol achieves authentication. Can we modify this protocol so that it uses cryptographic hash functions instead of encryption? Can we replace this with a hash, say, $H(R, K_{A-B})$? If Bob sends $H(R, K_{A-B})$ to Alice, then she cannot find R because of the one-way nature of cryptographic hash functions. From the hash value, Alice is not able to find the input, which was used to create that hash value.

So, because of the one-way property of cryptographic hash functions, we cannot replace this encryption process with a cryptographic hash function. So now, we discuss authentication using synchronized clocks. Recall that ap4.0 is a correct protocol, but it has the shortcoming that it uses three messages. Now, suppose Alice and Bob have a shared

symmetric key K_{A-B}, and they also have approximately synchronized clocks. So, in a network, every node has a measure of time.

So, for example, it may have a crystal oscillator, which is used to keep track of time. But these clocks of different nodes are not synchronized in general. For example, the local time at Alice might be 4 pm, but the local time at Bob might be 5 pm. So, these clocks are not synchronized in general in a network. But now we are assuming that Alice and Bob have approximately synchronized clocks.

So, for example, if the clock at Alice is, say, 4 pm, then the clock at Bob might be 4 pm + 1 millisecond. So, they have approximately synchronized clocks. There are synchronization protocols available which can be used to synchronize the clocks of different nodes in a network. Can we now modify ap4.0 using this synchronization so that it uses only one message instead of three messages? This shows a protocol that uses only one message.

Alice sends a message, 'I am Alice', along with K_{A-B}(timestamp) to Bob, where the timestamp is the current time at Alice. And since Alice and Bob have approximately synchronized clocks, the clock at Bob is also roughly timestamped. Bob decrypts the encrypted timestamp and checks whether it is within an acceptable clock skew. For example, if the timestamp received by Bob is, say, 4 pm, then Bob may accept that timestamp. Assuming that the local time at Bob is 4 pm + 1 millisecond, if the timestamp sent by Alice is 4 pm + 0.5 millisecond, then Bob may accept the timestamp.

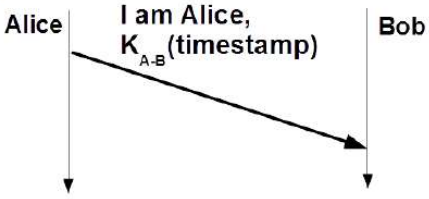
But if the received timestamp is 3.59 pm, in that case, Bob may reject the timestamp because it is off from the local time of Bob by a large margin. So, Bob checks whether the encrypted timestamp is within an acceptable clock skew, and if yes, then Bob accepts the timestamp. So, this protocol achieves authentication. So, there are some possible attacks on this protocol. Consider an eavesdropping intruder, say Trudy, who uses this K_{A-B}(timestamp) to impersonate Alice within an acceptable clock skew.

So, Trudy records this message and then immediately afterwards, Trudy tries to authenticate themselves as Alice to Bob. So, the crucial point is that since Trudy immediately sends another message to Bob claiming to be Alice, the timestamp is still fresh. So, not much time has elapsed, so the clock skew checked at Bob passes if this is done very quickly by Trudy. So, the difference is Bob remembers all the timestamps used by Alice for authentication until they expire, that is, until the timestamps are old enough

such that the clock skew check would consider them invalid. So, this defense works against this attack.

Now, suppose there are multiple servers for which Alice uses the same secret key K_{A-B}. Then, an eavesdropping intruder who acts fast enough could use K_{A-B}(timestamp) to impersonate Alice to a different server. So, Alice performs authentication at one server. Trudy overhears that message and Trudy gets K_{A-B}(timestamp) and immediately sends that to a different server to try to impersonate themselves as Alice. So, a defense against this is, instead of sending K_{A-B}(timestamp), Alice sends K_{A-B}(Bob | timestamp), where 'Bob' is the server's identifier.

- Defence:
  - Instead of sending K_{A-B}(timestamp), Alice sends K_{A-B}(Bob|timestamp), where "Bob" is the server's identifier



So, if Trudy sniffs this message and tries to authenticate themselves to some other server, in that case, this message won't work for authenticating to that other server because, instead of Bob, one will need to put the server name of that other server. So, for this reason, this protocol defends against this attack. So, we have added the server's name to the message, which is encrypted. So, for this reason, this attack, which is described here, does not work. So, this defense works against this attack.

Now, consider this protocol in which Alice sends 'I am Alice, K_{A-B}(timestamp)' to Bob. Suppose we want to use a hash function instead of encryption. So, Alice sends 'I am Alice, H(timestamp, K_{A-B})' to Bob. So, how can Bob verify that H(timestamp, K_{A-B}) is reasonable? Bob cannot reverse the function H(timestamp, K_{A-B}) to get timestamp and K_{A-B} because this is a one-way function.

- Suppose we want to use a hash function instead of encryption:
  - Alice sends "I am Alice, H(timestamp, K_{A-B})" to Bob
- How can Bob verify that H(timestamp, K_{A-B}) is reasonable?
  - Bob can find $H(t, K_{A-B})$ for all values of t within the acceptable clock skew and see if one of them matches H(timestamp, K_{A-B})

So, Bob can find H(t, K_{A-B}) for all values of t within the acceptable clock skew and then see if one of them matches H(timestamp, K_{A-B}). If one of them matches this, then Bob accepts the authentication request. But this has a shortcoming: when the clock resolution is high, for example, when time is measured in units of microseconds, then this is very computationally expensive because there would be a large number of possible timestamps, t, within the acceptable clock skew. So, this would incur a very large overhead. So, can we modify this protocol to make it less computationally expensive?

So, to modify this protocol, Alice sends the message, 'I am Alice, timestamp, H(timestamp, K_{A-B})' to Bob. So, Bob just uses this timestamp sent by Alice, appends the value K_{A-B} to it, finds its hash, and checks whether that matches the hash sent by Alice. So, this protocol also achieves authentication and is computationally more efficient than the previous protocol because Bob doesn't have to try out different values of t to find a value of H(t, K_{A-B}) that matches the value sent by Alice. Now, there is another attack known as the server database reading attack. Consider protocol ap4.0, which we discussed previously.

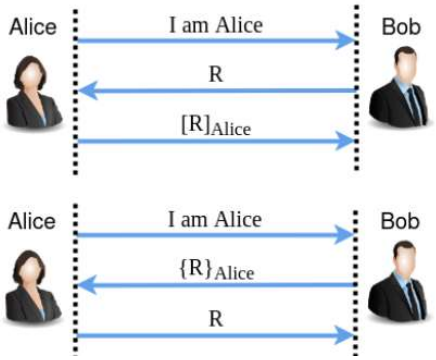
Someone who reads the database at Bob and obtains the secret K_{A-B} can later impersonate Alice. So, if someone reads the database at Bob, they'll obtain the secret K_{A-B} and then can authenticate themselves to Bob as Alice because they know K_{A-B}. So, when Bob sends a nonce, say R', they are able to find out K_{A-B}(R'). What is an example in which an intruder might read the server database? Suppose, for example, that there are many servers for which Alice uses the same key K_{A-B}.

The administrator of one of those servers may not have carefully secured the server, so some intruder might read the server database and obtain the secret K_{A-B}. Apart from ap4.0, the other protocols we discussed earlier also rely on a secret key K_{A-B}, shared between Alice and Bob. If this secret is leaked because of a server database reading attack, then the protocol fails. So, how can we defend against the server database reading attack? So, we can do this using public key cryptography.

So, suppose Alice and Bob don't have a shared symmetric key, but Alice has a public key and private key pair with the public key known to Bob. How can Alice authenticate herself to Bob? This shows two possible protocols that can be used to authenticate Alice to Bob. In the first figure, that is this one, Bob sends a nonce to Alice, that is R, and then Alice signs the nonce R, that is transforms R using a private key. This [R], this shows the signed nonce.

So, recall that earlier we denoted the signed nonce by this, $K_A^-(R)$. This is an alternative notation for the same quantity, $K_A^-(R)$. So, this is the nonce signed by Alice using a private key. Then Alice sends the signed nonce to Bob, and Bob applies the public key to the signed nonce and checks whether it matches the sent nonce R or not. So, this protocol works and achieves authentication correctly. In the second figure, Bob selects nonce R and encrypts it using Alice's public key to get $\{R\}_{Alice}$.

- In first fig., Bob sends a nonce R to Alice; Alice signs R (i.e., transforms R using her private key) to get $[R]_{Alice}$, which she sends to Bob
- In second fig., Bob selects a nonce R and encrypts it using Alice's public key to get $\{R\}_{Alice}$, which he sends to Alice; Alice decrypts it and sends result to Bob
- Which of these protocols, if any, defend against the server database reading attack?
  - ❑ Both; an intruder who reads database at Bob can only read Alice's *public* key
  - ❑ No confidential information stored at server



So, earlier we denoted this by $K_A^+(R)$. So, we are denoting it by this quantity now, $\{R\}_{Alice}$. So, Bob encrypts the nonce using Alice's public key to get this $\{R\}_{Alice}$, which he sends to Alice, then Alice decrypts it and sends the result to Bob. And Bob verifies whether that is equal to the nonce whose encrypted version he had sent. So, both of these protocols defend against the server database reading attack because there is no secret stored at Bob.

An intruder who reads the database at Bob can only read Alice's public key. There is no confidential information stored at the server. So, Alice's public key is anyway not a secret. So, reading the server's database will not compromise this protocol. Notice that if someone manages to read the database at Alice, then the protocol will be compromised because then Alice's private key will be leaked.

In that case, this protocol won't work. But we are talking about the server database reading attack against which these protocols are secure. So, we'll continue our discussion of authentication in the next class. Thank you.