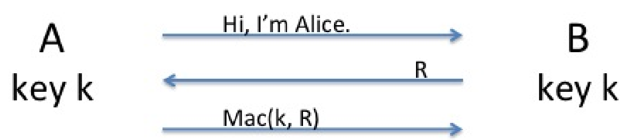
ISA 562: Information Security, Theory and Practice

Lecture 4

1 Identification protocols

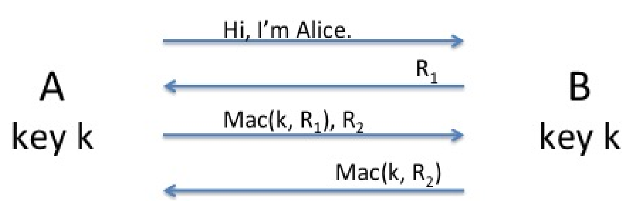
Now that we know how to authenticate messages using MACs, a natural question is, how can we use MACs to prove that we are who we say we are? For now, we will still assume shared keys, so the question we're asking is, if Alice and Bob share a key k, and they wish to communicate over the Internet, how can they each convince the other that they are, indeed, the holders of that shared key?

Our first thought might be to have Alice send an authenticated message, such as $m =$"Hi! I'm Alice. (I swear.)", $t = \mathsf{MAC}(k, m)$. But this is clearly subject to a replay attack: after listening in on their channel, Eve can simply send the same (m, t) pair tomorrow, and convince Bob that she is Alice. Instead, we use the following simple, interactive protocol, where $R \leftarrow bitset^n$ is a random challenge string, sometimes called a "nonce".

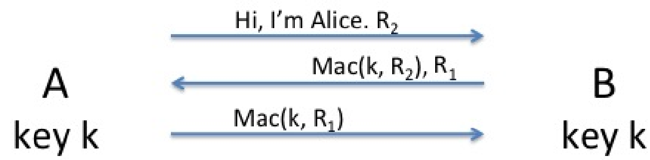


Because R is chosen at random from a large space, the probability that the same challenge string is ever used twice is negligibly small. So, even if Eve records this conversation and tries to use it tomorrow, she'll be given a new challenge string and will be unable to come up with a MAC that correctly verifies on the new challenge.

This is a secure way for Alice to convince Bob that she holds their shared key. But what if they want both need to be convinced? The obvious thing is to run the protocol twice, once in each direction. After removing 2 messages that are redundant, we have the following secure protocol.



It is tempting to think that we can shave off another message round, by having Alice send R_2 immediately, when she first says hello. The following protocol is ***insecure***, as we will show next.



This protocol is subject to a type of attack that is called a *reflection* attack. Suppose Eve wants to impersonate Alice. She sends the first message in the above protocol, and receives Bob's response: $\mathsf{MAC}(k, R_2), R_1$. Eve is supposed to compute $\mathsf{MAC}(k, R_1)$, but, of course, without knowing k, she's unable to do this. Instead, she pauses, and opens a 2nd, parallel session with Bob, sending Alice's first message again, but this time "reflecting" Bob's challenge right back at him: "Hi, I'm Alice. R_1". If Bob isn't maintaining state and checking for such an attack, then he will dutifully reply by authenticating R_1 for Eve, giving her exactly what she needs to go back and correctly finish the first session that she started. That is, in the 2nd session, Bob will send $\mathsf{MAC}(k, R_1), R_3$. Of course, to finish authenticating as Alice in the 2nd session, Eve would have to find a tag for R_3, which she cannot do. But she doesn't have to – instead she just drops the 2nd session, and uses $\mathsf{MAC}(k, R_1)$ as her response to Bob in the 1st session, which he accepts as valid.

It is worth considering why the 4 round protocol described previously is secure, and isn't subject to this attack. Intuitively, the reason the reflection attack fails in that protocol is because Eve is required to prove herself before Bob replies with any tags. In contrast, in this faulty construction, Bob will authenticate whatever message Eve gives to him, without first verifying that Eve is someone that he trusts. A good principal, then, is that the person initiating the session should prove themselves first; this way, someone malicious can't gain anything by initiating multiple sessions.

Even though we have a perfectly good 4-round solution, it might still be desirable to shave off another round from the protocol. There are a few possible ways to remove the reflection attack, without requiring 4 rounds. The first is to ensure that the random challenges have distinct formats. For example, Alice might prepend "Alice" to her challenge, while Bob pre-pends "Bob." This prevents Eve from using Bob's response to her challenge in one session as her own response to his challenge in another session, since the formatting will be incorrect. A second option is to maintain 2 MAC keys, one for each direction; then, Bob's MAC on R_1 is not the same as Alice's MAC on R_1.

2 Key distribution centers

There are two major problems with trying to scale out private key systems, such as the ones we've been discussing for encryption, MACs, and identification protocols. The first problem is that, if there are n users in the system, then every user would have to maintain n keys, one for each other party in the system. The total storage required is then $O(n^2)$. The other problem is that it becomes very difficult to distribute keys to all users: imagine if you had to visit a physical site anytime you decided to use your credit card at a new website. This inconvenience would be a pretty bit hurdle for online shopping and donating!

We will see shortly that public key cryptography helps solve both of these issues, but we first show a private key (i.e. shared key) mechanism for alleviating these problems. The idea is that each domain, such as a university, a corporation, a government agency or perhaps an Internet service provider, will establish a key distribution center (KDC). Each user will have to go in person to prove their identity and establish their shared key with the KDC. In a university system, this might be done when you first enroll at the university and pick up your ID card. Users only store that one single key. When Alice wants to talk to Bob, who is in her same organization, she contacts the KDC, authenticates in the manner described in the previous section, using her key k_a, and request a session key – a short-term key that can be used between her and Bob for the next short period of time. The KDC will generate this key, which we'll call k_{ab}, and encrypts a copy for Alice using the key she shares with the KDC: $\mathsf{Enc}(k_a, k_{ab})$. Additionally, the KDC encrypts a copy of k_{ab} for Bob, using the key that they share together: $\mathsf{Enc}(k_b, k_{ab})$. The KDC could send this directly to Bob, telling him that it is intended to be used with Alice, but more common is for the KDC to simply give this to Alice, allowing her to reach out to Bob herself. The encryption of k_{ab} under Bob's key, which is given to Alice, is often called a *ticket*.

This approach has helped in two ways. First, now Alice and Bob never have to meet in person to exchange a key. Secondly, each of them only has to keep track of a single key. This might sound less important than the first issue, but realize that in a large system, keys are likely to be revoked or changed, and asking every user to keep track of this is unrealistic. Of course, the KDC still needs to maintain n different keys, but doing this in one central place is far easier than having all users in the system trying to maintain the full list of keys.

Of course there are also some major drawbacks to this approach. The most obvious is that the KDC holds the keys to the castle: they know everyone's key, even the short-lived session keys, and they can decrypt any conversation, and impersonate any user in the system. This makes the KDC a high profile target, and if anything happens to the KDC, the whole system is really in trouble.

Another obvious problem is that this solution still doesn't scale to beyond the level of a mid-sized organization. What happens when Alice wishes to talk to someone that is not at the same university as she is? In this case, we can have different KDCs establish shared keys with each other, the same as a user establishes a shared key with a KDC. For example, if Alice had a key with the GMU KDC, and Bob had a key with the UMD KDC, Alice can approach her own KDC and ask to be put in touch with the UMD KDC. If those two KDCs share a key, then this can be done precisely as described above. Then, once Alice has established a session key with UMD KDC, she can ask that KDC to create a ticket for her to communicate with Bob. What happens if the UMD KDC and GMU KDC don't have a shared key themselves? Well, Alice just needs to find some path of connected KDCs from her own to Bob's, and she can request a sequence of session keys along this path until she's in touch with Bob. For the moment, we put off the question of how Alice can find such a path, as well as how she knows she can trust every entity along that path. We will revisit this when we talk about the public key infrastructure, which faces many of the same issues.

3 Password security

3.1 Password storage

If a password database becomes compromised, a malicious party can use the passwords to gain repeated access to the site in the future. Furthermore, because many users re-use their passwords across multiple sites, if one server becomes compromised and exposes their database, it is quite likely that the malicious party can use the recovered passwords to gain access to other sites. Encrypting the password database is not helpful here, because we're concerned about an adversary that fully compromises the server, as opposed to one that simply eavesdrops on traffic to and from the server, so unless the decryption key is stored on a separate server from the encrypted passwords, which would make password verification difficult, we have to assume that the malicious party will also have access to the decryption key.

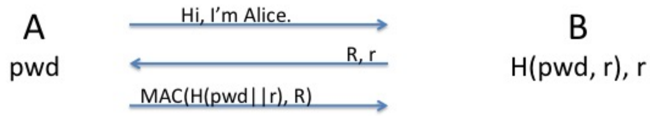
Instead, we might consider storing the hash of each password. Because passwords are low-entropy, an adversary can still mount an offline-attack (also called a dictionary attack) in which he tries every possible password, looking for the one that gives the correct hash, but at least this requires the adversary to do some work to recover the values. However, note that if passwords are re-used across sites, the adversary would only have to compute the hashes of popular passwords one time, and then can quickly check any newly compromised password database against the pre-computed hashes. To make the adversary perform this offline attack for every newly compromised database, servers storing passwords use a random *salt* (sometimes called a nonce), and hash the password together with the salt before storing it. For example, when a user sets their password to pwd, the server chooses a random value r, and stores $(\mathsf{userId}, H(\mathsf{pwd}||r), r)$. Then, when a user sends pwd to login, the server looks up r, computes $H(\mathsf{pwd}||r)$, and compares the result with the stored value. Because every server will use a different salt value when storing the password, the adversary cannot rely on pre-computed hash values for determining which user password corresponds to each hash value.

This can be strengthened even further, using repeated hashing: instead of computing $H(\mathsf{pwd}||r)$, the server can store $H^{\kappa}(\mathsf{pwd}||r)$, where $H^1(\mathsf{pwd}||r) = H(\mathsf{pwd}||r)$, and $H^i(\mathsf{pwd}||r) = H(H^{i-1}(\mathsf{pwd}||r))$. This requires the adversary to compute κ hashes for each password, making the offline-attack even more costly. It also means that the honest server will have to compute κ hashes for each login attempt, but this is not so bad, since we can expect that the number of login attempts will be much lower than the number of passwords stored in the database. The cost of computing κ hashes is hardly noticeable to the end-user. Another issue arises though: adversaries can purchase special hardware for performing these hash computations, gaining an advantage over the server. Some special purpose machines can perform SHA-256 hashes $10^5\times$ faster than a standard $x86$ architecture. Very recently, researchers have been designing *memory hard* hash functions, which require a lot of on-chip memory. Because memory consumes a lot of chip area, it is hard to also include a large number of hashing circuits on specialty chips that require a lot of memory.

3.2 Authenticating with a password

If Alice is trying to prove herself to Bob by demonstrating knowledge of her password, how should she send this value? We assume, of course, that Alice and Bob do not share a cryptographic key, or there would be no reason to rely on a password at all. So Alice cannot encrypt her password, and sending it in the clear is undesirable, since an eavesdropper can then immediately use the same value, at this and at other sites, to masquerade as Alice. Alice could ask Bob for the salt r that was used in storing the password, and send $H(\mathsf{pwd}||r)$ to the server to authenticate herself. This would require the adversary to perform an offline attack to recover the password before using it at other sites, but note that the adversary would immediately know Alice's login credentials for Bob's server, since the salted password suffices for that task: the adversary doesn't need to recover pwd if he knows $H(\mathsf{pwd}||r)$ for the appropriate salt r.

We give two constructions that would force the adversary to perform an offline attack before being able to impersonate Alice to Bob. The first is to use the authentication protocols described in Section 1, but using $H(\mathsf{pwd}||r)$ as the MAC key in the challenge response.



That is, after Alice says hello to Bob, Bob replies with a challenge nonce, R, and the salt r used in storing Alice's password. Alice computes $F(H(\mathsf{pwd}||r), R)$ and sends this to Bob. (Here F is a secure block cipher, such as AES.) Bob uses the stored hash of Alice's password, $H(\mathsf{pwd}||r)$, to compute the same MAC value on R, and then verifies that it matches the value sent by Alice. Note that for an eavesdropper to later forge a correct response to a new challenge, he would first have to brute-force the password, trying all values of pwd and the corresponding values $H(\mathsf{pwd}||r)$, to see which salted password gives the correct MAC on R. After recovering pwd in this way, the adversary can correctly compute the MAC on a new challenge during the next login session.

Another approach to ensuring that an eavesdropping adversary has to perform an offline attack before masquerading as the user is Lamport's password hashing scheme. The server stores $(H^{\kappa}(\mathsf{pwd}), \kappa)$, and when Alice wants to sign in, Bob sends her the current value of κ. Alice sends $y = H^{\kappa-1}(\mathsf{pwd})$, and Bob verifies that $H(y) = H^{\kappa}(\mathsf{pwd})$. If so, then he accepts Alice's authentication, and stores $(y, \kappa - 1)$ in place of $(H^{\kappa}(\mathsf{pwd}), \kappa)$. We could also combine this construction with a salt in the natural way. Note that an eavesdropper cannot recover pwd without running a dictionary attack.

4 Public key cryptography

4.1 Algebraic groups

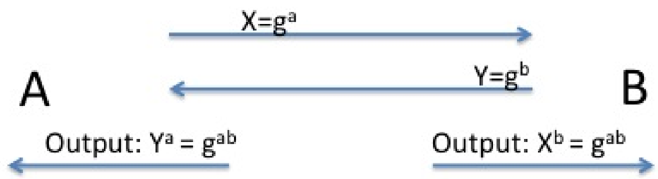
We don't formally define algebraic groups, but we give a few examples. $\mathcal{Z}_N^*$ is the set of all integers between 1 and $N-1$ that are relatively prime to N. For example, $\mathcal{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$ and $\mathcal{Z}_{15}^* = \{1, 2, 4, 6, 7, 8, 9, 11, 13, 14\}$. These groups are *closed* under multiplication, modulo N, which means that if we multiply two elements and take the result modulo N, we are always left with another element in the group. Letting $\times$ denote this operation, we use g^i to denote $\underbrace{g \times \cdots \times g}_{i \text{ times}}$. For example, looking at $\mathcal{Z}_{15}^*$ and taking $g = 7$, we have $g^1 = 7$, $g^2 = 49 \pmod{15} = 4$. $g^3 = 7 \times 7^2 = (7 \times 4) \pmod{15} = 28 \pmod{15} = 13 \pmod{15}$, and $g^4 = (7 \times 7^3) \pmod{15} = 7 \times 13 \pmod{15} = 91 \pmod{15} = 1 \pmod{15}$. Since $7^5 = 1 \times 7 = 7$, it is easy to see that at this point, the powers of 7 will repeat themselves. We say that 7 generates the sub-group $\{1, 7, 4, 13\}$. Taking another example, this time with $\mathcal{Z}_7^*$, and looking at the sub-group generated by 3, we have $3^1 = 3$, $3^2 = 2$, $3^3 = 6$, $3^4 = 4$, $3^5 = 5$, $3^6 = 1$, and after this the powers of 3 repeat themselves. As we can see, 3 generates all of $\mathcal{Z}_7^*$, so we call it a *generator*.

4.2 Diffie-Hellman key exchange

Let g be some generator of an algebraic group, and let ϕ be the number of elements in the group (also referred to as the *order* of the group)[1]. Alice and Bob do not have a shared key; their goal is to agree on a key, knowing that Eve might see every message that they each send. The protocol

[1]Technically, we require that the number of elements in the group is prime. This wasn't the case for either of the examples above, but we do not worry about how to find such groups here.

begins with Alice choosing an integer a, uniformly at random, such that $0 \leq a < \phi$, and Bob doing the same to select some integer $0 \leq b < \phi$.



The protocol is secure as long as the *decisional Diffie-Hellman* problem is hard: given g^a and g^b, it is hard to distinguish g^{ab} from a random group element. Technically, depending on what they will later use the key for, the parties might need to hash the result to get a key of the proper form.

4.3 Defining public key encryption and signatures

A similar idea gives us a method for building public key encryption and digital signatures. These cryptographic schemes are similar to the encryption schemes and MAC schemes we studied earlier, but instead of a single shared key, keys come in pairs: a public key, and secret key. Only the person holding the secret key in a public key encryption (PKE) scheme can decrypt a ciphertext, and only the person holding the secret key in a digital signature scheme can sign a message. On the other hand, *anybody* holding public key in the PKE scheme can encrypt messages, and anybody holding the public key in a signature scheme can verify that a message was properly signed. The secret keys are never intended to be shared, so if Alice and Bob want to exchange messages, each uses the other's public key to encrypt, and each uses their own secret key to decrypt.

The semantics of a PKE scheme are as follows. As before, the scheme is associated with a keyspace $\mathcal{K} = \mathcal{K}_1 \times \mathcal{K}_2$, a message space $\mathcal{M}$, and a ciphertext space $\mathcal{C}$. As per the previous discussion, the keyspace contains pairs of elements: the secret key is drawn from $\mathcal{K}_1$ and the public key from $\mathcal{K}_2$.[2] Of course, if the scheme is secure, these keys must be drawn in some correlated manner. Otherwise, there would be nothing stopping an adversary from drawing his own secret key, independent of Alice's public key, and using it to decrypt messages sent to Alice.

$\mathsf{Gen}(1^\kappa) \rightarrow (\mathsf{pk}, \mathsf{sk})$: takes as input a security parameter κ, and outputs a pair of keys.
$\mathsf{Enc}(\mathsf{pk}, m) \rightarrow c$: on input a public key $\mathsf{pk} \in \mathcal{K}_2$, and a message $m \in \mathcal{M}$, outputs a ciphertext $c \in \mathcal{C}$.
$\mathsf{Dec}(\mathsf{sk}, c) \rightarrow m$: on input a secret key $\mathsf{sk} \in \mathcal{K}_1$, and a ciphertext $c \in \mathcal{C}$, outputs a message $m \in \mathcal{M}$.

Signature schemes are defined similarly. We refer to the verification key as vk instead of pk.

$\mathsf{Gen}(1^\kappa) \rightarrow (\mathsf{vk}, \mathsf{sk})$: takes as input a security parameter κ, and outputs a pair of keys.
$\mathsf{Sign}(\mathsf{sk}, m) \rightarrow \sigma$: on input a secret key $\mathsf{sk} \in \mathcal{K}_1$, and a message $m \in \mathcal{M}$, outputs a signature σ.
$\mathsf{Vrfy}(\mathsf{vk}, \sigma, m) \rightarrow b$: on input a verification key $\mathsf{vk} \in \mathcal{K}_2$, a message $m \in \mathcal{M}$, and a signature σ, outputs a message a bit indicating whether the signature is valid.

[2]Typically the secret key and the public key are drawn from the same space, though, this doesn't have to be the case.

4.4 Revisiting authentication

In the most common setting that we encounter on the Internet, a client (Alice) and server (Bob) each want to authenticate themselves to the other, and establish a shared key. The server is a high-profile, well known entity, such as a bank or an online merchant, and the client is someone that holds a password, but has no knowledge of cryptography and, in particular, does not hold any key pairs for signing and encrypting messages. In this case, Bob will use his key pair to sign his message in the Diffie-Hellman key exchange. Alice's browser can verify the signature using the public verification key known to belong to Bob. After establishing the shared key, Alice can use it to encrypt her password, thereby convincing Bob that she is who she claims to be. They can then continue to use their shared key. The question of how Alice knows that she is using Bob's correct verification key is a tough one, which we discuss in the lecture slides on the public key infrastructure.