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 $m = \text{“Hi! I’m Alice. (I swear.)”}$, $t = \text{MAC}(k, m)$. 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 $R \leftarrow \text{bitset}^n$ 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: MAC($k, R_2$), $R_1$. Eve is supposed to compute 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 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 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: \( \text{Enc}(k_a, k_{ab}) \). Additionally, the KDC encrypts a copy of \( k_{ab} \) for Bob, using the key that they share together: \( \text{Enc}(k_b, k_{ab}) \). 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

TBD.

4 Public key cryptography

4.1 Diffie-Hellman key exchange

TBD
4.2 Defining public key encryption and signatures
TBD