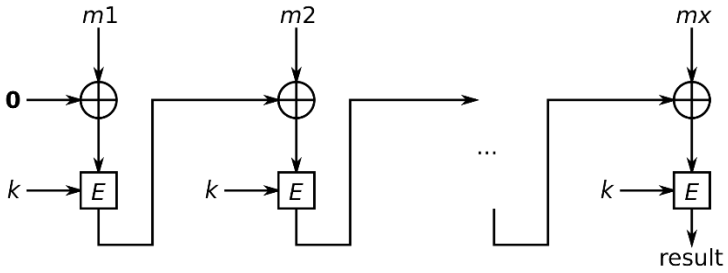
Justin Lai

Ben Lirio

MAC

Description

My algorithm is a CBC-MAC implementation. Here is a diagram from Wikipedia.



The algorithm first pads a message so that it is a multiple of the block size. It then does the following. For each block, xor the block with the current tag value (using 0 in the base case). Take this xored value and run it through the encryption provided in MacSkeleton. The output of this encryption function becomes the new current tag. Once there are no blocks left, output the most recent tag.

This CBC-MAC implementation is secure for any fixed length messages as long as both parties know the length of the message in advance. A possible extension that would make this algorithm secure for variable length messages is described in the vulnerabilities section.

Justification

This algorithm is correct because if follows the spec of CBC MAC. We are assuming that the CBC MAC spec is secure.

Vulnerabilities

The algorithm does not use a counter to prevent replay attacks. For example, an adversary can record a message from Alice to Bob on Day 1, then on some later time the adversary can replay the authenticated message. One countermeasure is to include a timestamp.

Another vulnerability is that an attacker can append an additional block so that the xor of the tag and the new block is equal to the first block of another message m'. One countermeasure is to include the number of total blocks in the first block. The verify function would also have to be changed so that it checks to make sure that the number of blocks is ok.

Assumptions

This algorithm assumes that the functions provided in MacSkeleton work as expected. In addition, we assume that the adversary is PPT. In addition, we assume that one-way-hash functions exist for PPT adversaries.

Default Message Tag

29BA1525FA2E2E390574CEAB96BF3F3F

PKI

Vulnerability

If Mallory gives Alice Bob's old public key instead of the new public key, and then Mallory gives Cindy Bob's new public key, then Cindy can perform a man in the middle attack as follows.

When Alice sends a message to Bob, she encrypts it with Bob's old public key. Cindy then intercepts this message, decrypts it, and re-encrypts it with Bob's new public key. Cindy then forwards this message to Bob, who uses his new private key to decrypt the message.

Alice will not be suspicious because she has no way of knowing that Bob’s public key has changed. Bob will not be suspicious because, from his perspective, Alice is sending him messages signed by his new private key.

Countermeasure

One countermeasure is to add a timestamp on the certificate C. When a user wants to send a message, before using the public key supplied by Mallory, the user should check to see if the certificate was signed within the last 24 hours. If it was not, the user should not trust the public key supplied by Mallory.
This means that sometime in the next 24 hours after Bob changes his public key, the old public key will no longer be valid.

RSA

How modular exponentiation is realized more efficiently in practice

There are several ways to optimize modular exponentiation in practice. First is using theory, we can improve the runtime by using mathematical shortcuts. For example, using binary exponentiation is much faster than standard methods. Second, we can improve runtime by selecting an efficient language. In this project, we are using Python, but since RSA is bottlenecked by efficiency, implementing it in a language like C or assembly is preferable. Another option is to create custom hardware to compute the RSA algorithm.

Exponent Size

Using small exponents in practice is ok and preferable when considering efficiency. Because distribution of keys comes from the choice of p and q, using a small m will not reduce the key space.

Note: There is some debate about using small exponents especially when messages are not padded properly, but in general using a small exponent is preferable.