Exercises for Network Security
1. Cryptography

Emmanouil Vasilomanolakis & Carsten Baum, DTU

February 14, 2024

Exercise 1.

(Unconditional Security) During the lecture you have learned that encryption algorithms can be broken if one can guess the secret key. There exists a symmetric key encryption scheme for which this is not the case - i.e., which is secure against such attacks. This is usually called the *One-Time Pad*. It is defined as follows:

Key Generation To generate the key k, flip a fair coin. If it is heads, then set $k = 0$, else set $k = 1$.

Encryption Encrypt the message bit m as follows: compute $c = m + k \bmod 2$ and output c.

Decryption Decrypt the ciphertext bit c as follows: compute $m' = c + k \bmod 2$ and output m'.

1. Show that this scheme actually decrypts to the right plaintext.
2. What can go wrong if you use the same key for two encryptions?
3. Can you use this scheme to also encrypt a long string of bits? How do you have to modify it? What is the disadvantage of this encryption scheme, in terms of practicality?

Bonus: We can show that an encryption cannot leak any information to an attacker without the key. For this, one can show that independent of m being 0 or 1, the ciphertext c will always be either a 0 or a 1 with probability 1/2. Therefore, without knowledge of the key, both plaintexts are equally likely. Can you prove this?

Exercise 2.

(AES) Assume you obtain the ciphertext

$$U2FsdGVkX1 + EqkWb5RzDhZyL6gSu/2hCuWRFkBFaO2U =$$

You know that it is an AES-128 ECB encryption of an UTF-8-encoded text that starts with the four characters "HELP". The ciphertext is encoded in Base-64.

1. Assuming that you have no further information about the key, why can you not simply run a brute-force attack to recover the rest of the message?
2. Assume that you have additionally learned that the key has the first 96 bits set to 0. Why is an attack now plausible[a]?

3. Assume that you learn that only 4 bits of the key are set to 1, while the remaining ones are set to 0. Does this mean that an attack is feasible?

4. The ciphertext is not given as a string, but encoded in Base64. Investigate what Base64 is. Why can it be advantageous to encode a ciphertext in Base64 instead of sending it directly over the channel?

5. The message mentioned in this exercise has been encrypted using the *openssl* tool available on any Linux system. Assume that the key has been derived from "crypto" (using the $-k$ option). Attempt to decrypt it!

[a] You can e.g. look at `https://bench.cr.yp.to/results-stream.html` and identify how many cycles it takes for one core on a modern Apple M1 processor (the standard processor in Mac Books) to encrypt/decrypt one byte with AES-128, from which you can derive an upper-bound on the runtime for a whole 128-bit block. This may aid you to estimate the runtime of an attack.

Exercise 3.

(Hashing $\neq$ Security) You register with the text messaging service Helo who stores your phone number as an identity when you register. To check who of your friends is available on Helo, the company uses the following process:

Setup The company chooses a cryptographic hash-function H. It gets hardcoded into the messaging app software of Helo. The company also sets up a server with a file "users" that is initially empty.

User registration The messaging app sends your phone number tel to the server of Helo. The server then stores $H(tel)$ in the file "users".

Friend discovery The app hashes each phone number tel_i in the phone book of the user and sends $H(tel_1), \ldots, H(tel_n)$ to the server of Helo. The server then responds with the list of indices $1, \ldots, n$ such that $H(tel_i)$ was in the file "users".

Helo claims that, using this technique, they will never learn the phone numbers in your phone book that are not registered with their service. Show that this is not the case, by explaining a simple program that could extract phone numbers from the messages that their server obtains. As example, assume all phone numbers have the length and format as in Denmark.

Exercise 4.

(The RSA cryptosystem) A famous public-key cryptosystem is the so-called RSA encryption scheme. We saw in the lecture that it works as follows:

Key Generation Choose two different primes $p, q > 2$. Let the public key be $N = p \cdot q, e = 5$ and the private key be $p, q, d = e^{-1} \bmod (p-1)(q-1)$.

Encryption To encrypt the message $m \in \mathbb{Z}_N$, compute and output $c = m^e \bmod N$.

Decryption Output $m' = c^d \bmod N$.

1. Test out that RSA is correct. For example, choose $p = 17, q = 13$, $m = 4$ and do the calculations for encryption and decryption. You can do all operations, also including the computation of d, using e.g. Wolfram Alpha.

2. What happens in the RSA cryptosystem if the message is 0 or 1? More generally, what happens if the attacker has an idea what the encrypted message could be?

3. Suggest how the problem described in 2 can be avoided.

Bonus: Prove that RSA is correct, using Euler's totient theorem.

Exercise 5.

(RSA Signatures in practice) Let us assume that you want to implement an application where

- The sender must be able to deliver messages $m_1, \ldots$ with integrity.
- Anyone must be able to verify that the sender sent the messages $m_1, \ldots$.

Naturally, you pick RSA signatures for the job:

- Let sender have a secret signing key sk and give every potential receiver a verification key vk.
- Whenever the sender sends a message m_i, it also sends $\sigma_i = Sign_{sk}(m_i)$.
- Any receiver only accepts a message m_i if it is accompanied by a signature σ_i such that $Ver_{vk}(m_i, \sigma_i)$ is correct.

1. Assume your messages are only a bit while your sender sends multiple messages. How can integrity become a problem?

2. Assume that your sender and receiver can keep a state. How can you fix the problem using a counter?

Solutions:

Exercise 1:

1. This can easily be seen because modulo 2, we have that $k + k = 0 \bmod 2$ (you can test this for both possible values of k). Then $c + k = m + k + k = m \bmod 2$.
2. If you have two ciphertexts $c_1 = m_1 + k \bmod 2$ and $c_2 = m_2 + k \bmod 2$, then you can compute $c_1 + c_2 = m_1 + k + m_2 + k = m_1 + m_2 \bmod 2$, which leaks the sum of the plaintexts.
3. Let's say the message consists of a string of bits $m_1 m_2 \dots m_\ell$. The idea is to use a key string $k_1 k_2 \dots k_\ell$ of equal length to generate a ciphertext $c_1 c_2 \dots c_\ell$ of length ℓ. To encrypt, we compute $c_i = m_i + k_i \bmod 2$ and we decrypt equivalently by computing $c_i + k_i \bmod 2$. Unfortunately, this means that the key has length ℓ, i.e is as long as the message to be encrypted.

Exercise 2:

1. The brute-force attack would have to test all 2^{128} possible keys for AES-128, which is computationally infeasible.
2. This means that we now only have to test out 2^{32} keys. To see how long this would take, let us make the following observations:
   - it takes around 16 cycles per byte[1] to encrypt/decrypt AES-128 on a modern Apple M1 processor (the standard processor in Mac Books) per core. For a whole ciphertext of 16 bytes, that is around 256 cycles.
   - The processor runs at 3200 MHz, i.e. it performs 3.200.000.000 cycles per second.
   - Therefore, one core of the processor can test approximately 12.500.000 ciphertexts per second.

   Using one core we should be able to find the right key in around 343 seconds, or less than 6 minutes.
3. We can use the binomial coefficient (see `https://en.wikipedia.org/wiki/Binomial_coefficient`) to estimate the number of possible keys, as this is equivalent to all subsets of $\{1, \dots, 128\}$ of size 4. This means there are only $\binom{128}{4}$, or around 10.6 million, keys. This is doable in less than a second as outlined above.
4. Encryption turns a 128bit string into another 128bit string. Even if the original string only consisted of visible characters, encoded in UTF-8 (8 bits to encode 1 character), the ciphertext when interpreted as a UTF-8 string may now contain information that is not printable which leads to errors in transmission. Base64 allows us to encode bit strings, 6 bits at a time, into printable characters. This way we can e.g. print them in an exercise sheet without any information being lost, as the original bitstring can be recovered without problems.
5. Here it is meant that "crypto" is the passphrase which openssl automatically turns into a key. By copying the ciphertext into a text file "encryption.txt" we can use the command "openssl aes-128-ecb -d -in encryption.txt -k crypto -a -out decryption.txt" to decrypt the ciphertext.

Exercise 3:

1. An attacker can simply precompute hashes from all phone numbers. In the Danish example, there are only 100.000.000 phone numbers, which amounts to 100 million hashes. Given e.g. exercise 2, this should not take a long time to compute. If one hash is 256 bits long, we can also estimate that storing a lookup-table would only require around $32 \cdot 100.000.000 \approx 2^{32}$ bytes, which is equal to 2^{22} Kilobytes or 2^{12} Megabytes, or around 4 Gigabytes. The provider could easily store this table and for every sent hash use it do look up the original phone number, which is the preimage of the hash.

[1] See e.g. `https://bench.cr.yp.to/results-stream.html`

Exercise 4:

1. $N = 221$ and $d = 77$. For $m = 4$ this gives $c = 140$
2. For $0, 1$ the message and the ciphertext are the same (so an attacker can always see this). More problematic, every message mod N translates into a unique ciphertext, so without additional tricks the attacker who knows what the plaintext could be can always "test" by encrypting the plaintext and comparing
3. Instead of starting from a message mod N, just only allow half of the bits of a plaintext will carry the actual message and we use random bits in the other half to "pad".

Exercise 5:

1. Here so-called Replay attacks can happen, because there are just two messages that can be sent. Once someone observed them, he can always replay them on the network.
2. Assume that sender and receiver keep state of a counter c. In addition to the message, the sender now signs both the message and c and increases c afterwards. The receiver checks if the signature he receives is valid and if the counter c that is part of the signed message agrees with his local counter. If so, then after verifying the message he increases his local counter as well.