CYBERSECURITY

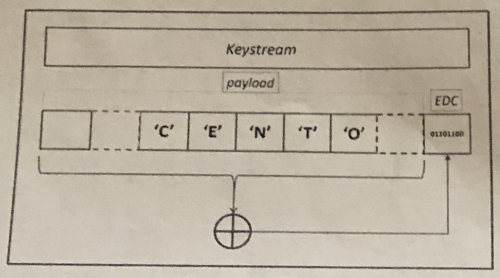
Master in Computer Engineering

23 July 2019

EXERCISE NO. 1 [CE (ALL) + ECS] #marks: 10

1. Prove that in a perfect cipher the number of keys cannot be smaller than the number of messages (*Shannon's theorem*).
2. Consider the mono-alphabetic substitution cipher.
   a. Prove that it is not a perfect cipher.
   b. Determine at least one cipher-text that would never decipher into plain-text "CS".

EXERCISE NO. 2 [CE (ALL) + ECS] #marks: 12



Let us consider the OTP-based secure communication scheme reported in the figure. According to that scheme the cleartext is composed of two fields: a) a payload, and b) an error detection code (EDC). The error detection code is obtained by bitwise *xoring* the payload byte-by-byte.

1) Argue whether the scheme is malleable.
2) Propose a malleability attack that, if successful, changes "CENTO" into "MILLE". Assume the adversary knows the position of string "CENTO" in the plaintext (and therefore in the ciphertext). The EDC is always at the end of the ciphertext.
3) Propose a modification to the communication scheme that prevents malleability attacks.

EXERCISE NO. 3 [CE (OLD PROGRAM) + ECS] #marks: 8

1) What are certificates for?
   a) Establishing an indissoluble link between an identifier and a public key.
   b) Establishing the privileges of the owner of the certificate.
   c) Establishing the trustworthiness of the certificate owner.
2) Describe the minimum set of data fields that you expect to find in a certificate.

EXERCISE NO. 4 [CE] #marks: 8

Considering the argument "A" as tainted, find and describe the vulnerabilities in the following function. Argue how an attacker can exploit such vulnerabilities, and what she can obtain. Suppose that all compiler optimizations are off.

```
std::vector<int> vector_int(100);
int func(int A) {
   // fill with progressive numbers
   // and insert A every 10 elements:
   int count = 0;
   for(auto i = vector_int.begin(); i != vector_int.end(); i++) {
      *i = count;
      if(count % 10 == 0) vector_int.insert(i, A);
      count++;
   }
   // return element at position A:
   return vector_int.at(A);
}
```