University of Padua
Department of Information Engineering

Information Security Report
Laboratory Session 3

Naïve entity authentication scheme

Author:
Bellin Arturo
Tascioglu Ayca Begum
Lazzaro Davide Maria
Varotto Francesco

Teacher:
Nicola Laurenti

21st December 2020

Task 1

In the script task1.py it has been implemented the naïve authentication protocol between two entities: A and B.

Complexity

The protocol has been asked to be polynomial time in l_c and l_k, and so it is its implementation, in fact (note: I only consider the complexity of generating numbers and the computations using them, not the one for sending the information):

1. The setup phase is realized through the setup() method; its computational complexity only depends on the uniform and random generation of the key, made by the method *generateUniformNumber()*, which is O(l_k);.
2. The phase 1 is O(1);
3. During phase 2, B is supposed to generate a random, binary and uniform challenge c with l_c bits in length, so it is O(l_c);
4. In phase 3 A must: convert c to its decimal representation (operation which is O(l_c)), sum its digits (O(l_c), but we can have a (very) tighter bound since the sum of the digits of c outputs a number that can be a lot smaller than c, we will see it next), compute the sum $t = k + n$ ($O(max(l_n, l_k))$, but, at least at the beginning, $l_k >> l_n$, so it becomes O(l_k)), convert t to its decimal representation and sum its digits (both operations, at least at the beginning, are O(l_k)), then compute the product s = $s_c * s_t$ ($O((max(l_c, l_k + l_n))^2)$). Finally, she computes the binary representation of s ($O(l_s) \in O(l_k)$);
5. Phase 4 has the same complexity of the previous one.

The maximal computational complexity comes from step number 3, precisely from the multiplication between s_c and s_t, which is $O((max(l_c, l_k + l_n))^2)$.
Now, let us assume that $n = 0$ (for simplicity) and $l_k = 4$: the biggest sum we can get summing the digits of t is 36 (when t is equal to 9999) so the length of s_t is less than the square root of l_k. If $n = 0$ (for simplicity) and $l_k = 5$, then s_t can be at most 45, and its length is less than the square root of l_k. So: when will s_t have three digits? Supposing that $n = 0$, then k has to be equal to 999999999999, so $s_t << \sqrt{l_k}$.
The same way of thinking can be applied to c and s_c.
From this intuitive proof, we can tighten the bound from $O((max(l_c, l_k + l_n))^2)$ to $O((max(\sqrt{l_c}, \sqrt{l_k + l_n}))^2)$.
Supposing that $l_k > l_c$ (very likely) and $l_k > l_n$ (e.g. at the beginning), the bound becomes $O((\sqrt{l_k})^2) = O(l_k)$, so it is linear to the length of k (as we can observe looking at figure 1).

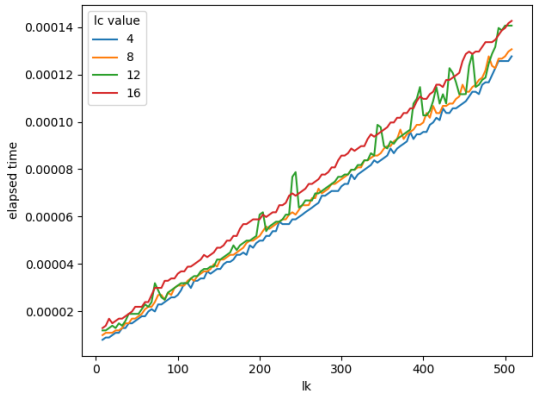
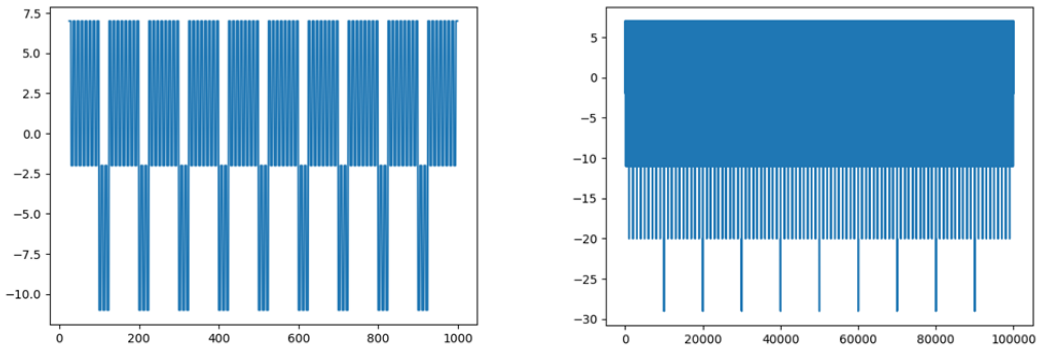


Figure 1: Computational complexity of the legitimate protocol

Task 2

The eavesdropper C wants to break the protocol and at round $n' = n$–25 he has been able to observe the legitimate round, so he got acquainted of: the id of A, the challenge c, the round n and the response r that A computed. Elaborating this knowledge, he can get: s, which is simply r in base 10, s_c, which he is able to obtain through c and s_t ($= s/s_c$). Since he does not know the key k, C cannot achieve t, but he can get some information about the distribution of the s_t's in a general round n and the ones in the round n'.



(a) with 1000 rounds

(b) with 1000000 rounds

Figure 2: periodicity of the difference between s_t and s'_t

Supposing that the number of rounds of the authentication protocol between A and B is equal to 1000, and that k is equal to 0, C can observe the behavior

represented in figure 2*a* (note that if k is different from 0, then the figures would only be translated, so nothing would change).

Let us define s_t as the sum of the digits of t in round n and s'_t the sum of the digits of t during round n'. C can observe that their difference is distributed among these numbers: 7, -2 and -11.
Their frequencies instead are the following (figure 3a):

```
frequencies of St diff:
[[-11 135]
 [ -2 440]
 [  7 400]]
```

(a) with 1000 rounds

```
frequencies of St diff:
[[  -29   135]
 [  -20  1440]
 [  -11 14400]
 [   -2 44000]
 [    7 40000]]
```

(b) with 1000000 rounds

Figure 3: Frequency of the difference between s_t and s'_t

C can note that $s_t - s'_t = -2$, 45% of the times.

Instead, using 1000000 rounds the system has the behavior represented in figure 2*b*: now, C can observe that their difference is distributed among the numbers: 7, -2, -11, -20 and -29 but -2 is always the most frequent (figure 3b), and it appears 44% of the times (very likely as before).

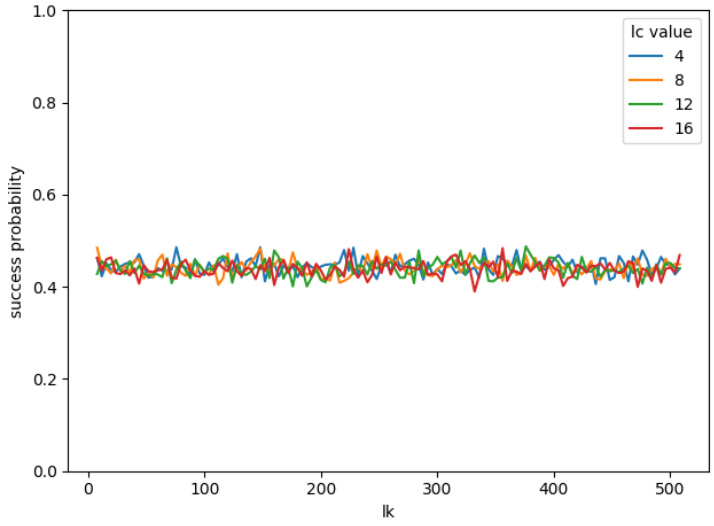


Figure 4: Success probability of the attack of task 2

So, C can exploit this fact in this way: when he computes s_t, in order to have a collision, he can simply subtract 2 to the result he obtained and

operate with the resulting s_t. Working in this way, he gets accepted more or less with probability 0.44, as can be seen in figure 4:

As regards the computational complexity of the attack, we can see its behavior vs lk, using different values of lc in figure 5: as seen before, the computational complexity remains always linear because the forger does not make any other operation than the ones related to the authentication protocol (he only subtracts -2 from the computed s_t value, which is irrelevant) so it remains linear, as proved before.

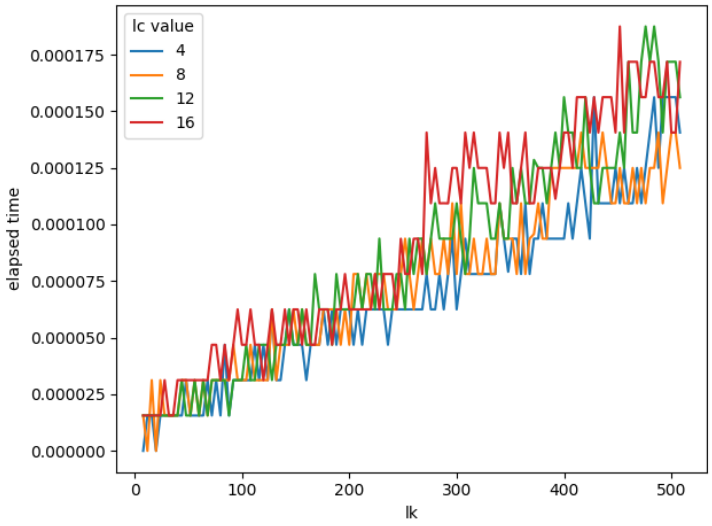


Figure 5: Computational complexity of the attack of task 2

Task 3

In this case C is not able to observe any previous round of the legitimate protocol, and he needs to rely only on the information available in the current round, in particular: n,c and r. As explained before, it is possible to find s_c, so C is just missing s_t in order to compute s and subsequently r and break the protocol. s_t is the sum of the digits in base 10 of $t = k + n$, where n is known and k is the secret key shared between A and B. There is no possibility of finding k but maybe C does not need to!

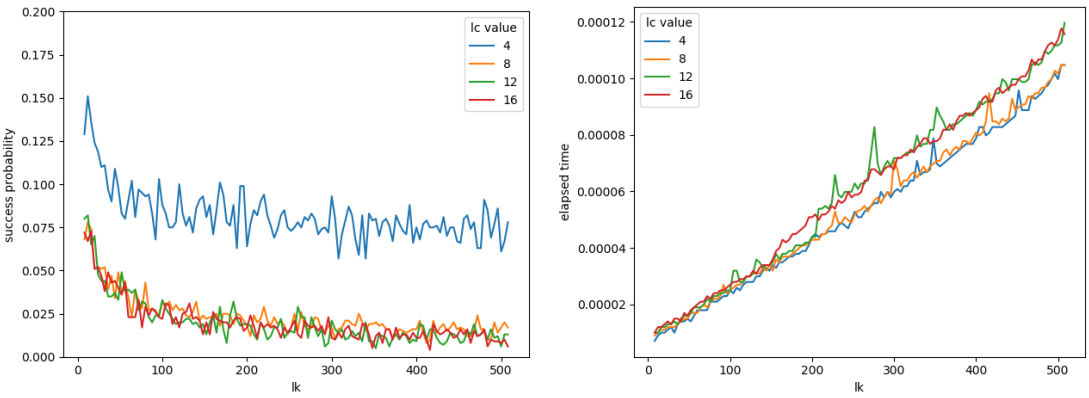
Let us focus on the function $\mathcal{F}(x) = y$ where y is the sum of the digits of x in base 10. It is easy to note that $\mathcal{F}$ is not a one-to-one function so inverting it is not a viable option. One possible approach to this problem is to notice that even if x is randomly and uniformly distributed, we cannot say the same for y! In fact the distribution of y is similar to a Gaussian where there are values that

are much more frequent than others.

Our attack is based on this property: we do not try to find t but we search for the most probable value of s_t. We compute an approximation of it by considering the expected value of each digit of t under the assumption that they are uniformly distributed in the interval $0 - 9$. This is obviously not always true, especially for the most significant digits, but it is a very fast and easy approximation. To find the number of digits of t in base 10 we take the maximum between the number of digits of n and the number of digits of k (let's call this value l_t^{10}). Recall that for Kerckhoff's assumption the attacker knows l_k and can compute the number of digits of k in decimal using the formula:

$$\log_{10}(2^{l_k}) = l_k / log_2 10 \tag{1}$$

So, putting it all together, the value that C guesses for S_t is $E(S_t) = 9/2 * l_t^{10}$. The success probability and the computational complexity can be seen in Figure 6. The computational complexity remain linear for the same reason explained in task 2, since the attacker performs only a O(1) operation.



(a) Success probability of the attack

(b) Computational complexity of the attack of task 3

Figure 6: Attack of task 3, for several different values of l_c