Nomenclature

- **authdata** = raw message received by the server

- **m** = concatenated message given to the hash function
- chunk = a 512bits long number processed by the main sha1 operation
- chunk’s result = an aggregate of five 32bits numbers that are the result of sha1’s main process on the chunk

Process

1. Look at the SHA-1 algorithm
   - I tried to see if there is an obvious way to predict the outcome of the leading zeros without completing every step of the algorithm, but since a cryptographic hash function is supposed to have the avalanche effect : each output bit should depend on every input bits

2. Attempt to contact the server with openssl command line. No problem.
   - The protocol is just like described in the example

3. Implementation of SHA-1 in Rust
   - so I can grasp better how it works
   - so I can have control on the computed chunks

4. Understanding of SHA-1 for the problematic
   - Each 512bits chunk generate 5 numbers that are added to the 5 numbers of the previous chunk
   - We can reuse the result of the previous chunks of *m* and extend/replace the next chunks, so we don’t need to recompute everything. We can reuse the results of only parts of *m* that fill completly a chunk, so we can reuse the full result of *m* only when its bit *s*ize is a multiple of 512.
   - For each chunk of *m* we have 2^{512} numbers to choose from, so we don’t need to extend the number of chunks much, 1 additional chunk that we can modify freely is enough, unless we find a chunk’s result that is somehow interesting
   - We can try to select which chunk’s results are more probable to give the end result we are looking for
   - I assume that we can’t know what will be the result of these 5 numbers given the input unless we pass it to the sha1 main process (loop of 80 iterations), so the bits of the output numbers are assumed to be equiprobable (each bit’s value has a probability 1/2)
   - Therefore, if the input can’t help determining the output at all, as it appears random, we might as well replace the random generator of the input by a simple iteration over the range [0...2^{512}-1], it’ll be faster and we have no risk to pull the same number again
     - I realized I can’t do that, because it could lead to uncorrect utf-8 characters, and this is one of the constraints for communicating with the server
     - so we can just iterate over every utf-8 characters, they are much less numberous, but still numberous enough
5. Study of the probabilities

   1. Which couples of numbers sum to n consecutive leading 0 ? (in the following, **|** designates the concatenation operator, and **C(x)** the 1-complement of x)
      - any couple of numbers (x | a, (C(x) + 1) | b) such as x is of size n and a + b doesn’t overflow

        - any couple of numbers (x | a, C(x) | b) such as x is of size n and a + b overflows
    - P(finding C(x) | x) = P(finding C(x)+1 | x), so we care only about the probability of overflowing
2. What are the odds that the n_th most significant bit produces a carry? Let’s call **P(C_n = 1)** the probability that the n_th most significant bit produces a carry
    1. P(C_0 = 1) = 1/4
    2. P(C_(n+1) = 1) = P(C_(n+1) = 1 | C_n = 0) * P(C_n = 0) / P(C_n = 0 | C_(n+1) = 1)
        1. P(C_(n+1) = 1 | C_n = 0) = 1/4
        2. P(C_n = 0 | C_(n+1) = 1) = 1/2
        3. So we have P(C_(n+1) = 1) = 1/2 * (1-P(C_n=1))
        4. This can be represented by a sequence S_0 = 1/4 , S_(n+1) = 1/2 * (1 – S_n)
        5. This solves to be S_n = 1/3 + 1/(-12*(-2)^n), which converges to 1/3 at infinite
    3. In conclusion, we have **P(C_n = 1) = 1/3 + 1/(-12*(-2)^n)**, so we have around twice more chance not to have a carry at the n_th most siginificant bit when n is big enough
3. If we know that the most significant bit is 1 on one of the two numbers, then this means half of every possible values for the second number will trigger a carry when summing, so I have the intuition that the chances to have a carry should increase drastically with every consecutive leading 1. So we could try to figure out what is the probability of having a carry knowing one of the numbers we are summing.
    1. Let x be a given number of bit size n, and y a number being picked randomly in the range [0, 2^n-1]
    2. The smallest number s such that x + s overflows is C(x) + 1 and we have x + C(x) + 1 = 2^n
    3. Therefore, every number over C(x)+1 makes the sum with x overflow, and every number under it doesn’t. So the number of numbers that make the sum with x overflow is #[C(x) + 1, 2^n-1] = 2^n-1 – (C(x) + 1) +1, and knowing that C(x) = 2^n - x -1, we have #[C(x) + 1, 2^n-1] = x
    4. So the probability of overflowing x + y is **x/2^n**
4. As previously said, most significant bits are the ones that matter the most. In fact **P(overflow | k most significant bits are 1) = 1-1/2^k**. As an example, P(overflow | 5 most significant bits are 1) = 0.96875, so there is more than 96 % of chance to overflow.
5. So the probability not to find a final solution with a suffix number that have k most significant bits equal to 1 (or 0 depending on the case we are looking for), after m tries is **1 – (1-(2^k-1)/2^(difficulty+k)) ^ m**

The algorithm

1. Make sure that *authdata* is a multiple of 512, extend it with arbitrary values otherwise.

2. Compute the intermediate values using the possibly extended *authdata.*

    - *Those two steps are done to prevent recomputing this one chunk for each thread/work-item*

3. Generate a new chunk for each work-items with a different UTF-8 string each.

    - We do this for each work-item to have different starting points, so they don’t compute the same thing

4. Compute a new chunk until we find one with P(overflow) and one with 1-P(overflow) at maximum values, considering only the first k bits

    1. What we are looking for in general is :

    - any number $v = v_0 | v_1$ such as v_0 is of size difficulty and with P(overflow | v_1) is as low as possible
    - any number $v = v_0 | v_1$ such as v_0 is of size difficulty and with P(overflow | v_1) is as high as possible

5. So we have 2 chunks, one that maximize the probability to overflow, and another that lowers it. Now we try different strings in the final chunk (with the padding and the message size) until we find the number of zeros we are looking for, starting from the two previous result's chunks.