The Final Poor Puzzle

Well done if you have made it this far! I have bad news, though… In real cryptography, it is almost never the case that what makes a message insecure is the way it was encrypted. Normally, the attacker knows exactly how a message was encrypted and also knows that it would be impossible to decrypt the message without the key. What they attack is human error elsewhere. This is certainly true in this week's puzzle. I have opted for an encryption known to be mathematically perfect, meaning that you can't decrypt it without the key, which is as long as the message and is randomly generated. So don't just guess it! You can get the key by heading to www.hobson.space/poor-print/three where you have to enter a password (only numerical digits of length less than 100) to retrieve it; with this information, you know that there are 1.11×10^{99} possible passwords. If you could check 50 password a second, it would take 7×10^{89} years to check all possible passwords, bearing in mind that the universe has only been around for 1.4×10^{10} years; so really don't just guess it… But as promised, I will tell you exactly how the message was encrypted – as well as every other stage in this encryption system – with a guarantee that there is a flaw!

How the Encryption Works

We first define our alphabet,

$$\Sigma = [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, .\ ,\ , /]$$

and define Σ_i to be the i^{th} element of our alphabet (this is 1-based, not 0-based). For each letter we want to encode, we take the index, i, and generate a random key, k. We then use these in the function:

$$E : \langle i, k \rangle \mapsto i + k \ mod\ 30$$

In case you are unfamiliar, $a \ mod \ b$ is the remainder when we calculate $\frac{a}{b}$, so, for example, $5 \ mod \ 6 = 5, 6 \ mod \ 6 = 0, 7 \ mod \ 6 = 1$, etc. This means that to find the letter, we take the cipher number, say 5, we take the key, say 7, and we do $E_{m-k \ mod \ 30}$ so in our example, $\Sigma_{5-7 \ mod \ 30} = '.'$

There is a different key for each letter in the cipher. What you just read in this section was a hideously complicated way of saying that we shift each letter the amount specified by the key, so A with key 2 = C… In my defence, I said I would explain *exactly* how it worked…

Getting the Key

In order to get the key, you can head to http://www.hobson.space/poor-print/three, where you can enter a password in return for the key. There is only one password which consists of numbers and is between 1 and 100 digits long. I will now explain how the password checker works. The potential password is stored in a list, for example, the password '13728' would be in a list that looks like [1,3,7,2,8]. This is then checked against the real password. Now for some more detail!

The old way of computing lists would be to allocate n blocks of memory next to each other and store the address of the first block; so, say we stored the word "hello", we would allocate 5 blocks of memory and store the address of the first in a variable called $text$. So the value at address $text$ is 'h', the value at address $text + 1$ is 'e' and the value at address $text + 4$ is 'o'.

Sadly, there is a problem with this way of storing lists. What if we don't know how big the string of text will be? We can't know how much memory to allocate, and reallocation is a mess. To solve this, we use *linked lists:* each element of the lists is stored as a pair $(element, next)$, where next is the address of the next pair. This way, we do not need to pre-allocate memory because the pairs can be distributed through memory. At each point in the linked list, we say that the pair we

are considering is the *head* of the list, and the rest of the lists (accessed by following where $next$ points) is the tail. So in our example ‘hello’, ‘h’ is the head and ‘ello’ is the tail. When we follow the link to ‘e’, we can no longer access ‘h’, so we call ‘e’ the head, and ‘llo’ the tail. The end of the lists is indicated by setting $next = 0$

In the password checker online, I chose to use Haskell because it is a very secure language. For reasons too complicated to explain in this brief outline, its default implementation of lists is *linked lists*. The password checker is implemented somewhat like this:

$$checker\langle (p_h, p_t), (x_h, x_t)\rangle \mapsto \begin{cases} False & p_h \neq x_h \\ False & p_t = 0 \neq x_t \vee x_t = 0 \neq p_t \\ True & p_t = x_t = 0 \wedge p_h = x_h \\ checker\langle p_t, x_t\rangle & otherwise \end{cases}$$

where p is the password and x is the text that the user (you) thinks is the password. The subscripts h and t represent the head and tail of the lists. The code is a little more complicated than this, but this is the general idea.

The Cipher Text

18,1,26,4,20,5,19,20,21,0,10,27,19,18,5,23,20,25,13,6,11,3,5,29,22,20,13,18,17,29,28,21,9,27,26,10,8,9,7,5,26,23,16,26,17,12,16,2,28,8,2,20,15,13,12,3,3,14,4,8,25,17,22,19,24,11,6,20,21,10,3,11,27,7,2,5,26,14,12,3,23,24,10,5,9,19,14,26,14,2,6,12,6,2,6,17,21,11,3,6,21,2,21,8,8,15,13,17,16,6,4,29,11,15,1,21,16,0,8,22,11,19,19,9,4,26,13,20,0,10,14,4,3,22,3,21,23,14,9,25,5,26,4,27,0,10,8,26,22,7,24,2,9,20,28,27,28,25,20,22,9,25,13,28,10,26,11

In this is hidden the web address of part two (type it into your browser in **lower case**). Good Luck!