UIUCTF 2018: Xoracle (250)

ecx86

In this problem, we deal with multiple encryptions of the same plaintext, namely many random-length, random-valued one-time-pads. The problem could also be framed as a many-time pad attack where the attacker wants to deduce the pad. One key observation is that

$$c_i \oplus c_{i+k} = (p_i \oplus o_i) \oplus (p_{i+k} \oplus o_{i+k}) = (p_i \oplus o_i) \oplus (p_{i+k} \oplus o_i) = p_i \oplus p_{i+k},$$

where p is the plaintext, o_i the corresponding one-time-pad, and k is the pad's length $|o|$.

Instead, we will take a statistical approach: WLOG, fix a keylength k. Then, collect many ciphertexts $c_0 \cdots c_i$. Next, for a given i, we could, for each ciphertext, calculate: $c_i \oplus c_{i+k}$. Naturally, for most ciphertexts, our fixed k is incorrect, so $(o_i \oplus o_i + k)$ is some random byte. It is crucial to note, however, that $(o_i \oplus o_i + k)$ is randomly distributed from 0 to 256, and likewise, $c_i \oplus c_{i+k} = (p_i \oplus p_{i+k}) \oplus (o_i \oplus o_i + k)$ would be random too. However, if our fixed keylength was by chance correct, then $c_i \oplus c_{i+k} = (p_i \oplus p_{i+k}) \oplus 0$. In other words, for a large corpus of ciphertexts c_i and a keylength k, we could measure The frequencies of $c_i \oplus c_{i+k}$ from 0 to 256, and the correct value $p_i \oplus p_{i+k}$ would show up as a spike. This allows us to calculate $p_i \oplus p_{i+k}$ for any i, which we denote as $(i, i+k)$, where $(i, j) = p_i \oplus p_j$. However, a major limitation is that $128 \geq k \geq 255$ because of the way the one-time-pads are generated.

Now, it's useful to note that since $(i, j)(j, k) = (i, k)$. For brevity, the $\oplus$ operator has been elided. Using this property, we can extend our oracle from $(i, i+k)$ to (i, j) for any $i, j \in [0, l]$. To see this, consider any two i, j. WLOG, say $i < j$. There are three cases: First, if $k = j - i$ is acceptable; in which case, we are done. Next, if $j - i > 255$, i.e. k is too high; we can rewrite (i, j) as $(i, i+255)(i+255, j)$. Note that the left symbol has an acceptable gap $k' = 255$, whereas the right symbol has a gap $k' = j - i - 255 = k - 255$. In other words, the gap has strictly decreased from k to k'. Then, we can continue recursively rewriting the right symbol until it is acceptable, at which point we are done. Finally, if $j - i < 128$, i.e. k is too low; we can rewrite (i, j) as $(i, i+255)(j, i+255)$. Like earlier, the left symbol's gap is acceptable, while the right symbol's gap $k' = i + 255 - j = 255 - k$; however, since we know $k < 128$, that means $128 \geq k' \geq 255$, and we are done.

Now, by chaining multiple of these xor pairs (i, j), we can achieve $(i_1, i_2, i_3, \cdots i_n)$ for any even n. For example, $(1, 2)(1, 3) = (2, 3)$ and $(1, 2)(3, 4) = (1, 2, 3, 4)$. In other words, we could calculate the combined xor sum of any even number of bytes from the plaintext:

$$(i_1, i_2, i_3, \cdots i_{2n}) = \bigoplus_{m=1}^{2n} p_{i_m}$$

Another way we can think about these combined xor sums is a bitstring representation. For a bitstring of length l, we say the ith digit is 1 if p_i is included in the xor sum. So really, we are working in the group $G =$ (l-bit numbers of even Hamming weight with the operator $\oplus$). It is easy to verify this: G is closed and associative under $\oplus$, the identity is 0, and each element is its own inverse. Likewise, the group is Abelian (e.g. commutative).

Unfortunately, because our group G only includes even Hamming weight xor-sum inclusion bitstrings, it's impossible to calculate the value of any plaintext byte by itself. So instead, we calculate $(0, i)$ for each i from 0 to l. In effect, we have effectively reduced the random keylength one-time-pad to a single-byte pad $p \oplus p_0$. Now, we simply have to try all 256 possibilities of the key p_0, calculating $p_0 \oplus (p \oplus p_0)$. For the correct p_0, this would result in the original plaintext p. Another way to think about this is that by brute-forcing any one of the bytes p_0, then we could say:

$$G \cup p_0 = (\{0, 1\}^l, \oplus, 0),$$

meaning we could access the xor-sum of any combination of plaintext bytes, including the singleton sums p_i which together form the plaintext p.

Finally, we use the linux utility 'file' to find which of the decryptions has a known format.