Individual Project Assignment for CE 4024

Introduction

The assignment provided to the students involved a proxy communicating with a server. The client allowed the users to try to log in and send messages and the server responded according to the client message. We were only allowed to use the encrypted client messages and server responses in .dat format to solve two problems. In the first one, we find the users that successfully logged in. In the second one, we capture every attempt made by the users to log into the server and display if they were successful or not.

The Vulnerability

In my initial attempt to decrypt the messages, I tried to break the RC4 encryption. When I was studying this protocol, I realized it involves a random number generator (even if it's pseudo), so breaking it would require brute force and a lot of resources. So, I decided to violate the protocol itself.

The biggest vulnerability in this system is that the same secret key is used more than once for encryption - first by the client to encrypt its messages, and then by the server while encrypting its responses. Given that both the message and the response are of equal lengths, the same ciphers are used for any given pair of client message - server response conversation.

The XOR type - key used in RC4 is perfectly secure, with one major problem. The key cannot be used more than once, and that is exactly what this setup is doing, allowing me to implement an attack by "peeling away" the messages one by one. I will discuss this attack in detail in the next section

The Attack

Once I figured out that a one time pad is used twice in every conversation between the client and the server, I decided to exploit it by using the basic logic relation in XORs. Let's abbreviate the actual and encrypted client messages as M and Em respectively, and the actual and encrypted server responses as R and Er respectively. Let's abbreviate the stream cipher used for both these encryption (during the same conversation) be abbreviated as K.

Now, we know that the protocol is defined as follows:

$M \oplus K = Em$

$R \oplus K = Ek$

We can verify that, the following xor logic rules holds:

$K \oplus K = 0$ *and* $K \oplus 0 = K$.

We can use this to come up with:

$M \oplus K \oplus R \oplus K = Em \oplus Ek$ *or*

$M \oplus R = Em \oplus Ek$.

So by grabbing the right pair of encrypted message and encrypted response, we can xor the two to get the xor-ed value of the actual messages and responses. Once we end with the "jumbled" form of the actual conversation, we can use violate the following XOR rule:

$K \oplus M \oplus K = M$

As we know the possible formats of the client messages and server responses, we just need to XOR the right known strings to get eliminate one of the messages - leaving us with just the other (at least in part). I will briefly explain how i specifically did it for every problem, but the logic has the same basic approach.

Problem 1

While using python, I decided to use hex has the medium for XORing. So all the known strings and byte-messages were converted into hex and back into string or bytes as needed for the computations. It is known that the message size is 128 bytes, so we grab an encrypted message and response as a pair (of 128 bytes each) to further manipulate them.

First, I XOR the encrypted message and response to get the XORed actual message and response (let's call this val). I use the known words (*WELCOME* and *LOGIN*), to "peel" the message and the response one by one. When val is XORed with *WELCOME*, we are left with a message of the form "*LOGIN us...*" (Assuming username is "username", … is some jumbled text). Similarly, when val is XORed with *LOGIN,* we are left with a message of the form "*WELCOM..*" (… is some jumbled text)

Checking for those starting words, we can grab the right conversations from the message logs needed to find out which users logged in. Now, the trick is the word *WELCOME* is 2 words longer than the word *LOGIN*. Therefore, we can use "*E* " as a known word, to grab the first two letters of the username. (*M* gets clubbed with the space after *LOGIN*.) Once we know the first two letters of the username, we can use the same logic and grab the next two letters, because both *LOGIN* and *PASSWORD* are followed by the same letters!

I stop once we reach a space at the end - as that indicates that the username has ended, and we have started to enter the password section of the client message.

We do this for all the messages in the logs, pushing the successfully logged in users in a list which is finally output to a file in the format required by this assignment.

Problem 2

Problem 2 is very closely related to problem 1, and builds easily on top of it. It has 3 parts, one where the user successfully enters the username and the password, one where the username is correct but there is a password mismatch and one where the username doesn't exist in the database.

The latter two conditions have pretty much the same implementation, with just the known words differing (*PASSWORD MISMATCH* vs. *INCORRECT USERNAME*.) For both these cases, the trick is to add enough spaces after the known words to balance the extra letters in the client message (where he/she tries to log in) to grab the password as well. I assumed it was all 0s in hex once we reach the end of the word, but I found it that spaces are used to pad to the right end of the message to make it 128 bytes.

Once I XOR the encrypted message and response, and then xor it again with the known word with the right number of spaces, I end up with the wrong attempt the user has made to log in and I format it according to the requirement of this assignment.

The first condition, where the user successfully logs in, just builts right on top of problem 1. I use the same idea of padding spaces at the end of the known word, once we reach the end, to grab the passwords. There is a slight difference in the way I handle this when we reach the end of the username with the first or second letter as space, but that's just a simple computational fix as I increment my counter to grab two words in a row.

Finally, I output the string formatted to meet the requirements to a file named Problem2.txt.

Takeaways and Conclusion

The biggest lesson learn from this assignment is when a one-time pad cipher is used, the keys are intended to be used just once in the entire interaction. These ciphers are vulnerable to an attack if the same key is used twice, allowing hackers to get an xored version of two actual messages transmitted through this "secure" channel. Another important thing I learnt is that XORing actual messages with their keys might not be the best way of encrypting important messages due to the simplicity in recovering jumbled messages owing to the logic rules of XOR.

During my research on RC4 encryption (when I was actually trying to break it), I realized that it in vulnerable itself to attacks like "bar mitzvah attack", even though it provides a partial part of the actual message. Therefore, a major takeaway would be to use more secure encryption in my applications.

Finally, I learnt that Python makes your life harder when you want to XOR stuff, and converting among hex, bytes and strings and ascii is not as simple as one would imagine! Also, using a whiteboard to tackle cryptography/big programming problems really helps, is one of my biggest takeaways from this project.