Liam Brew
1 October 2020

Professor Triandopoulos
CS 306

Homework 01

Question 1

1.1: Two basic security properties that should be considered are integrity and confidentiality. Integrity is important as since communication is taking place over an insecure channel, it must be protected against possible manipulation by nefarious actors through methods such as MitM attacks. Confidentiality is important as the communicated content, especially the key, must remain hidden from potential attackers. This would reduce the likelihood of attacks occurring in the first place.

1.2: This protocol does not satisfy the above two properties. This is because only Alice's portion of the protocol is probabilistic due to its incorporation of a random aspect. Bob's is not, meaning that his deterministic portion can more easily be replicated. This ease of replication violates confidentiality, and the potential for an attacker to replicate the process and use the results of it to interfere with the Alice-Bob communication through a nefariously generated message violates integrity.

Question 2

2.1: With a key length of 1, the attacker can easily determine the password by looking at the initial few characters and examining their difference, thereby revealing information about the offsets required to decipher the message. Similarly, with a key length of 2 this encryption algorithm will not be valid here as well. This is due to the nature of the two passwords at play, *abcd* and *bedg*. Because of the nature of the passwords in that a-c, b-d, and e-g are all the same distance apart, the attacker can easily tell which password to use. For a key length of 3, there are still only two possible options for the attacker to pick from. These are a,b,c,(a+3) or b,e,d,(b+5). This greatly enhances the attacker's likelihood of infiltrating the system. Finally, with a key length of 4, the system is more secure. This is because as the length of both passwords are 4, the attacker will be unable to determine any new information.

2.2: The minimum amount of chosen plaintext required to recover the secret key is 25 letter characters. This is because knowing that the English language is comprised of 26 letter characters, the remaining letter can be inferred given the presence of 25 others. I found the shortest possible plaintext that can recover this key is the sentence "waltz bad nymph for quick jigs vex". This cipher is perfectly secure against a cyphertext-only attacker under the following condition: when the probability of mapping letters and of mapping ciphertext are equally likely to occur.

Question 3

I believe that I was able to solve the cipher. I did this through a combination of crib dragging, reasoning and brute forcing. I wrote a Python program to assist in the required mathematical computation. The first method of this program, *xor*, performed the $\oplus$ operation on two input strings, returning the result. This code is shown below and made use of Python documentation[1]. To test if this method was working, I made use of an online XOR calculator[2].

```python
def xor(str1, str2):
    xor_str = ""
    for char1, char2 in zip(str1, str2):
        xor_char = int(char1, 16) ^ int(char2, 16)
        xor_str += hex(xor_char)[2:]   # strip leading hex digits
    return xor_str
```

I pledge my honor that I have abided by the Stevens Honor System

Once I confirmed that my XOR method was operational, I wrote the *crib_drag* method. *crib_drag* takes two inputs, a ciphertext and a hex-format guess word and XORs them together. If the size of the guess word is smaller than that of the ciphertext, then the ciphertext is broken up into appropriately sized blocks and each of these is XORed with the guess word. When creating my *crib_drag* method I referenced online documentation regarding crib dragging[3,4]. My *crib_drag* method is found below.

```
def drag_crib(cipher, word):
    count = 0
    for i in range(0, len(cipher), len(word)):  # split cipher into word-
sized blocks and analyze
        new_cipher = cipher[i:i + len(word)]
        hex_xor = xor(new_cipher, word)
        bytes_xor = bytes.fromhex(hex_xor)
        xor_string = bytes_xor.decode("ASCII")
        fw.write('%d:%s\n' % (count, xor_string))
        print(f'{count}:{xor_string}')
        count += 1
```

I based the next step of my program off security research I've previously performed that consisted of hacking WEP routers (brute forcing common passwords). Using a list of the most frequently used words of the English language[5], I was able to decrypt enough of the ciphertext to establish a sufficient baseline for reasoning and estimation of the rest. I achieved this by crib dragging each of these words through the ciphertext and saving the results in the original *_results.txt* file. This work is encompassed in my *top_words* function, seen below:

```
def top_words():
    fr = open("top_words.txt", 'r')
    top_words = fr.readlines()
    fr.close()

    cipher_text = xor(cipher1, cipher2)

    for word in top_words:
        fw.write("\n################################\n%s\n" % word)

        word = binascii.hexlify(word.encode('utf-8'))
        word = word.decode('utf-8')

        drag_crib(cipher_text, word)
```

As you may expect, the initial results of this crib drag were numerous, with the vast majority being invalid. I manually went over the results file and removed any word that I determined to be infeasible (illegal characters, nonsensical format etc.), resulting in a great reduction of possibilities that are found in *pruned_results.txt*. These possibilities were words that contained crib blocks of potentially valid strings, such as English characters or parts of words. I initially focused on blocks that contained partial English words, such as those found within the words *to, from, room, vowel, voice,* and *front*. I reverted

to manual control of my crib drag function here, entering variations of these words and their crib blocks in the hopes of finding more information about the ciphertext. None of these proved fruitful, but my next word, *test*, did. The results of my manual manipulation on *test*-related variations is found in *test_results.txt*. As I decrypted more and more of the cipher text, the remainder became easier to determine. Eventually, I was able to completely decode the first message, resulting in the following: "Testing testing can you read this".

From here, I XORed this message with its ciphertext and determined the key, which is the following: "youfoundthekey!congratulations!!!". This allowed me to determine the rest of the messages by XORing the ciphertexts with the key. The full conversation is:

```
Testing testing can you read this
Yep I can read you perfectly fine
Awesome one time pad is working
Yay we can make fun of Nikos now
I hope no student can read this
That would be quite embarrassing
Luckily OTP is perfectly secure
Didnt Nikos say there was a catch
Maybe but I didnt pay attention
We should really listen to Nikos
Nah we are doing fine without him
```

While I believe that this is the correct answer, I also believe that my algorithm could be made more efficient. While I tried to leverage my previous cybersecurity knowledge in writing this program, I understand that the solution was still mostly obtained primarily through luck. For example, if the word *test* wasn't in my sample size, then it is doubtful that I would have been able to calculate the answer efficiently. Additionally, my method required a lot of manual work. I look forward to hopefully seeing the recommended and most efficient solution in class.

References:

1. Python documentation on ASCII conversions: https://docs.python.org/2/library/binascii.html
2. Online XOR calculator: https://onlinehextools.com/xor-hex-numbers
3. Conceptual explanation and examples of basic crib dragging: http://travisdazell.blogspot.com/2012/11/many-time-pad-attack-crib-drag.html
4. Crib dragging experiment in Ruby: https://samwho.dev/blog/toying-with-cryptography-crib-dragging/
5. List of common English words: https://gist.github.com/deekayen/4148741