CSE/CEN 598: Hardware Security and Trust
Project 1: Cryptography
Due date: September 11th, 2023

You are the CTO of a troubled food delivery robot startup, and have are rapidly losing market share to these guys:

Yesterday, one of their robots accidentally ended up in your garage. Being a good citizen, you freed the robot and let it continue blocking sidewalks and getting lost, but not before you noticed that it dropped an SD card with all of its software, also by accident. Looking at the card, you figure out that it contains an image of a whole operating system. You spin it up in a VM, and notice that the system periodically phones home by running the `/home/cen598/Documents/project1/mothership` binary. The binary prints data to console, but it is are encrypted! You are sure that by cracking this code, you will be able to save your startup.

Looking at the encrypted data you collected, you are reasonably sure that the program is using RC4, and that the output is hex encoded (each two symbols encode 1 byte of information). Any insight that you can get from the program or the output may help save your dying startup, so you get to work.

Notes:

1. There's are 120 points to gain in this lab, but the points will saturate, so don't stress if you don't solve everything.
2. Questions are not sorted by difficulty. Answers to some questions solve other ones.
3. There are many different ways to solve this lab. While we hope to have corralled you into the solutions we're looking for, if you do find a shortcut, feel free to to describe it, as no points will be deducted for getting around our defenses.

Setting up the VM:

The machine has an account `cen598` with password `cen598`. Again, for this lab, you will find the `mothership` binary in

```
/home/cen598/Documents/project1/mothership
```

You can run the binary by entering the `~/Documents/project1` directory and running `./mothership`

Problem 1: just pen and paper for me, please (30 points):

when you run the binary as:

```
./mothership --key "<hex key>" --plaintext "<hex text>"
```

e.g.,

```
./mothership --key "1234ABCD" --plaintext "1234123412341234"
```

you seem to get an RC4 encrypted output, also in hex. However, when you run `mothership` without any inputs, it seems to use it's own inputs - maybe some internal business logic?

You notice two things about the data in the collected ciphertexts:

- The ciphertext seems to have fixed size
- The ciphertext seems to have repetitive portions

1.1: Having a dataset of ciphertexts is a good start. Collect 10+ ciphertexts which you will use later in your analysis.

1.2: There exist two interesting cryptographic nuggets that you can glean from the ciphertexts, what are they?

1.3: What is the kind of data is the program sending home? Explain your answer.

1.4: By knowing how RC4 works, there may exist some attacks that you can apply simply by collecting enough ciphertext. What can you learn from the binary's output just by analyzing and comparing the ciphertexts? Give examples and briefly explain your work.

Problem 2: helps to read the spec (30 points):

While you don't have the plaintext messages matching the collected ciphertexts, you do have that `mothership` binary. If you try to run `./mothership`, you get a neat little help text. It seems that `mothership` can be used to encrypt your inputs, assuming that you feed it text in the right format. The binary encrypts text passed in as hex values, but hex is laborious to write manually unless you had way too much free time in high school.

2.1: Write a wrapper program around the `mothership` binary that accepts ascii-encoded plaintext text (not hex-encoded), and outputs hex-encoded ciphertext.

2.2: Stress test the program a bit. Are there any limitations to the plaintext? Give examples.

2.3: The previous question was straightforward, but this one is very hard. Write a program that accepts hex-encoded cipher text, and decrypts it to ascii-encoded plaintext. The programs should cancel out, i.e., piping one program into another should give you the original input. For example, running:

```
$ ./encrypt.py "some key" "hello world!" | ./decrypt.py "some key"
"hello world!"
```

gives you the same output as input.

Problem 3: don't roll your own crypto, kids (30 points):

While the RC4 cipher *might* be fine, your competitors definitely don't know how to use it properly. They have made a couple of egregious mistakes worthy of mockery. To start, you might want to look at differential cryptanalysis, it's an amazing tool!

3.1: Running the program on some of your strings, you notice that the ciphertext is larger than the input string. It seems that the program appends something to your plaintext input before encrypting it. What is this input? Explain your steps.

3.2: Try slightly changing your inputs to the program. What do you observe, and how can this be abused? Provide examples.

3.3: Constructing specific ciphertext using the program might be a useful tool. Using the insight from above, get the program to output the following 256-byte ciphertext:

```
b4c96d3da42cc639fc37481e08bc8f30d823d7f2f5bb826561825359e75c8cf6
7db76e56646c92f0efa9bc7f8440d55e460558c787bf9b6abf5dbffa29701bb3
9683beab3972c367ea6af37746d4d89473fecea7d2a65f88d2246100f4126a62
e08ccfdf69e63c05318ccbaea9452e207427b5e0f1000a9db329d8952d7aeaa6
318161a67473151c293145b7fcf9675d2321d2e5637fe0a9fa03929da40ca32a
d225f33218957eeb80756453edb6bac2176de7e47e7d890da82b8a176277f32c
2a48f9eb50041bc46a524b26770da271e09dc3f07159ad62678654cff009d361
e4acd9d43ec875dc3590af76836d558f00992ffe37080e9c2d2b0d0362648b42
```

Choose any key you want. You are graded by how close you can approach the ciphertext.

3.4: Write and submit a program that finds the plaintext that generates arbitrary ciphertext passed as input. Choose any key you want. Your solution must be a binary that runs on the VM.

3.5 What are the mistakes that your competitors have made? How do you go about proving they exist, without access to the source? Make your case and provide examples.

Problem 4: having keys is overrated (30 points):

Based on some key, RC4 builds some (hopefully random-looking) stream which it then XORs with the plaintext. You might not know the key, but you can attempt a ciphertext-only attack to at least learn this stream. Hopefully you've collected enough ciphertext messages!

4.1: Using what you have learned above, try and predict as many bits of the stream as possible. Note that the output from the binary may be limited in size, but this is an implementational artifact and there is no guarantee that your competitor's server doesn't accept larger messages. You will be graded by how many bits you correctly guess, up to 1024 bytes. Explain your steps and provide the stream in a text file in hex format.

4.2: By now you might have guessed that the ciphertext you've recorded contains some orders sent back to your competitors servers. Construct a fake order that maximizes order value, and submit the ciphertext you arrived at. For this task, you will be graded on a scale. Again, the binary outputs fixed-size ciphertext, so you can't send infinite-sized requests. Submit a ciphertext in the report as a solution.