Warmup

Due: Thursday, February 7, 2019 @ 11:59 pm

Contents

1 Introduction

One of the focus points of CS166 is automation. It's one thing to make an attack work, but quite another to create a tool which can carry out that attack reliably, potentially accepting different configuration parameters or arguments. In the security world, attacks are only taken seriously once they can be proven to work reliably, and automation is one of the best ways to demonstrate this.

In this mini-project, you will get your feet wet with automation. The problems in this project are easy, and the solutions will be obvious. The point isn't to challenge you to figure them out, but rather to create software which can solve these problems automatically, and which work reliably. The idea is to get you comfortable with automation before you have to tackle harder problems in future projects, when worrying about the details of automation would be a distraction.

1.1 Grading

This project is graded on completion. For CS166 students, the Guess problem is worth 100% of the grade for this project—for CS162 students, the Guess problem is worth 60% of the grade and the Cipher problem is worth 40% of the grade.

1.2 Programming Language

You are required to write your solution in Go. *Prior knowledge of Go is not a prerequisite for this course!* However, later projects in this course will have components written in Go, and we'd like to get everyone

familiar with its syntax early on before moving on to more complex projects with Go components.

There are many resources both online and physical which teach Go. We won't recommend any one in particular—you can find these easily with a little Googling. Nothing that we do in this course will require particularly advanced features of Go, so simply picking up what you need on the fly from online tutorials, Stack Overflow, and so on, should be sufficient.

2 CS166: Guess

Guess is a simple number guessing game which runs as a network server listening for TCP connections. When the server is initiated, it generates a secret target number. When clients connect, they can guess numbers. If the client's guess is lower than the target number, the server will respond with the string "Up..."; if the client's guess is higher than the target number, the server will respond with "Down...". If the client guesses the correct target number, the server will respond with "Correct!".

2.1 Assignment

There's a copy of the `guess` binary at `/course/cs166/pub/warmup/guess`. It accepts a single argument: the address for the server to listen on. For example, to run the `guess` server on `localhost` at port `1234`, you can invoke the server in the following way:

```
/course/cs166/pub/warmup/guess localhost:1234
```

Your assignment is to write a program which will connect to the server and automatically send guesses to the server until it finds the correct number. Once it's guessed the correct number, your program should print that number to stdout and then terminate.

2.2 Stencil Code

In order to help you get started, we have provided stencil code for this project at `/course/cs166/pub/warmup/stencil`. The stencil code contains a `Makefile` and a `sol.go` file.

You will need to compile your Go program before running it. If you're using the stencil code, you can compile your program by running the `make client` command (or just `make`). This will build a binary named `client` from which you can run your client.

2.3 Specifics

- Your program should accept two arguments: the host and the port of the `guess` server to connect to (for example, running `./client localhost 1234` will connect your client to address `localhost:1234`). This argument format is slightly different from that of the server program.
- Your program should have proper input validation—if the wrong number of arguments are passed, or they are improperly formatted, it should print an error and exit.
- The possible target numbers are all unsigned 24-bit integers (in other words, between 0 to 16,777,215, inclusive), and your guesses should be as well. If they are not, the server will respond with an error and disconnect you.
- Each guessed number should be sent encoded as a base-10 ASCII number. Each guess should be sent on its own line (that is, followed by a newline character). Responses from the server will also be newline-separated.

3 CS162: Cipher

The following problem only needs to be completed by students enrolled in CS162. If you're not in CS162, you do not need to complete the Cipher-specific sections of the stencil code, though you should make sure your Guess program still compiles and runs without error.

The guessing game program from the previous problem can also communicate securely by encrypting commands to/from the server using a secret key shared between the server and the client. The encryption algorithm that it uses is a simple but secure cipher called XTEA (eXtended Tiny Encryption Algorithm).[1] Your task is to modify your script from the previous problem to support this encrypted protocol. This problem will help you get used to working with byte and bit manipulation and data encoding.

3.1 Specifics

- You don't need to know how XTEA works in depth; you only need to implement the algorithm correctly. Your implementation should be equivalent to the C implementation given in the Wikipedia article on XTEA.[2] Delta is **0x9E3779B9** and the number of rounds should be **64**. Be careful, though—C and Go are different languages, so making sure that the two implementations behave exactly the same can be tricky. See the troubleshooting tips below for more details.
- The keys used should be 128 bits long, just as in the C reference implementation. You should modify your program to accept a hex encoded string representing the key as the third argument. Thus, your program should work with the following invocation: `./client <host> <port> [<key>]` where `key` should be optional and should enable encryption if present.
- Internally, the XTEA key is split into 4 unsigned 32-bit integers (the reference C implementation accepts them in this format directly). For example, if your key is `0x112233445566778899aabbccddeeff`, your unsigned 32-bit integers would be `0x00112233`, `0x44556677`, `0x8899aabb`, and `0xccddeeff`. The stencil code splits the 128-bit key into the 4 integers for you—the integers can be found in the `key` field of the `XTEACipher` struct.
- The `guess` binary can be run with a key as a second argument, telling it to accept encrypted guessed and respond with encrypted responses. Thus, you can invoke the server with `guess <host>:<port> [<key>]`. You should use the same key for the server as you do for your client.
- In encrypted mode, the `guess` server speaks the same protocol as in unencrypted mode, except that each guess should be first encrypted, and the hex encoding of the encrypted guess should be sent (one ciphertext per line). The server will respond in the same way as before, except these responses will also be encrypted, and the ciphertexts sent as hex back to the client.
- Since the XTEA cipher operates on fixed-size blocks of 8 bytes, you will need to handle the case where the plaintext you want to encrypt is shorter than a full block. In this case, you have two options: pad your guess with leading zeros so that it becomes 8 bytes (for example, pad "123" to "00000123"), or send the original number, but set the remaining bytes to null (for example, pad "123" to "123\0\0\0\0\0"). The server will remove any trailing null bytes. Because the former approach only works with numbers, the server's responses (which are not numbers) are padded using the latter approach.

3.2 Help! Nothing Works! (Troubleshooting Guide)

First, you should ensure that your XTEA cipher works correctly. A particularly useful test to run is to make sure that, using randomly-generated keys and randomly-generated messages, decryption is actually the inverse of encryption (that is, $D(k, E(k, m)) = m$). When in doubt, use parentheses liberally and examine the operations one at a time to make sure they do what you expect.

[1] `https://en.wikipedia.org/wiki/XTEA`
[2] `https://en.wikipedia.org/wiki/XTEA#Implementations`

When sending guesses to the server, keep in mind that any error text will come back unencrypted. You should watch for errors, as they will help you figure out where your code may be going wrong.

Before you decrypt the server's encrypted response, be sure to remove the newline character from the ciphertext and check that the ciphertext is the correct length. Then, once the response is decrypted, you'll need to remove any null characters from the server's encrypted responses before doing any comparisons, since the string "Up..." != "Up...\0\0\0".

4 Handing In

Your handin should consist of three files: `sol.go`, a `Makefile`, and a `README`. `sol.go` should contain the source code of your Go program. Your `Makefile` should contain the necessary commands needed such that running the `make client` command compiles your `sol.go` program into a binary named `client` (the stencil code already does this for you). Your `README` should contain a high-level overview of your program.

CS166 students can hand in by running `cs166_handin cs166_warmup` from a directory containing these files. CS162 students should instead run `cs166_handin cs162_warmup`.

4.1 Late Handins

Late days will be applied automatically to this project. Please consult the syllabus for more information about late handins.

If there are extenuating circumstances preventing you from completing an assignment on time (e.g., illness), please contact the instructor before the assignment is due via the "Extension Requests" form on the `http://cs.brown.edu/courses/cs166/resources/` page.