CS162 Warmup

Due: 11:59 pm, Tuesday February 6th

Contents

1 Introduction

One of the focus points of CS162 is automation. Its 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 isnt 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.

2 Grading

This project will be graded on completion. It is not worth any part of your semester grade, but it is a prerequisite for future projects: we will refuse to grade any of your project work if you have not turned in working solutions to these problems.

3 Ruby/Python

You can write your solution in either Ruby or Python. Prior knowledge of Ruby or Python is not required for this class! Ruby and Python are each a lingua franca of security in their own right. Feel free to pick whichever language you are most comfortable in. In order to help you get started, we have provided stencil code for both languages at /course/cs166/pub/warmup/stencil.

There are many resources both online and physical which teach Ruby and Python. Nothing that we do in this course will require particularly advanced features of Ruby or Python, so simply picking up what you need on the fly from online tutorials, Stack Overflow, and so on, should be sufficient. For Ruby you can use this tutorial: `https://www.tutorialspoint.com/ruby/` and for learning Python you can try this tutorial: `https://www.tutorialspoint.com/python/`. You can also find plenty of other online tutorials on Ruby or Python by doing a little Googling on your own.

4 Problems

4.1 Guess

Guess is a simple number guessing game. It runs as a network server listening for TCP connections. When clients connect, they can guess numbers, and the server will respond with "Up..." or "Down..." Your assignment is to write a script which will connect to the server and automatically make guesses until it finds the correct number, at which point the server will respond with "Correct!" The numbers are all unsigned 24-bit integers, and your guesses should be as well. If they are not, the server will respond with an error and disconnect you.

Theres a copy of the guess binary at `/course/cs166/pub/warmup/guess`. It accepts a single argument which is the host and port to listen on (ie, localhost:1234).

Your solution should be a single ruby/python file called sol.rb/sol.py which accepts two arguments: a host, and a port (this is easier to deal with in Ruby/Python than the host:port syntax which our server accepts). Your script 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. Assuming everything works properly, your script should interact with the server and, once its guessed the correct number, print that number to stdout.

4.1.1 Specifics

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.

4.2 Cipher

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.

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

4.2.1 Specifics

- You dont 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 is a very different language from Ruby or Python, and 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 sol.rb/sol.py file to accept a hex encoded string representing the key as the third argument. Thus your program should work with the following invocation: `sol.rb <host> <port> [<key>]` or `sol.py <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 just accepts them in this format directly). You should split your 128-bit key into the 4 integers (ie, if your key is 0x112233445566778899aabbccddeeff, your unsigned 32-bit integers would be 0x00112233, 0x44556677, 0x8899aabb, and 0xccddeeff).
- 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 ruby/python 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 servers responses (which are not numbers) are padded using the latter approach.
- For an example, if you guess "1", after padding your guess will be "00000001", which in hex is:

  "\x30\x30\x30\x30\x30\x30\x30\x31"

  Use XTEA to encrypt "\x30\x30\x30\x30\x30\x30\x30\x31":

  v[0] = 0x30303030

  v[1] = 0x30303031

  After encipher it becomes:

  v[0] = 0x2bcfe596

  v[1] = 0xe966d842

  When you send v0 and v1 to the server, you should combine them as strings in their original hex format.

4.2.2 Ruby Tip

Your XTEA cipher will expect messages to be two 32-bit integers, but in this project you need to encrypt strings. Ruby has a useful method on strings, unpack[3], which can easily convert a string into an integer. Note that this is different than using a to i method since unpack can convert arbitrary ASCII strings into

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

[3] `http://ruby-doc.org/core-2.3.0/String.html\#method-i-unpack`

their binary representation and return a number representing that string. unpack requires a format string as an argument to tell it what kind of data to return and what byte ordering to use. The 32-bit unsigned network format ("N") is particularly useful for this project, as it will encode strings into a big-endian format as expected by the server. Note that unpack returns an array, so youll need to access the first element to get the number.

Some other useful Ruby methods you should look into are: `to_s()`, `sprintf` and Ruby's array slicing notation (`var[start..end]`).

In order to invert this operation, Ruby has outfitted Arrays with another method, pack[4], which also takes a format string and can reconstruct the original string from the packed array of numbers. Again, you should use the 32-bit unsigned network format for this project. In pack, you may also provide a count of elements to unpack. Your XTEA decryption method will likely return an array of two elements, on which you can then use .pack("N*") to recover a string.

If youre using Python, you can find similar functionality in the struct package.

4.2.3 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). Be particularly careful of the behavior of bit-wise operations in C vs. Ruby/Python, and of how the two languages handle overflows. When in doubt, use use parentheses liberally and examine the operations one at a time to make sure they do what you expect.

When sending guesses to the server, keep in mind that the 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.

5 Handing In

To hand in, run `cs166_handin cs162_warmup` in a directory containing your sol.rb/sol.py file.

[4] `http://ruby-doc.org/core-2.2.0/Array.html\#method-i-pack`