CSC 152 programming assignment: Chat Encrypt

Your company is making an application and wants it to support a secure chat function. You have been assigned to write two functions to handle the encryption and decryption.

```
void chat_encrypt(void *k, void *pt, int ptlen, void *ct);
void chat_decrypt(void *k, void *ct, int ctlen, void *pt);
```

The encryption function should encrypt the plaintext in pt using key k and write the result to ct along with the nonce used. The decryption function should decrypt the ciphertext in ct using key k and the nonce, and write the result to pt. The encryption is to be AES256-CTR as implemented by OpenSSL.

All buffers are preallocated and ready to use. The key `k` is 32 bytes and will be the same for encryption and decryption. The ct buffer will always be exactly 12 bytes longer than the pt buffer and should contain a 12 byte random nonce followed by the ciphertext. It is the encryption function's responsibility to generate a random nonce using a secure source of randomness such as `/dev/urandom` or OpenSSL's `RAND_bytes`. The initial counter should be zero, so the initial IV passed to OpenSSL should be 12-bytes random nonce followed by 4 bytes of zero.

To simplify your coding task, your code does not need to be robust. You may assume that each of your calls returns successfully. A production system would need to test each call for errors and react accordingly.

Name your file chat_security.c

Hints

- The documentation I use most for OpenSSL programming are:
  - OpenSSL Wiki's EVP examples. Find the closest example and study the webpage for it.
  - OpenSSL's man pages. Make sure the OpenSSL version you are using matches the man page you look at. I usually find the man page for the function using internet search (eg, search for "openssl rand_byes").
- Study the EVP code on the Wiki for symmetric encryption and decryption. Also study the cbc_encrypt.c and cbc_decrypt.c code supplied in Homework 1. I have supplied an optional OpenSSL video example that may help too.
- A good way to test your code is to make sure it's interoperable. Your decrypt should successfully decrypt the output of your encrypt.
- There were some readings in Week 1 that you might want to review.
- Before you submit your code, compile it with extra command line arguments `-Wall -Werror -fsanitize=address` and make sure it still runs correctly. The `-W` options will force you to eliminate your compiler warnings. The `-fsanitize` option will attempt to uncover illegal memory accesses by your program.

Testing and Grading

Typically the assignment is posted before the submission website is ready. You should write and self test your code until you believe it is ready. Shortly before the assignment is due, the submission website will be opened.

Programs in this class are tested against test cases. If your program behaves as the test case expects (ie, according to spec) then credit for the test case is given. Otherwise no credit is given for the test case. This credit/no-credit grading means that it is very important that you test your code well before submission.

You may assume that the inputs are error free.

Testing is a little like a game. You should try to think of all sorts of weird inputs that are legal because they don't violate the specification and verify that your code does the right thing for them. Any ambiguities in the specification should be asked about well before the due date. Throughout this class you will be expected to put some thought into test cases, ask questions, and test your code thoroughly.

The easiest test setup for an assignment that asks you to write a particular function is something like this.

```
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

int foo(void *a, int alen, void *b, int blen) {

}

#if 1     // Set to 1 while testing and 0 for submission
#include <stdio.h>
int main() {
  // Write test from your function here
  return 0;
}
#endif
```

Your main can then contain your test code. Once you're satisfied and ready to submit, set the conditional compilation to 0 to stop it.

Submission

1. The file(s) you turn in should have a small comment block at the top containing your name, the date, anyone you worked with, and the URL of any online source that provided significant help to you. It is an important habit to give credit to important sources of help. For example:

   ```
   // contains.c by Ted Krovetz. Submitted for CSC 152 July 4, 1776.
   // Pair-programmed with Ellen Watermellon
   // Used the "reverse" code from https://www.techiedelight.com/reverse-a-c-string
   ```

2. I will compile your code with gcc flags `-Wall -Werror -fsanitize=address`. Wall "enables all the warnings about constructions that some users consider questionable, and that are easy to avoid". Werror turns all warnings into errors that prevent successful compilation. Fsanitize=address causes runtime checks to detect some (but maybe not all) illegal memory accesses. You should run tests with these settings as well.
3. Here is a file that you can use to verify that your code will link to my tester correctly: chat_security_link.c. If the following compiles without errors your chat_security.c will successfully link with my testing code: `gcc -lcrypto -Wall -Werror chat_security_link.c chat_security.c` (if you get OpenSSL link errors move the -lcrypto to the end).
4. Read carefully Fileinbox submission. Follow that procedure to submit only the file chat_security.c

Collaboration

You may collaborate with *one or two* other students on this homework if you wish, or work alone. Collaboration must be true collaboration however, which means that the work put into each problem should be roughly equal and all parties should come away understanding the solution. Here are some suggested ways of collaborating on the programming part.

- Pair programming. Two or three of you look at the same screen and only one of you operate the keyboard. The one at the keyboard is the "driver" and the other is the "navigator". The driver explains everything they are doing as they do it and the navigator asks questions and makes suggestions. If the navigator knows how to solve the problem and the driver does not, the navigator should not dictate solutions to the driver but instead should tutor the driver to help them understand. The driver and navigator should switch roles every 10-15 minutes. Problems solved this way can then be individually submitted.
- Code review. The members of the collaborative each try to solve the problem independently. They then take turns analyzing each other's code, asking questions trying to understand each other's algorithms and suggesting improvements. After the code reviews, each of you can then fix your code using what you learned from the reviews. Do not copy code. If the result of code review is that your code needs changes to be more like your partner's, do not copy it. Instead recreate your own variant without looking at your partner's.

The goal is to learn enough from one another so that you each can do similar problems independently in an exam situation.

If you want a collaborator but don't know people in the class, you can ask on Discord and/or use the group-finding post on Piazza.