Challenge 1

Deadline: Feb 13, Sunday 23:59 ET

It's great that you've registered for this awesome course, but now it's time to prove you have what it takes. Welcome to your very first challenge!

This challenge is a basic knowledge check. Think of it as a rite of passage if that makes it sound cooler. You will write C code in a Linux environment, use the C Socket API for basic networking, and even get to use a cool little cryptography library.

It's likely the least interesting assignment you will do in this course, but please bear with it for now. It'll be worth it. Here are your learning goals:

- Take this as an early opportunity for a self-assessment of your C knowledge. Are you comfortable with memory management, working with C strings, and overall handling all that juicy low-level power C gives you? With great power comes great responsibility. In a couple months we will get more intimate with C, so identify your weak points now, and the rest will be a breeze.
- Learn to implement cryptography, and do it using a library that 99.99% of the population should be using instead of messing with things they don't understand.

Your Task

Log into warhead, and take a look at the TCP service running on the IP address **192.168.1.77**, port **4000**. Your task is to *nicely* ask this service for the **solution token** to this challenge. And it will tell you, simple as that!

But wait... something's wrong. It looks like there's interference in your communication channel. You *sometimes* get bad data, and *sometimes* don't. Maybe there is an attacker on the network corrupting your traffic, or maybe cosmic rays are flipping your bits on the wire, who knows? More importantly, how to tell? What if you submit the wrong solution and embarrass yourself right when you thought you could become a leet hacker?

Worry not, as this service uses a state-of-the-art secure communication channel when talking with you. Obviously, this means encrypting AND integrity-protecting the data in transfer. It should then be trivial to check if the data you receive has actually been modified on the way! Life is beautiful.

Write a C program to communicate with the server according to the protocol described below. Specifically, you have to:

1. Open a TCP connection to the service socket.
2. Follow the protocol and ask for the solution. You need to encrypt your message payload as described in the protocol.
3. When you receive a response, check its integrity. If it's good, fine. Otherwise, repeat step 2, ask again, repeat step 3, check integrity again, rinse & repeat till you get a message that's not corrupted.

So you finally got a good response, decrypted the message, and got the **token**? Good. BUT WAIT, don't submit it for grading yet! The token is a very long 128-byte value with non-printable characters and other gibberish. We want something nicer for grading. So you need to massage this token a bit first. Specifically:

1. Compute the hash of the token, which will give you shorter (but still binary) data.
2. Encode this hash value in *base64*, which will represent the same data in printable characters only.

...and you're done! That's the string you need to submit for grading.

The Crypto

Remember the golden rules of doing cryptography? (If not go review Session 1 slides, they are important rules!)

Sodium is a crypto library developed specifically with that philosophy in mind -- it does the right thing by default, and much of its API is designed to avoid giving you any options or flexibility that you'd be better off *without*. You are going to use libsodium for the cryptographic operations in this assignment. The service you will be talking to does the same, so that you can easily pass the payload you receive from it to the appropriate decryption function provided by the library.

warhead will always have the latest libsodium installed. If you plan to develop the code on your personal machine, make sure you download, compile, and use the same version, as some required functionality is not available in older releases. You can check the version number installed on warhead with the command: `pacman -Qi libsodium`

Read the Sodium documentation here, this contains all the information you need for this assignment: https://doc.libsodium.org/

Sodium's standard encryption function performs **authenticated encryption**, which is a combination of encryption and MAC, so that you don't need to do them separately! Easy, right?

To encrypt your payload, decrypt the server's payload, and do all integrity checks, you are going to use these functions: https://doc.libsodium.org/secret-key_cryptography/secretbox

Once you successfully get your token, hash it with:
https://doc.libsodium.org/hashing/generic_hashing
Set your hash output length to `crypto_generichash_BYTES`

...and finally base64 encode the hash with: https://doc.libsodium.org/helpers
Here you should use the base64 encoding with the `sodium_base64_VARIANT_ORIGINAL` variant.

As a final note, don't forget to check out the rest of the documentation, especially the **Quickstart** sections to see how you should compile your programs, what headers to include, and how to initialize the library in your code.

Your Encryption Key

Your key is under your home directory, in a file called `key`. Its size is `crypto_secretbox_KEYBYTES` bytes. The server already knows your key, so that you can talk with it. **Don't lose your key, and don't share it with anyone.**

The Protocol

Copy the below snippet into your code. These are the constants and message format you need to use.

```
#define TOKEN_SIZE 128
#define PAYLOAD_SIZE crypto_secretbox_MACBYTES + TOKEN_SIZE

#define MSG_ASK "Can I get the solution to the challenge, please?"
#define STATUS_BAD 0
#define STATUS_GOOD 1

struct message {
      int hacker_id; /* this is just the number part of the ID */
      int status;
      unsigned char nonce[crypto_secretbox_NONCEBYTES];
      unsigned char payload[PAYLOAD_SIZE];
};
```

To send a message:

1. Set `hacker_id` to N, as in hacker*N* in your lab account.
2. `status` is unused.
3. Set `nonce` to a cryptographically secure random value, see the Sodium documentation on how you can generate this value. You will use a nonce when encrypting your message, and this is how you let the server know the same value too!
4. You should always send the server `sizeof(struct message)`-sized messages. Create a `TOKEN_SIZE`-sized buffer, put `MSG_ASK` into it. Remember that this is a C string, and it should end with a null character. The contents of the rest of the buffer don't matter, but you will need to send the entire buffer anyway, as just explained.
5. Encrypt this buffer using Sodium's authenticated encryption function. Remember that you are encrypting the whole `TOKEN_SIZE`-sized buffer! This function will give you a concatenation of the resulting ciphertext with the MAC tag, so it is going to be larger than what you input, i.e., output size is `TOKEN_SIZE + crypto_secretbox_MACBYTES`. That is what `PAYLOAD_SIZE` is exactly, and this output is the value you are going to put into `payload`.
6. Send the message. If done right, you can receive a response back next.

To receive a message:

1. Read `sizeof(struct message)` bytes from the socket.
2. Check `status`.
    a. If it is `STATUS_BAD`, there was a problem with your message. Print and read the contents of `payload`, it is going to include a plaintext (not encrypted!) message explaining the problem. Getting `STATUS_BAD` is not a trick I play on you, it genuinely means that you screwed up and sent the server an incorrectly prepared message :) So read the error message and fix the problem.
    b. If it is `STATUS_GOOD`, `payload` will contain the **encrypted** token. Decrypt/integrity check the contents of `payload`, and remember that you need to use the `nonce` sent in the message by the server for that! Don't confuse the sizes here. Decryption takes the full `PAYLOAD_SIZE` buffer (it needs both the ciphertext and the MAC), but gives you a `TOKEN_SIZE` output (just the plaintext message).

So, repeat after me:

Encryption takes `TOKEN_SIZE`, outputs `PAYLOAD_SIZE`.
Decryption takes `PAYLOAD_SIZE`, outputs `TOKEN_SIZE`.

That's all Folks!

Tips and Hints

- You will use the C Socket API to communicate with the server. It is no secret in programmers' circles that this API sucks big time. Even connecting to a simple IP address is a chore. Unfortunately, we'll have to deal with that. So, if this is your first time doing sockets programming in C, don't be discouraged, look at online examples, and ask questions on Piazza. Official documentation: https://www.gnu.org/software/libc/manual/html_node/Sockets.html ...but maybe skip that and Google for tutorials. There're tons of them online.
- Before dealing with crypto, make sure you can connect to the server and communicate properly. That is, you should be able to send `sizeof(struct message)`-sized messages, and immediately read the same size response. Send the server a *correctly formatted* message with junk data, read the response. You should be able to see that the status is bad, and read the error message in the payload successfully.
- If you can send the message, but don't get a response back (i.e., your code blocks forever), you've probably sent an incorrect, short message. The server will wait to read all `sizeof(struct message)` bytes before responding.
- The server may send you corrupt payloads for a long time (i.e., the integrity check fails). Don't be afraid to try 50, 100, 150, 200, 250... times, who knows when you'll succeed ;) If I suspect that you are actually doing something wrong, wasting your time waiting too long, or otherwise DoSing the server, I will cut you off after some time with a `STATUS_BAD` and an appropriate error message. This should not happen under normal circumstances, so check that you are using Sodium correctly. **IMPORTANT:** When retrying after receiving a corrupt payload, DO NOT disconnect and reconnect to the server. You NEED to reuse the existing connection/socket. The server is designed to always return corrupt messages for a while, so if you keep disconnecting and starting the process from scratch, you will never solve the challenge.

Submission

1. Create the challenge directory tree `~/submissions/challenge1/`
2. Create `submission.txt` in the challenge directory. Copy the solution (i.e., the correct token, hashed, and base64-encoded) into this file on a single line.
3. Place your source code into the challenge directory in a file named `code.c`
4. Run `submit challenge1`

Wait a few moments, and check the results. If you got it right, pour a drink. Don't forget to check out the Hall of Fame: https://www.khoury.northeastern.edu/home/kaan/rankings.html

Good luck, and happy encrypting!

If the above doesn't make any sense, be sure to read the Lab Environment document and understand how the grading system works.