ECE568 – Computer Security

Lab #3: Two-Factor Authentication

Overview

The purpose of this lab is to familiarize yourself with both two-factor authentication and HMACs, by creating a pair of applications to both generate and validate one-time passwords that are compatible with Google Authenticator. You will need to spend some time reading and understanding the RFC specification documents that define this two-factor authentication protocol.

You may work on this lab individually or in groups of two. Your code must be entirely your own original work. The assignment should be submitted by 11:59:59pm on Sunday, November 20th. Please only submit a README file and the two ".c" files for Part 1 and Part 2:

```
submitece568s 3 README generateQRcode.c validateQRcode.c
```

Background

Google Authenticator uses an open-source specification that defines two types of "one-time passwords":

- **Time-based:** Time-based One-Time Password (TOTP) algorithm; and,
- **Ticket-Based:** HMAC-based One-Time Password (HOTP) algorithm.

A time-based password (TOTP) automatically expires after a fixed amount of time (30 seconds, by default), at which point it is automatically replaced with new password. Ticket-based passwords (HOTP) require the user to confirm when a password has been used, at which point that password is expired and a new one is generated. HOTP is specificed in the RFC 4226 standard document, and TOTP is specified in RFC 6238.

Both RFC documents can be found in the …/doc/ directory, alongside the code for the lab. You will need to refer to both of the RFC documents for an explanation of how the passwords are formed, including how the HMAC is calculated. (Note that both TOTP and HOTP use identical HMAC calculations: the only difference is whether they include *the time* or *a counter* in the hashed value. As a result, you do not need to implement multiple HMAC implementations.)

Part 1: Generating an otpauth:// URI

The Google Authenticator app (and many free clones) are available for free on all major smartphone platforms, and provide a convenient way for users to add two-factor authentication to many online services.

Your service provider (Google Mail, Dropbox, GitHub, etc.) will normally generate a *secret key* for your account; this *secret key* needs to be communicated securely to the app running on your smartphone, and must then be protected (otherwise, anyone who discovers the secret can bypass your two-factor authentication). Normally, it is encoded and displayed as a 2D barcode on your screen, which the app then scans to securely read the *secret key* with minimal risk of interception.

This process starts by encoding the secret key in a special URI (similar in structure to a URL used for web browsing). The two formats we will be using in this lab are:

```
    otpauth://hotp/ACCOUNTNAME?issuer=ISSUER&secret=SECRET&counter=1
    otpauth://totp/ACCOUNTNAME?issuer=ISSUER&secret=SECRET&period=30
```

The first form is used to encode a HOTP key, while the second form is used for a TOTP key.
Each of the two forms has several parameters we must fill in:

- **ACCOUNTNAME:** The name of the account (e.g., "gibson"). Any special characters in this string (like spaces) should be properly encoded. (I have provided a `urlEncode()` function that you may use for this purpose.)
- **ISSUER:** The name of the service (e.g., "Facebook"). As with the previous field, all special characters must be encoded. (*e.g.*, "U of T" becomes "U%20of%20T".)
- **SECRET:** The 80-bit secret key value, encoded in Base-32. (I have provided a `base32_encode()` function that you may use for this purpose.) If fewer than 20 hex characters are provided by the user, then you should pad the value with leading zeros and treat it as an 80-bit number.

You are to finish writing the generateQRcode.c program (please keep all of your code in this one file). That generates these URIs and the associated barcodes. (I have provided a basic function for printing the properly-formatted barcodes on the screen.)

When you run the generateQRcode program, it should produce output in (exactly) the following format, including the barcodes:

```
$ ./generateQRcode ECE568 gibson 12345678901234567890

Issuer: ECE568
Account Name: gibson
Secret (Hex): 12345678901234567890

otpauth://hotp/gibson?issuer=ECE568&secret=CI2FM6EQCI2FM6EQ&counter=1
```

```
otpauth://totp/gibson?issuer=ECE568&secret=CI2FM6EQCI2FM6EQ&period=30
```

If you have access to a device that supports the Google Authenticator app (or any compatible app), you can scan the barcodes from your screen and use your mobile app to generate codes for the second part of this assignment. Otherwise, you can use the pre-compiled application in the …/util/ directory to generate appropriate values for you, as you need them for testing:

```
$ ./util/generateValues 12345678901234567890
HOTP value: 803282
TOTP value: 892402
```

Part 2: Validating the Codes

In the second part of this lab, you are to complete the code in validateQRcode.c in order to have it generate the HOTP and TOTP values from the *secret* and then verify whether the user has provided correct values. Please ensure your program creates output in exactly the following form:

```
$ ./validateQRcode 12345678901234567890 803282 134318

Secret (Hex): 12345678901234567890
HTOP Value: 803282 (valid)
TOTP Value: 134318 (valid)
```

Your program should print “invalid” instead of “valid” if either user-provided value is incorrect.

In order to verify the values, you will need to use the provided SHA1 function to create an HMAC. This is the same style of HMAC that we reviewed in class; please see the RFC docs included in the lab for the description of the inner/outer padding, and how to truncate the HMAC to only six characters for the output. (Note that, when calculating the HMAC, you should be including the *secret* in its binary form – not as a hexidecimal or base32 string!)

The provided SHA1 functions can be used in the following manner:

```
SHA1_INFO        ctx;
uint8_t          sha[SHA1_DIGEST_LENGTH];

sha1_init(&ctx);
sha1_update(&ctx, data, dataLength);
// keep calling sha1_update if you have more data to hash...
```

```
      sha1_final(&ctx, sha);
```

The final call to `sha1_final()` will write the SHA1 hash of the data (in a binary form) into the `sha[]` array (which you can then use in your HMAC calculation).

General

For the purposes of this lab, you can assume that all data inputs will be properly-formatted. We will not be testing your code with invalid characters, missing inputs or inputs that are too long.