ECE568 – Computer Security

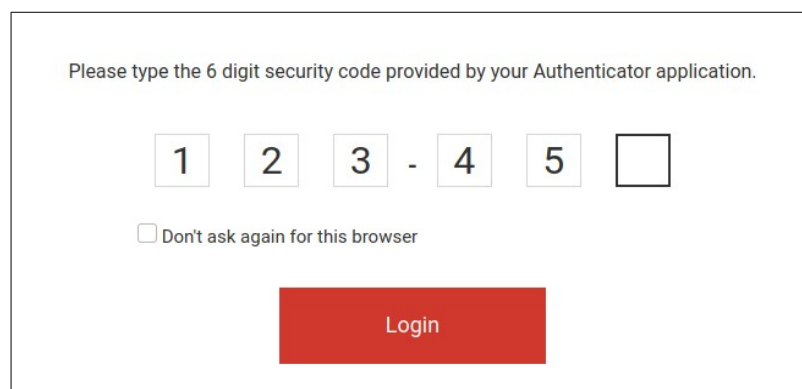# Lab #2: Two-Factor Authentication

Overview

The purpose of this lab is to familiarize yourself with mobile *multi-factor authentication* (MFA), by creating a pair of applications that use a mobile device to authenticate a simulated "login" session. 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 Friday, February 17th. Please copy your completed files into a new directory and only submit a README file, the two ".c" files and your *mobile_mfa.py* for Part 1 and Part 2:

```
submitece568s 2 README generateQRcode.c validateQRcode.c
        mobile_mfa.py
```

Part 1: TOTP (Google Authenticator)

Google Authenticator uses an open-source specification that defines two types of "one-time passwords" that can be used to provide a strong layer of security to online accounts:

Formally, the standard includes two different algorithms for generating one-time passwords:

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

We will be looking at the first of these, **TOTP**, in Part 1 of this lab assignment[1]. A time-based password (TOTP) generates a six-digit number that the user can enter along with their password; it automatically expires after a fixed amount of time (30 seconds, by default), at which point it is automatically replaced with new password. This "second factor" provides good protection against phishing and password compromise.

---

[1] HOTP is very similar, from a technical perspective – but creates a more-complicated user experience; as a result, it is seldom used in practice.)

The TOTP specification is described in RFC 6238. A copy of the TOTP spec (RFC 6238) can be found in the *…/part1/doc/* directory, alongside the code for the lab. You will need to refer to the RFC documents for an explanation of how the one-time passwords are formed, including how the HMAC is calculated. (Other sources, like Wikipedia, may be useful as well.)

Step 1.1: 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 that secret key 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 format we will be using in this lab are:

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

There are 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 "URL-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**%20**of**%20**T".)
- **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.) For simplicity, please assume that all secrets will be provided to your application as (exactly) 20-character Base-32 values, with all letters in uppercase: we will **not** test with other, invalid input.

You are to finish writing the generateQRcode.c program (please keep all of your code in this one file) to generate 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://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
TOTP value: 892402
```

(Note that the TOTP value is time-dependent: *your* output will be different when you run the command, and it will change to a new value every 30 seconds.)

Step 1.2: 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 TOTP value 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 134318

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

Your program should print "**invalid**" instead of "valid" if the 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 <u>binary</u> 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).

Part 2: Biometric Multi-Factor Authentication

As the risk to on-line transactions increases, more advanced forms of multi-factor authentication (MFA) are starting to appear. A number of MFA solutions, like Duo, Okta, Ping and Microsoft Authenticator, are becoming very popular to protect logins to VPN services, and other critical enterprise applications. In this section, you will be working with a next-generation *Biometric Multi-Factor Authenticator* that is being used by a number of financial institutions to secure wire transfers, and other high-value transactions.

Given the need for sensitivity around biometric information, I'll start with a couple of notes on privacy for this lab assignment:

1. You should always be cautious about how and where your biometric information is being recorded, and how it will be used – so, I'd like to start with some assurance related to this lab. In this assignment, your biometric information will only be recorded (in a secure format) on *your own* mobile device, and it will <u>not</u> be transmitted anywhere else. Neither I, nor anyone else, will have access to your biometric data, or any recordings, at any time: they will remain private, and under your control. You will delete all of this data at the end of the lab, when you delete the app off your phone.

2. *Please* follow the privacy instructions, detailed below, when you install the app. This will ensure that your location data is not captured. (Financial institutions use behavioural information, like your location and movements, for additional anti-fraud purposes; to preserve everyone's privacy, we will *not* be using this feature in this lab.)

If you have any questions/concerns, please do not hesitate to ask; this is a security course, and taking care of your own on-line security is an important part of that.

With that out of the way...

Mobile authentication services work by providing a second layer of authentication that is *out-of-band* (i.e., using a different device than your desktop/laptop). This makes it significantly harder to for an attacker to compromise your login – because they would need to steal your password that you enter on your desktop *and* they would also need to compromise the app on your mobile phone. A normal login flow with a MFA looks like this:

1. The user, on their laptop/desktop, attempts to log into their *service provider*;
2. That *service provider* makes an API call to the MFA authentication service (Duo, Microsoft Authenticator, etc.);
3. The MFA authentication service connects to the user's mobile phone, asks for permission for the login to continue, and potentially performs some step to verify the user's identity;
4. The *service provider* makes periodic calls to the MFA authentication service, to see if the user has approved the login;
5. Once the *service provider* knows that the user has successfully approved the login, the laptop/desktop is permitted to connect.

The goals for this part of the lab are to familiarize you with three important aspects of typical APIs that are used for securing logins with Multi-Factor Authentication:

1. Using REST API calls to communicate with a cloud service;
2. The process for authenticating with a cloud-based service platform; and,
3. The process for provisioning a commercial MFA authenticator for an "employee" (you, in this case), and then using that authenticator to validate a simulated "login" session.

IMPORTANT: As with any MFA solution, you require "administrator" permissions to create, modify and delete users. For this lab, everyone is being given "administrator" rights to an actual, production MFA account that has been set up for us to all use for this lab assignment. I am trusting that everyone will, please, not abuse this access. (Attempting to alter the account settings will disrupt the ability for everyone to access the cloud resources and complete this lab.) Please **DO NOT** attempt to intentionally access the service (or make changes to the account settings) in ways that aren't required to complete this lab. As a particular note: please do not attempt to change the credentials contained in *.../setup/server_config.py*. (Repeatedly attempting to log in with invalid credentials will lock out access, preventing everyone from accessing it: please don't do this.) (Thank you!)

Part 2.1: API Message Flow

There are three steps to configuring and using the mobile MFA application:

1. **Download the application onto a mobile device.** We will be using the BioConnect mobile authenticator for this lab assignment. (Important disclaimer: this product is produced by my company, but that gives me some added ability to ensure both end-user privacy and API access for this tool.) On either an iOS or an Android device, visit the app store and download the free BioConnect Mobile app. (There is no charge for this app.)
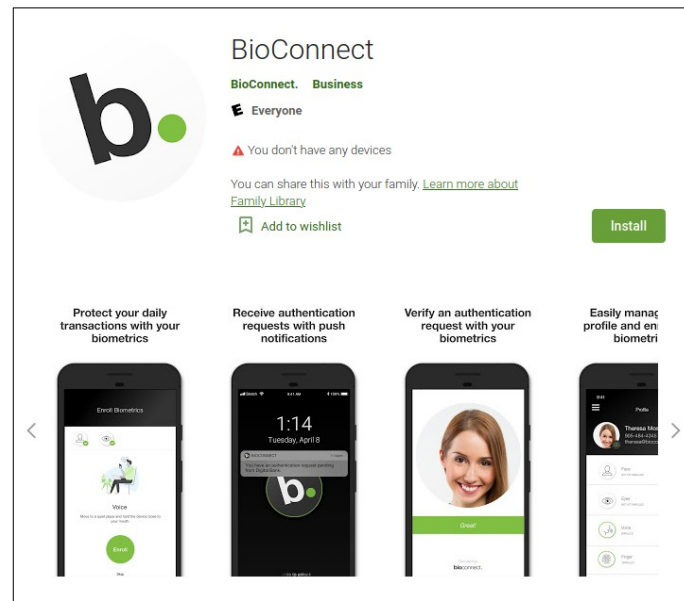
I'm aware that we have a few students who are outside of Canada, studying remotely at the moment; if you are currently in a country that does not have the app in your local App Store, please email me (courtney.gibson@utoronto.ca) with your Apple ID / Play Store ID (your login name only: not your password!) and I can arrange for you to obtain a private copy of the app through TestFlight.

The app will ask for permission to access the camera and microphone; this will be used to enrol your biometric data onto the phone, so you can identify yourself and authenticate your "login". (None of this will be sent off of your phone or available to anyone other than you.)

IMPORTANT: You will be prompted at some point to allow the app to access your *location information*. While this is an important anti-fraud tool for commercial applications, it is <u>not required for this lab</u>. To protect your privacy, please **decline this**.

2. **Set up your ECF environment.** The ECF environment is missing a couple of the libraries that will be useful for this lab. A script is provided in the ".../setup/" directory (included with the rest of the lab files) that will install your own local copies of the required Python libraries:

```
$ cd setup
$ ./setup.py
$ cd ..
```

3. **Register your mobile device with the cloud service.** Now that you have set up your environment, you have been provided with a Python script that will log into the cloud service and start a process to securely pair your mobile device:

```
$   cd part2
$   ./mobile_mfa.py
```

The script will display a QR code on the screen. (You may need to shrink the font on your terminal window for the QR code to display properly.) If you open the app and scan the QR code, it will start the pairing process on the phone. You will be prompted to enrol yourself in a few different ways that can later be used to securely identify yourself.

At this point you are now ready to start writing to code that will allow your own (simulated) login service to start using mobile phones to authenticate your users...

Part 2.2: Validating a login

In order to use the mobile phone to validate a login session, you will need to complete the code for three functions inside of *mobile_mfa.py*:

getAuthenticatorStatus

This function connects to the cloud service to check whether the user has successfully activated their mobile phone. You do this by doing a GET call to:

https://.../v2/users/**<userId>**/authenticators/**<authenticatorId>**

The value of *userId* is set in *BioConnect.createUser()*, and the value of *authenticatorId* is set in *BioConnect.createAuthenticator()*. The function requires the same API keys set in the header as the other API calls (see the other functions provided in mobile_mfa.py as a reference).

The API call will return a JSON structure that returns a number of values, describing the current status of the mobile device, including:

{[…],"status":"active","face_status":"enrolled","voice_status":"enrolled","fingerprint_status":"enrolled","eye_status":"enrolled",[...]}

Your code should only report that the device is "active" when the *status* is "active" **and** when at least one of the biometric modalities (*face_status, voice_status, fingerprint_status, eye_status*) is "enrolled".

sendStepup

This function connects to the cloud service and pushes an authentication request (a "stepUp") to the mobile phone. You do this by doing a POST call to:

> https://.../v2/user_verifications

The PUT call needs to send a few parameters to the cloud service:

> user_uuid   = <userId>
> transaction_id = <randomString>
> message   = "Login request"

The value of *userId* is set in *BioConnect.createUser*(), and the same value should be sent as the *user_uuid* in this call. The "transaction_id" can be any random string. Please use "Login request" as your *message*.

The API call will return a JSON structure that returns a number of values, describing the authentication request that was just created, including the verificationId:

> {"user_verification":{"uuid":"1TNCPQYNN5J3AA7J0BR0SCASBW","status":"pending", [...]}

You will need to store this value for use in the next step.

getStepupStatus

This function connects to the cloud and checks if the user has successfully responded to the verification request and authenticated themselves on their mobile phone. You do this by doing a GET call to:

> https://.../v2/user_verifications/<verificationId>

The API call will return a JSON structure that returns a number of values, describing the current status of the verification request, including:

> {"user_verification":{"uuid":"1TNCPQYNN5J3AA7J0BR0SCASBW","status":"pending", [...]}

The status will be one of:

- **pending:** Still waiting for the user to respond (and the "login" should wait)
- **success:** The user has successfully authenticated (and the "login" should proceed)
- **declined:** The user has either failed or declined (and the "login" should fail)
- **expired:** The user did not respond in time (and the "login" should fail)

Once these functions are created, you should be able to successfully simulate a "login", using your mobile device to authenticate the session:

```
login: ece568
password: password
login successful
```

General

For the purposes of (both parts) 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, invalid HTTP replies, or inputs that are too long.