# Project 5: Web Crawler

Interact with web services via HTTP

**This project is due at 11:59pm on Monday, March 25, 2024.**

## Description

This assignment is intended to familiarize you with the HTTP protocol. HTTP is (arguably) the most important application level protocol on the Internet today: the Web runs on HTTP, and increasingly other applications use HTTP as well (including Bittorrent, streaming video, Facebook and Twitter's social APIs, etc.).

Your goal in this assignment is to implement a web crawler that gathers data from a fake social networking website that we have set up for you. There are several educational goals of this project:

- To expose you to the HTTP protocol, which underlies a large (and growing) number of applications and services today.
- To let you see how web pages are structured using HTML.
- To give you experience implementing a client for a well-specified network protocol.
- To have you understand how web crawlers work, and how they are used to implement popular web services today.

## What is a Web Crawler?

A web crawler (sometimes known as a robot, a spider, or a scraper) is a piece of software that automatically gathers and traverses documents on the web. For example, lets say you have a crawler and you tell it to start at https://www.wikipedia.com. The software will first download the Wikipedia homepage, then it will parse the HTML and locate all hyperlinks (i.e., anchor tags) embedded in the page. The crawler then downloads all the HTML pages specified by the URLs on the homepage, and parses them looking for more hyperlinks. This process continues until all of the pages on Wikipedia are downloaded and parsed.

Web crawlers are a fundamental component of today's web. For example, Googlebot is Google's web crawler. Googlebot is constantly scouring the web, downloading pages in search of new and updated content. All of this data forms the backbone of Google's search engine infrastructure.

## Fakebook

We have set up a fake social network for this project called Fakebook. Fakebook is a very simple website that consists of the following pages:

- **Homepage**: The Fakebook homepage displays some welcome text, as well as links to several random Fakebook users' personal profiles.
- **Personal Profiles**: Each Fakebook user has a profile page that includes their name, some basic demographic information, as well as a link to their list of friends.
- **Friends List**: Each Fakebook user is friends with one or more other Fakebook users. This page lists the user's friends and has links to their personal profiles.

WARNING: DO NOT TEST YOUR CRAWLERS ON PUBLIC WEBSITES

Many web server administrators view crawlers as a nuisance, and they get very mad if they see strange crawlers traversing their sites. **Only test your crawler against Fakebook, do not test it against any other websites**.

## Username and Password

Your username for Fakebook is your **Khoury** username (not your Northeastern username). To get your password, you must SSH to the Khoury login machine (*login-students.khoury.northeastern.edu*, or one of its many clones) and run a command provided by Khoury systems. This is the same command you used to obtain your credentials for Project 2 (FTP client):

```
$ 4700-ftp-password [Your NUID]
```

**Input your NUID with leading zeroes.** The output of this command is a long string that is your password for Fakebook. Here is a full example of SSHing to the Khoury login machine and generating a Fakebook password:

```
cbw@clx-ubuntu:~$ ssh cbw@login-students.khoury.northeastern.edu
[cbw@login-students ~] 4700-ftp-password 001234567
Your password is: 9c2bf5c59edb601cc01a52c1f3452ddf4bd87f86f27bc87bc6839bc067057eb4
```

# High-level Requirements

Your goal is to collect *5 secret flags* that have been hidden somewhere on the Fakebook website. The flags are unique for each student, and the pages that contain the flags will be different for each student. Since you have no idea what pages the secret flags will appear on, and the Fakebook site is very large (tens of thousands of pages), your only option is to write a web crawler that will traverse Fakebook and locate your flags.

Your web crawler must execute on the command line using the following syntax:

```
$ ./crawler <-s server> <-p port> <username> <password>
```

The `-s` and `-p` arguments are each optional and they represent the server and port your code should crawl, respectively. If either or both are not provided, you should use `www.3700.network` for the server and 443 for the port. The arguments `username` and `password` are used by your crawler to login to Fakebook. You may assume that the root page for Fakebook is available at `https://<server>:<port>/fakebook/`. You may also assume that the login form for Fakebook is available at `https://<server>:<port>/accounts/login/?next=/fakebook/`.

Your web crawler should print **exactly fives lines of output to STDOUT**: the five *secret flags* discovered during the crawl of Fakebook, each terminated by a `\n` character. Your web crawler should not print out anything other than those five flags. If your program encounters an unrecoverable error, it may print an error message before terminating.

**Secret flags may be hidden on any page on Fakebook, their exact location on each page may be different, and pages may contain multiple flags**. Each secret flag is a 64 character long sequences of random alphanumerics. All secret flags will appear in the following format (which makes them easy to identify):

```
<h3 class='secret_flag' style="color:red">FLAG: 64-characters-of-random-alphanumerics</h3>
```

# Sockets, Ports, and TLS

Fakebook uses HTTPS, which means that the full protocol stack is HTTP over TLS over TCP. Thus, in this project, your web crawler will need to connect to Fakebook using a TCP socket wrapped in TLS. Note that in HTTPS, the TCP socket gets wrapped in TLS immediately after connection, before any HTTP protocol messages are sent. This is similar to the TLS version of the simple client you wrote in Project 1, where you needed to implement both a non-secure and secure version of your client.

By convention, HTTP uses TCP port 80 and HTTPS uses port 443. Thus, in this project, you will be connecting to Fakebook on port 443 unless the `-p` option is specified.

# HTTP and (Il)Legal Libraries

and parse HTML. However, you may not use **any** libraries/modules/etc. that implement HTTP or manage cookies for you. You may also not use any all-in-one scrapers, such as BeautifulSoup.

For example, if you were to write your crawler in Python, the following modules would all be allowed: *socket*, *urllib.parse*, *html*, *html.parser*, and *xml*. However, the following modules would **not** be allowed: *urllib*, *urllib2*, *httplib*, *requests*, *pycurl*, and *cookielib*.

Similarly, if you were to write your crawler in Java, it would **not be legal** to use *java.net.CookieHandler*, *java.net.CookieManager*, *java.net.HttpCookie*, *java.net.HttpUrlConnection*, *java.net.URLConnection*, *URL.openConnection()*, *URL.openStream()*, or *URL.getContent()*.

If students have any questions about the legality of any libraries please post them to Piazza. It is much safer to ask ahead of time, rather than turn in code that uses a questionable library and receive points off for the assignment after the fact.

# Starter code

Very basic starter code for the assignment in Python is available on the Khoury GitHub server. Provided is a simple implementation of a client that simply fetches the root page of the site using HTTP 1.0. Because it does not implement TLS, it receives an error form the server. To get started, you should create a copy of the starter code on your local machine (or on the Khoury Linux machines if you wish):

```
git clone https://github.khoury.northeastern.edu/cs3700/crawler-starter-code.git
```

## Implementation Details and Hints

In this assignment, your crawler must implement HTTP/1.1 (not 0.9 or 1.0). This means that there are certain HTTP headers like `Host` that you must include in your requests (i.e., they are required for all HTTP/1.1 requests). We encourage you to implement `Connection: Keep-Alive` (i.e., pipelining) to improve your crawler's performance (and lighten the load on our server), but this is not required, and it is tricky to get correct. We also encourage students to implement `Accept-Encoding: gzip` (i.e., compressed HTTP responses), since this will also improve performance for everyone, but this is also not required. If you want to get crazy, you can definitely speed up your crawler by using multithreading or multiprocessing, but again this is not required functionality.

> WARNING: PLEASE BE KIND AND LIMIT YOUR PARALLELISM
>
> While we do our best to keep our web servers well-provisioned so they can handle the load generated by a full class of students, the servers can easily crumble in the face of crawlers that are massively parallel, i.e., send many requests in parallel at the same time. If you plan to implement parallelism (e.g., through async IO, threads, or multiprocessing) we ask that you limit your parallelism to five (5) requests at a time. This will enable you to build and test your parallel implementation while still ensuring that the web server has enough bandwidth left over to serve other students.

One of the key differences between HTTP/1.0 and HTTP/1.1 is that the latter supports *chunked encoding*. HTTP/1.1 servers may break up large responses into chunks, and it is the client's responsibility to reconstruct the data by combining the chunks. Our server may return chunked responses, which means your client must be able to reconstruct them. To aid in debugging, you might consider using HTTP/1.0 for your initial implementation; once you have a working 1.0 implementation, you can switch to 1.1 and add support for chunked responses.

In order to build a successful web crawler, you will need to handle several different aspects of the HTTP protocol:

- HTTP `GET` - These requests are necessary for downloading HTML pages.
- HTTP `POST` - You will need to implement HTTP `POST` so that your code can login to Fakebook. As shown above, you will pass a username and password to your crawler on the command line. The crawler will then use these values as parameters in an HTTP `POST` in order to log-in to Fakebook.

- Cookie Management - Fakebook uses cookies to track whether clients are logged in to the site. If your crawler fetches the Fakebook login page, Fakebook will return a session cookie to your crawler. Your crawler should store this cookie, and submit it along with each HTTP `GET` request as it crawls Fakebook. If your crawler fails to handle cookies properly, then your software will not be able to successfully crawl Fakebook.

In addition to crawling Fakebook, your web crawler must be able to correctly handle HTTP status codes. Obviously, you need to handle `200 - OK`, since that means everything is okay. Your code must also handle:

- `302 - Found`: This is as an HTTP redirect. Your crawler should try the request again using the new URL given by the server in the *Location* header.
- `403 - Forbidden` and `404 - Not Found`: Our web server may return these codes in order to trip up your crawler. In this case, your crawler should abandon the URL that generated the error code.
- `503 - Service Unavailable`: Our web server may **randomly** return this error code to your crawler. In this case, your crawler should re-try the request for the URL until the request is successful.

If you encounter HTTP `500` errors, please contact the course staff. These may be legitimate bugs in our server software and are not intentional.

I highly recommend the HTTP Made Really Easy tutorial as a starting place for students to learn about the HTTP protocol. Furthermore, the developer tools built-in to Chrome and Firefox are both excellent for inspecting and understanding HTTP requests.

In addition to HTTP-specific issues, there are a few key things that all web crawlers must do in order function:

- **Track the Frontier**: As your crawler traverses Fakebook it will observe many URLs. Typically, these uncrawled URLs are stored in a queue, stack, or list until the crawler is ready to visit them. These uncrawled URLs are known as the frontier.
- **Watch Out for Loops**: Your crawler needs to keep track of where it has been, i.e., the URLs that it has already crawled. Obviously, it isn't efficient to revisit the same pages over and over again. If your crawler does not keep track of where it has been, it will almost certainly enter an infinite loop. For example, if users A and B are friends on Fakebook, then that means A's page links to B, and B's page links to A. Unless the crawler is smart, it will ping-pong back and forth going A->B, B->A, A->B, B->A, ..., etc. If you find that your crawler does not find all of its flags and keeps running, you are likely stuck in such a loop.
- **Only Crawl The Target Domain**: Web pages may include links that point to arbitrary domains (e.g., a link on google.com that points to cnn.com). **Your crawler should only traverse URLs that point to pages on the specified server (e.g., www.3700.network, if no `-s` is provided)**. For example, it would be valid to crawl *https://www.3700.network/fakebook/018912/*, but it would not be valid to crawl *https://www.facebook.com/018912/*. Your code should check to make sure that each URL has a valid domain before you attempt to visit it.

## Logging in to Fakebook

In order to write code that can successfully log-in to Fakebook, you will need to reverse engineer the HTML form on the log-in page. **Students should carefully inspect the form's code, since it may not be as simple as it initially appears.** The key acronym you should be on the lookout for is *CSRF*.

## Language

You can write your code in whatever language you choose, as long as your code compiles and runs on Gradescope. Do not use libraries that are disallowed for this project. Similarly, your code must compile and run on the command line. You may use IDEs (e.g., Eclipse) during development, but do not turn in your project without a Makefile. Make sure you code has **no dependencies** on your IDE. We provide starter code in Python; you are welcome to use this, but if you decide use another language, you will need to port the starter code to your language yourself.

# Grading

The grading in this project will consist of

| Item | Percentage of Grade |
| --- | --- |
| Program correctness | 70% |
| Correct flags submitted | 15% |
| Style and documentation | 15% |

## Submitting Your Project

To turn-in your project, you must submit the following four things:

1. The thoroughly documented source code for your crawler. It should be named `crawler`, or if you are using a compiled language, your Makefile should compile an executable `crawler`.
2. A `Makefile` that compiles your code.
3. A plain-text (no Word or PDF) `README.md` file. In this file, you should briefly describe your high-level approach, any challenges you faced, and an overview of how you tested your code.
4. A file called `secret_flags`. This file should contain the *secret flags* of all group members, one per line, in plain ASCII. For example, a group of two should have a file with exactly ten lines in it.

Your `README.md`, `Makefile`, `secret_flags` file, source code, etc. should all be placed in the root of a compressed archive (e.g., a .zip or .tar.gz) and then uploaded to Gradescope. Do not use the archive option on a Mac – Gradescope does not parse these archives. Instead, use the `zip` program on the command line. Alternatively, you can check **all** these items into Github, download a zip file from Github, and submit that to Gradescope.

← Project 4: Reliable Transport Protocol

Project 6: Content Delivery Network →

**GENERAL**

Syllabus

Schedule

**PROJECTS**

Project 1: Socket Basics

Project 2: FTP Client

Project 3: BGP Router

Project 4: Reliable Transport Protocol

Project 5: Web Crawler

Project 6: Content Delivery Network