Chapter 4. Finding Subdomains

To scope out and test API endpoints, you should first be familiar with the domain structure a web application uses. Today it is rare for a single domain to be used to serve a web application in its entirety. More often than not, web applications will be split into client and server domains at minimum, plus the well-known *https://www* versus just *https://*. Being able to iteratively find and record subdomains powering a web application is a useful first recon technique against that web application.

Multiple Applications per Domain

Let's assume we are trying to map MegaBank's web applications in order to better perform a black-box penetration test sponsored by that bank. We know that MegaBank has an app that users can log in to and access their bank accounts. This app is located at *https://www.mega-bank.com*.

We are particularly curious if MegaBank has any other internet-accessible servers linked to the *mega-bank.com* domain name. We know MegaBank has a bug bounty program, and the scope of that program covers the main *mega-bank.com* domain quite comprehensively. As a result, any easy-to-find vulnerabilities in *mega-bank.com* have already been fixed or reported. If new ones pop up, we will be working against the clock to find them before the bug bounty hunters do.

Because of this, we would like to look for some easier targets that still allow us to hit MegaBank where it hurts. This is a purely ethical corporate-sponsored test, but that doesn't mean we can't have any fun.

The first thing we should do is perform some recon and fill our web application map with a list of subdomains attached to *mega-bank.com* (see

Figure 4-1). Because *www* points to the public-facing web application itself, we probably don't have any interest in that. But most large consumer companies actually host a variety of subdomains attached to their primary domain. These subdomains are used for hosting a variety of services from email, to admin applications, file servers, and more.

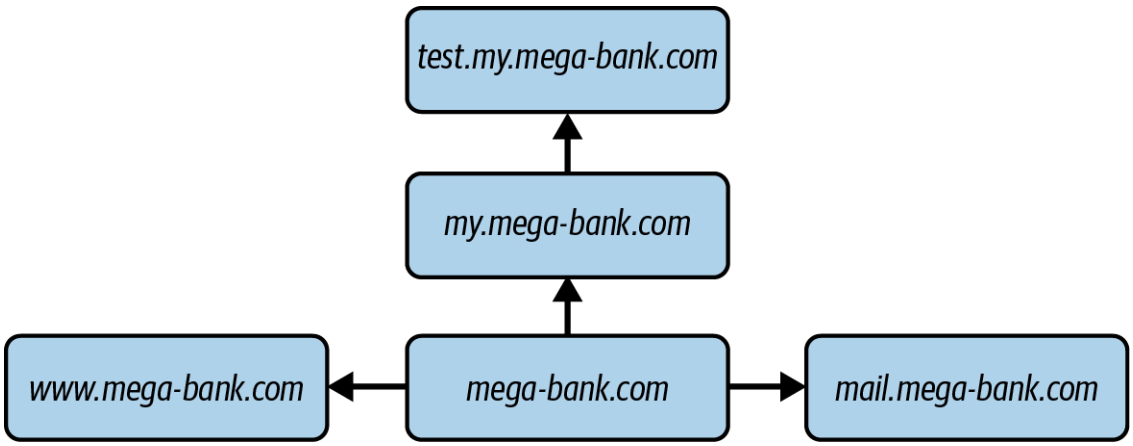

Figure 4-1. Mega-bank.com simple subdomain web—often these webs are significantly more complex and may contain servers not accessible from an external network

There are many ways to find this data, and often you will have to try several to get the results you are looking for. We will start with the most simple methods and work our way up.

The Browser's Built-In Network Analysis Tools

Initially, we can gather some useful data simply by walking through the visible functionality in MegaBank and seeing what API requests are made in the background. This will often grant us a few low-hanging fruit endpoints. To view these requests as they are being made, we can use our own web browser's network tools or a more powerful tool like Burp or ZAP.

Figure 4-2 shows an example of Wikipedia browser developer tools, which can be used to view, modify, resend, and record network requests. Freely available network analysis tools such as this are much more powerful than many paid network tools from 10 years ago. Because this book

is written excluding specialized tools, we will rely solely on the browser for now.

As long as you are using one of the three major browsers (Chrome, Firefox, or Edge), you should find that the tools included with them for developers are extremely powerful. In fact, browser developer tools have come so far that you can easily become a proficient hacker without having to purchase any third-party tools. Modern browsers provide tooling for network analysis, code analysis, runtime analysis of JavaScript with breakpoints and file references, accurate performance measurement (which can also be used as a hacking tool in side-channel attacks), as well as tools for performing minor security and compatibility audits.

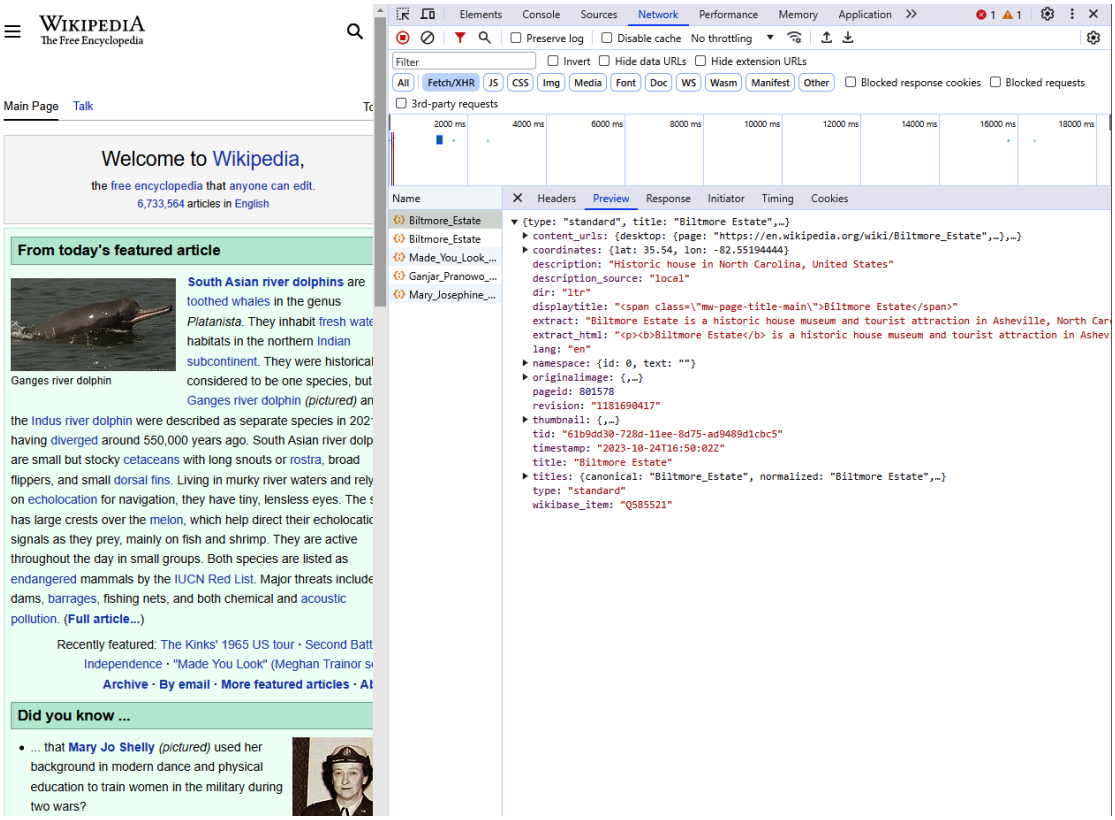

Figure 4-2. The Wikipedia.org browser developer tools network tab showing an async HTTP request made to the Wikipedia API

To analyze the network traffic going through your browser, do the following (in Chrome):

1. Click the triple dots on the top right of the navigation bar to open the Settings menu.
2. Under “More tools” click “Developer tools.”

3. At the top of this menu, click the “Network” tab. If it is not visible, expand the developer tools horizontally until it is.

Now try navigating across the pages in any website while the Network tab is open. Note that new HTTP requests will pop up, alongside a number of other requests (see Figure 4-3).

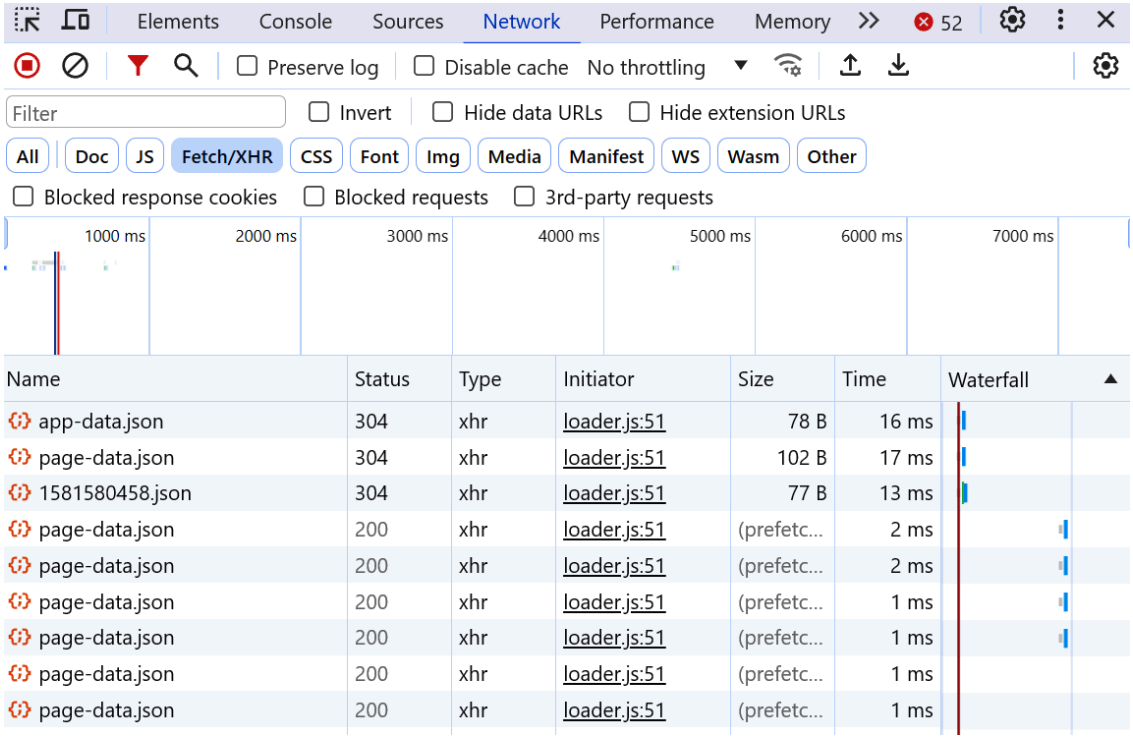

Figure 4-3. Network tab, used for analyzing network traffic that flows to and from your web browser

You can use the Network tab in the browser to see all of the network traffic the browser is handling. For a larger site, it can become quite intimidating to filter through.

Often the most interesting results come from the XHR tab, under the Network tab, which will show you any HTTP POST, GET, PUT, DELETE, and other requests made against a server, and filter out fonts, images, videos, and dependency files. You can click any individual request in the lefthand pane to view more details.

Clicking one of these requests will bring up the raw and formatted versions of the request, including any request headers and body. In the Preview tab that appears when a request is selected, you will be able to see a pretty-formatted version of the result of any API request.

The Response tab under XHR will show you a raw response payload, and the Timing tab will show you very particular metrics on the queuing, downloading, and waiting times associated with a request. These performance metrics are actually very important as they can be used to find side-channel attacks (an attack that relies on a secondary metric other than a response to gauge what code is running on a server; for example, load time between two scripts on a server that are both called via the same endpoint).

By now you should have enough familiarity with the browser Network tab to start poking around and making use of it for recon. The tooling is intimidating, but it isn't actually that hard to learn.

As you navigate through any website, you can check the request → headers → general → request URL to see what domain a request was sent to or a response was sent from. Often this is all you need to find the affiliated servers of a primary website.

Taking Advantage of Public Records

Today the amount of publicly available information stored on the web is so huge that an accidental data leak can slip through the cracks without notice for years. A good hacker can take advantage of this fact and find many interesting tidbits of information that could lead to an easy attack down the line.

Some data that I've found on the web while performing penetration tests in the past includes:

- Cached copies of GitHub repos that were accidentally turned public before being turned private again
- SSH keys
- Various keys for services like Amazon AWS or Stripe that were exposed periodically and then removed from a public-facing web application
- DNS listings and URLs that were not intended for a public audience
- Pages detailing unreleased products that were not intended to be live

- Financial records hosted on the web but not intended to be crawled by a search engine
- Email addresses, phone numbers, and usernames

This information can be found in many places, such as:

- Search engines
- Social media posts
- Archiving applications, like *archive.org*
- Image searches and reverse image searches

When attempting to find subdomains, public records can also be a good source of information because subdomains may not be easily found via a dictionary, but could have been indexed in one of the services previously listed.

Search Engine Caches

Google is the most commonly used search engine in the world, and it is often thought to have indexed more data than any other search engine. By itself, a Google search would not be useful for manual recon due to the huge amount of data you would have to sift through in order to find anything of value. This is furthered by the fact that Google has cracked down on automated requests and rejects requests that do not closely mimic that of a true web browser.

Fortunately, Google offers special search operators for power searchers that allow you to increase the specificity of your search query. We can use the `site:<my-site>` operator to ask Google to only query against a specific domain:

```
site:mega-bank.com log in
```

Doing this against a popular site will usually return pages upon pages of content from the main domain and very little content from the interesting subdomains. You will need to improve the focus of your search further to start uncovering any interesting stuff.

Use the minus operator to add specific negative conditions to any query string. For example, `-inurl:<pattern>` will reject any URLs that match the pattern supplied. Figure 4-4 shows an example of a search that combines the Google search operators `site:` and `-inurl:<pattern>`. By combining these two operators we can ask Google to return only *Wikipedia.org* web pages that are about puppies while leaving out any that contain the word "dog" in their URL. This technique can be used to reduce the number of search results returned and to search specific subdomains while ignoring specific keywords. Mastery of Google's search operators and operators in other search engines will allow you to find information not easily discovered otherwise.

We can use the operator `-inurl:<pattern>` to remove results for the subdomains we are already familiar with, like www. Note that it will also filter out instances of www from other parts of a URL, as it does not specify the subdomain but the whole URL string instead. This means that *https://admin.mega-bank.com/www* would be filtered as well, which means there could be false positive removals:

```
site:mega-bank.com -inurl:www
```

You can try this against many sites, and you will find subdomains you didn't even think existed. For example, let's try it against the popular news site Reddit:

```
site:reddit.com -inurl:www
```

The first result from this query will be *code.reddit.com*—an archive of code used in the early versions of Reddit that the staff decided to make available to the public. Websites like Reddit purposefully expose these domains to the public.

Figure 4-4. A Google.com search that combines the Google search operators `site:` *and* `-inurl:<pattern>`

For our pen test against MegaBank, if we find additional domains that are purposefully exposed and not of interest to us, we will simply filter them out as well. If MegaBank had a mobile version hosted under the subdomain *mobile.mega-bank.com*, we could easily filter that out as well:

```
site:mega-bank.com -inurl:www -inurl:mobile
```

When attempting to find subdomains for a given site, you can repeat this process until you don't find any more relevant results. It may also be beneficial to try these techniques against other search engines like Bing—the large search engines all support similar operators.

Record anything interesting you have found via this technique and then move on to other subdomain recon methods.

Accidental Archives

Public archiving utilities like *Archive.org* are useful because they build snapshots of websites periodically and allow you to visit a copy of a website from the past. *Archive.org* strives to preserve the history of the internet, as many sites die and new sites take their domains. Because *Archive.org* stores historical snapshots of websites, sometimes dating back 20 years, the website is a goldmine for finding information that was once disclosed (purposefully or accidentally) but later removed. The particular screenshot in Figure 4-5 is the home page of *Wikipedia.org* indexed in 2003—over two decades ago!

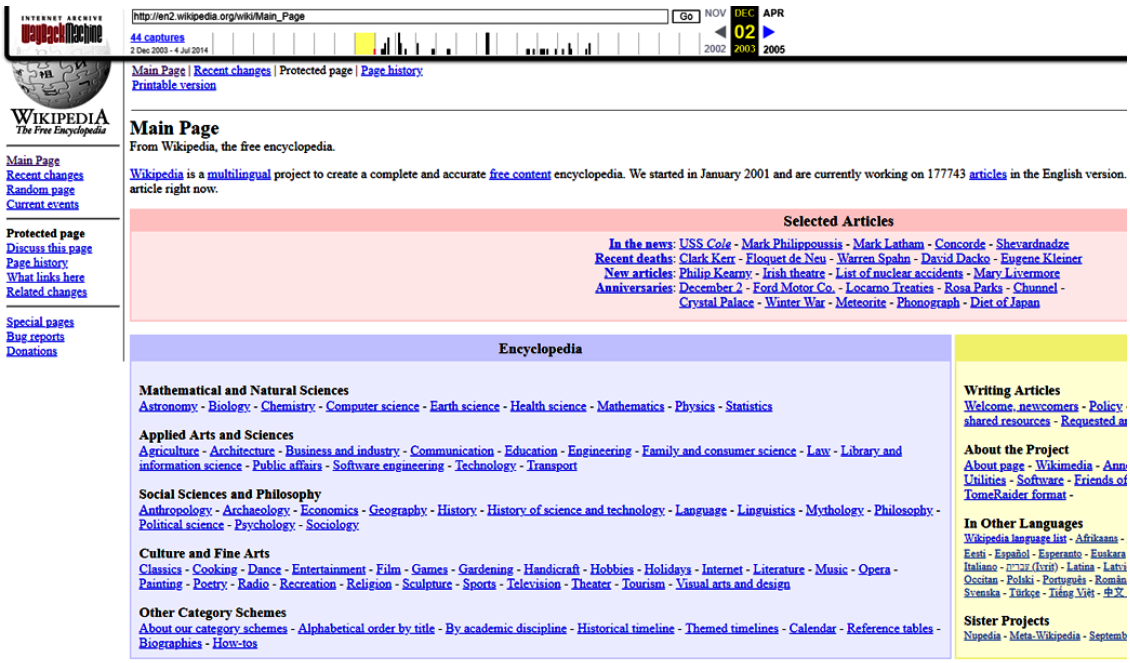

Figure 4-5. Archive.org, a San Francisco-based nonprofit that has been around since 1996

Generally speaking, search engines will index data regarding a website but try to crawl that website periodically to keep their cache up to date.

This means that you should look in a search engine for relevant *current* data, but for relevant *historical* data, you might be better off looking at a website archive.

The *New York Times* is one of the most popular web-based media companies by traffic. If we look up its main website on *Archive.org* (*https://www-nytimes-com.ezproxy.sfpl.org*), we will find that *Archive.org* has saved over 500,000 snapshots of the front page between 1996 and today.

Historical snapshots are particularly valuable if we know or can guess a point in time when a web application shipped a major release or had a serious security vulnerability disclosed. When looking for subdomains, historical archives often disclose these via hyperlinks that were once exposed through the HTML or JS but are no longer visible in the live app.

If we right-click on an *Archive.org* snapshot in our browser and select "View source," we can do a quick search for common URL patterns. A search for *file://* might pull up a previously live download, while a search for *https://* or *http://* should bring up all of the HTTP hyperlinks.

We can automate subdomain discovery from an archive with these simple steps:

1. Open 10 archives from 10 separate dates with significant time in between.
2. Right-click "View source," then press Ctrl-A to highlight all HTML.
3. Press Ctrl-C to copy the HTML to your clipboard.
4. Create a file on your desktop named *legacy-source.html*.
5. Press Ctrl-V to paste the source code from an archive into the file.
6. Repeat this for each of the nine other archives you opened.
7. Open this file in your favorite text editor (VIM, Atom, VSCode, etc.).
8. Perform searches for the most common URL schemes:
    - *http://*
    - *https://*
    - *file://*
    - *ftp://*
    - *ftps://*

You can find a full list of browser-supported URL schemes in the specification document, which is used across all major browsers to define which schemes should be supported.

Social Snapshots

Every major social media website today makes its money from the sale of user data. Depending on the platform, this can include public posts, private posts, and even direct messages in some cases.

Unfortunately, today's major social media companies go to great efforts to convince users that their most private data is secure. This is often done through marketing messages that describe the great lengths undertaken to keep customers' data out of reach. However, this is often only said in order to assist in attracting and maintaining active users. Very few countries have laws and lawmakers modernized enough to enforce the legitimacy of any of these claims. It is likely that many users of these sites do not fully understand what data is being shared, by what methods it is being shared, and for what goals this data is being consumed.

Finding subdomains for a company-sponsored pen test via social media data would not be found unethical by most. However, I implore you to consider the end user when you use these APIs in the future for more targeted recon.

For the sake of simplicity, we will take a look at the Twitter API as a recon example. Keep in mind, however, that every major social media company offers a similar suite of APIs typically following a similar API structure. The concepts required to query and search through tweet data from the Twitter API can be applied to any other major social media network.

Twitter API

X, the company formerly known as Twitter, has a number of offerings for searching and filtering through its data (see Figure 4-6).

These offerings differ in scope, feature set, and data set. This means the more data you want access to and the more ways you wish to request and

filter that data, the more you will have to pay. In some cases, searches can even be performed against X's servers instead of locally. Keep in mind that doing this for malicious purposes is probably against X's ToS, so this usage should be restricted to white hat only.

At the very bottom tier, X offers a trial "search API" that allows you to sift through 30 days' worth of tweets, provided you request no more than 100 tweets per query and query no more than 30 times per minute. With the free tier API, your total monthly queries are also capped at 250. It will take about 10 minutes' worth of queries to acquire the maximum monthly data set offered at this tier. This means you can only analyze 25,000 tweets without paying for a more advanced membership tier.

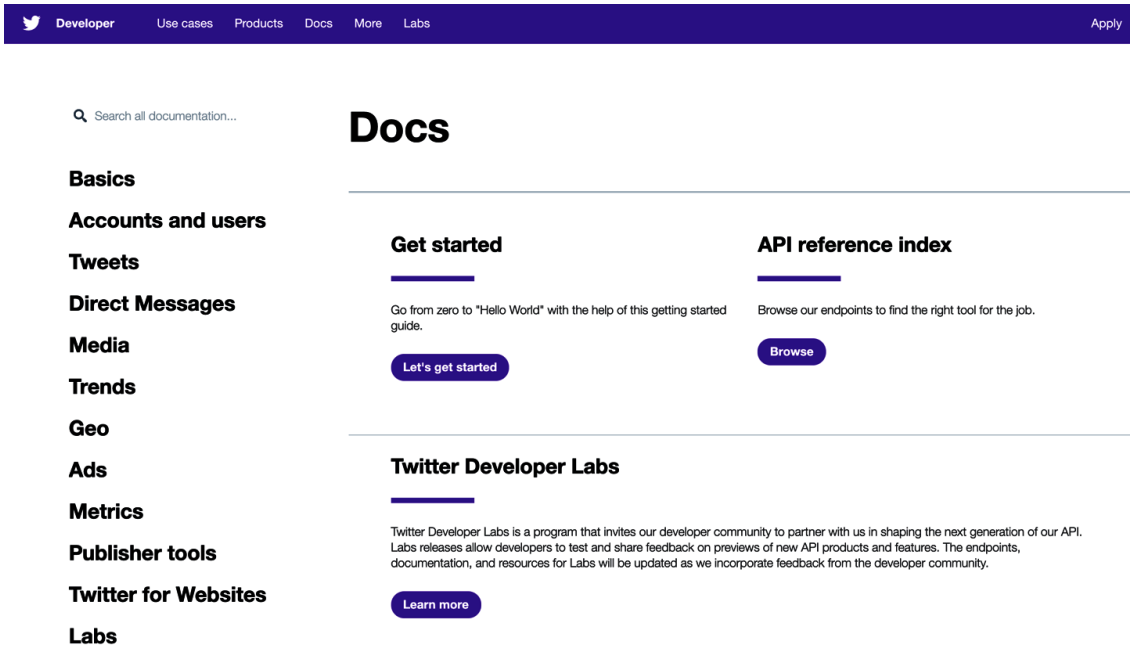

Figure 4-6. X's API developer docs will quickstart your ability to search and filter through user data

These limitations can make coding tools to analyze the API a bit difficult. If you require X for recon in a work-sponsored project, you may want to consider upgrading or looking at other data sources.

We can use this API to build a JSON that contains links to **.mega-bank.com* in order to further our subdomain recon. To begin querying against the X search API, you will need the following:

- A registered developer account
- A registered app

- A bearer token to include in your requests in order to authenticate yourself

Querying this API is quite simple, although the documentation is scattered and at times hard to understand due to lack of examples:

```
curl --request POST \
  --url https://api.twitter.com/1.1/tweets/search/30day/Prod.json \
  --header 'authorization: Bearer <MY_TOKEN>' \
  --header 'content-type: application/json' \
  --data '{
           "maxResults": "100",
           "keyword": "mega-bank.com"
           }'
```

By default, this API performs fuzzy searching against keywords. For exact matches, you must ensure that the transmitted string itself is enclosed in double quotes. Double quotes can be sent over valid JSON in the following form: `"keyword": "\"mega-bank.com\""`.

Recording the results of this API and searching for links may lead to the discovery of previously unknown subdomains. These typically come from marketing campaigns, ad trackers, and even hiring events that are tied to a different server than the main app.

For a real-life example, try to construct a query that would request tweets regarding Microsoft. After sifting through enough tweets, you will note that Microsoft has a number of subdomains it actively promotes on X, including:

- *careers.microsoft.com* (a job posting site)
- *office.microsoft.com* (the home of Microsoft Office)
- *powerbi.microsoft.com* (the home of the PowerBI product)
- *support.microsoft.com* (Microsoft customer support)

Note that if a tweet becomes popular enough, major search engines will begin indexing it. So analyzing the Twitter API will be more relevant if you are looking for less popular tweets. Highly popular viral tweets will

be indexed by search engines due to the amount of inbound links. This means sometimes it is more effective to simply query against a search engine using the correct operators, as discussed previously in this chapter.

Should the results of this API not be sufficient for your recon project, X also offers two other APIs: streaming and firehose.

X's streaming API provides a live stream of current tweets to analyze in real time; however, this API only offers a very small percentage of the actual live tweets as the volume is too large to process and send to a developer in real time. This means that at any given time you could be missing more than 99% of the tweets. If an app you are researching is trending or massively popular, this API could be beneficial. If you are doing recon for a startup, this API won't be of much use to you.

X's firehose API operates similarly to the streaming API, but guarantees delivery of 100% of the tweets matching a criteria you provide. This is typically much more valuable than the streaming API for recon, as we prefer relevancy over quantity in most situations.

To conclude, when using X as a recon tool, follow these rules:

- For most web applications, querying the search API will give you the most relevant data for recon.
- Large-scale apps, or apps that are trending, may have useful information to be found in the firehose or streaming APIs.
- If historical information is acceptable for your situation, considering downloading a large historical data dump of tweets and querying locally against those instead.

Remember, almost all major social media sites offer data APIs that can be used for recon or other forms of analysis. If one doesn't give you the results you are looking for, another may.

Zone Transfer Attacks

Walking through a public-facing web app and analyzing network requests will only get you so far. We also want to find the subdomains attached to MegaBank that are not linked to the public web app in any way.

A *zone transfer attack* is a kind of recon trick that works against improperly configured *Domain Name System* (DNS) servers. It's not really a "hack," although its name would imply it is. Instead, it's just an information-gathering technique that takes little effort to use and can give us some valuable information if it is successful. At its core, a DNS zone transfer attack is a specially formatted request on behalf of an individual that is designed to look like a valid DNS zone transfer request from a valid DNS server.

DNS servers are responsible for translating human-readable domain names (e.g., *https://mega-bank.com*) to machine-readable IP addresses (e.g., 195.250.100.195), which are hierarchical and stored using a common pattern so they can be easily requested and traversed. DNS servers are valuable because they allow the IP address of a server to change without having to update the application users on that server. In other words, a user can continually visit *https://www.mega-bank.com* without worrying about which server the request will resolve to.

The DNS system is very dependent on its ability to synchronize DNS record updates with other DNS servers. DNS zone transfers are a standardized way that DNS servers can share DNS records. Records are shared in a text-based format known as a *zone file*.

Zone files often contain DNS configuration data that is not intended to be easily accessible. As a result, a properly configured DNS primary server should only be able to resolve zone transfer requests that are requested by another authorized DNS secondary server. If a DNS server is not properly configured to only resolve requests for other specifically defined DNS servers, it will be vulnerable to bad actors.

To summarize, if we wish to attempt a zone transfer attack against MegaBank, we need to pretend we are a DNS server and request a DNS zone file as if we needed it in order to update our own records. We need to first find the DNS servers associated with *https://www.mega-bank.com*. We can do this very easily in any Unix-based system from the terminal:

```
host -t mega-bank.com
```

The command `host` refers to a DNS lookup utility that you can find in most Linux distros as well as in recent versions of macOS. The `-t` flag specifies we want to request the nameservers that are responsible for resolving *mega-bank.com*.

The output from this command would look something like this:

```
mega-bank.com name server ns1.bankhost.com
mega-bank.com name server ns2.bankhost.com
```

The strings we are interested in from this result are `ns1.bankhost.com` and `ns2.bankhost.com`. These refer to the two nameservers that resolve for *mega-bank.com*.

Attempting to make a zone transfer request with `host` is very simple and should only take one line:

```
host -l mega-bank.com ns1.bankhost.com
```

Here the `-l` flag suggests we wish to get a zone transfer file for *mega-bank.com* from `ns1.bankhost.com` in order to update our records. If the request is successful, indicating an improperly secured DNS server, you would see a result like this:

```
Using domain server:
Name: ns1.bankhost.com
Address: 195.11.100.25
Aliases:
```

```
mega-bank.com has address 195.250.100.195
mega-bank.com name server ns1.bankhost.com
mega-bank.com name server ns2.bankhost.com
mail.mega-bank.com has address 82.31.105.140
admin.mega-bank.com has address 32.45.105.144
internal.mega-bank.com has address 25.44.105.144
```

From these results, you now have a list of other web applications hosted under the *mega-bank.com* domain, as well as their public IP addresses!

You could even try navigating to those subdomains or IP addresses to see what resolves. With a little bit of luck, you have greatly broadened your attack surface!

Unfortunately, DNS zone transfer attacks don't always go as planned like in the preceding example. A properly configured server will give a different output when you request a zone transfer:

```
Using domain server:
Name: ns1.secure-bank.com
Address: 141.122.34.45
Aliases:

: Transfer Failed.
```

The zone transfer attack is easy to stop, and you will find that many applications are properly configured to reject these attempts. However, because attempting a zone transfer attack only takes a few lines of Bash, it is almost always worth trying. If it succeeds, you get a number of interesting subdomains that you may not have found otherwise.

Brute Forcing Subdomains

As a final measure in discovering subdomains, brute force tactics can be used. These can be effective against web applications with few security mechanisms in place; however, against more established and secure web

applications, we will find that our brute force must be structured very intelligently. Brute forcing subdomains should be our last resort as brute force attempts are easily logged and often extremely time-consuming due to rate limitations, regex, and other simple security mechanisms developed to prevent such types of snooping.

WARNING

Brute force attacks are very easy to detect and could result in your IP addresses being logged or banned by the server or its admin.

Brute forcing implies testing every possible combination of subdomains until we find a match. With subdomains, there can be many possible matches, so stopping at the first match may not be sufficient.

First, let's stop to consider that unlike a local brute force, a brute force of subdomains against a target domain requires network connectivity. Because we must perform this brute force remotely, our attempts will be further slowed due to network latency. Generally speaking, you can expect anywhere between 50 and 250 ms latency per network request.

This means we should make our requests asynchronous and fire them all off as rapidly as possible rather than waiting for the prior response. Doing this will dramatically reduce the time required for our brute force to complete.

The feedback loop required for detecting a live subdomain is quite simple. Our brute force algorithm generates a subdomain, and we fire off a request to `<subdomain-guess>.mega-bank.com`. If we receive a response, we mark it as a live subdomain. Otherwise, we mark it as an unused subdomain.

Because the book you are reading is titled *Web Application Security*, the most important language for us to be familiar with for this context is JavaScript. JavaScript is not only the sole programming language currently available for client-side scripting in the web browser, but also an

extremely powerful backend server-side language thanks to Node.js and the open source community.

Let's build up a brute force algorithm in two steps using JavaScript. Our script should do the following:

1. Generate a list of potential subdomains.
2. Run through that list of subdomains, pinging each time to detect if a subdomain is live.
3. Record the live subdomains and do nothing with the unused subdomains.

We can generate subdomains using the following:

```
/*
 * A simple function for brute forcing a list of subdomains
 * given a maximum length of each subdomain.
 */
const generateSubdomains = function(length) {

    /*
     * A list of characters from which to generate subdomains.
     *
     * This can be altered to include less common characters
     * like '-'.
     *
     * Chinese, Arabic, and Latin characters are also
     * supported by some browsers.
     */
    const charset = 'abcdefghijklmnopqrstuvwxyz'.split('');
    let subdomains = charset;
    let subdomain;
    let letter;
    let temp;

    /*
     * Time Complexity: o(n*m)
     * n = length of string
     * m = number of valid characters
     */
    for (let i = 1; i < length; i++) {
```

```
            temp = [];
            for (let k = 0; k < subdomains.length; k++) {
              subdomain = subdomains[k];
              for (let m = 0; m < charset.length; m++) {
                letter = charset[m];
                temp.push(subdomain + letter);
              }
            }
            subdomains = temp
        }

        return subdomains;
    }

    const subdomains = generateSubdomains(4);
```

This script will generate every possible combination of characters of length `n`, where the list of characters to assemble subdomains from is `charset`. The algorithm works by splitting the `charset` string into an array of characters, then assigning the initial set of characters to that array of characters.

Next, we iterate for duration `length`, creating a temporary storage array at each iteration. Then we iterate for each subdomain and each character in the `charset` array that specifies our available character set. Finally, we build up the temp array using combinations of existing subdomains and letters.

Now, using this list of subdomains, we can begin querying against a top-level domain (*.com*, *.org*, *.net*, etc.) like *mega-bank.com*. In order to do so, we will write a short script that takes advantage of the DNS library provided within Node.js—a popular JavaScript runtime.

To run this script, you just need a recent version of Node.js installed on your environment (provided it is a Unix-based environment like Linux or Ubuntu):

```
const dns = require('dns');
const promises = [];
```

```
/*
 * This list can be filled with the previous brute force
 * script, or use a dictionary of common subdomains.
 */
const subdomains = [];

/*
 * Iterate through each subdomain, and perform an asynchronous
 * DNS query against each subdomain.
 *
 * This is much more performant than the more common `dns.lookup()`
 * because `dns.lookup()` appears asynchronous from the JavaScript,
 * but relies on the operating system's getaddrinfo(3) which is
 * implemented synchronously.
 */
subdomains.forEach((subdomain) => {
  promises.push(new Promise((resolve, reject) => {
    dns.resolve(`${subdomain}.mega-bank.com`, function (err, ip) {
      return resolve({ subdomain: subdomain, ip: ip });
    });
  }));
});

// after all of the DNS queries have completed, log the results
Promise.all(promises).then(function(results) {
  results.forEach((result) => {
    if (!!result.ip) {
      console.log(result);
    }
  });
});
```

In this script, we do several things to improve the clarity and performance of the brute forcing code. First import the Node DNS library. Then we create an array `promises`, which will store a list of `promise` objects. `Promises` are a much simpler way of dealing with asynchronous requests in JavaScript and are supported natively in every major web browser and Node.js.

After this, we create another array called `subdomains`, which should be populated with the subdomains we generated from our first script (we will combine the two scripts together at the end of this section). Next, we use the `forEach()` operator to easily iterate through each subdomain in the `subdomains` array. This is equivalent to a `for` iteration, but syntactically more elegant.

At each level in the subdomain iteration, we push a new `promise` object to the `promises` array. In this `promise` object, we make a call to `dns.resolve`, which is a function in the Node.js DNS library that attempts to resolve a domain name to an IP address. These `promises` we push to the `promise` array only resolve once the DNS library has finished its network request.

Finally, the `Promise.all` block takes an array of `promise` objects and results (calls `.then()`) only when every `promise` in the array has been resolved (completed its network request). The double `!!` operator in the result specifies we only want results that come back defined, so we should ignore attempts that return no IP address.

If we included a condition that called `reject()`, we would also need a `catch()` block at the end to handle errors. The DNS library throws a number of errors, some of which may not be worth interrupting our brute force for. This was left out of the example for simplicity's sake but would be a good exercise if you intend to take this example further.

Additionally, we are using `dns.resolve` versus `dns.lookup` because although the JavaScript implementation of both resolve asynchronously (regardless of the order they were fired), the native implementation that `dns.lookup` relies on is built on `libuv`, which performs the operations synchronously.

We can combine the two scripts into one program very easily. First, we generate our list of potential subdomains, and then we perform our asynchronous brute force attempt at resolving subdomains:

```
const dns = require('dns');
```

```
/*
 * A simple function for brute forcing a list of subdomains
 * given a maximum length of each subdomain.
 */
const generateSubdomains = function(length) {

    /*
     * A list of characters from which to generate subdomains.
     *
     * This can be altered to include less common characters
     * like '-'.
     *
     * Chinese, Arabic, and Latin characters are also
     * supported by some browsers.
     */
    const charset = 'abcdefghijklmnopqrstuvwxyz'.split('');
    let subdomains = charset;
    let subdomain;
    let letter;
    let temp;

    /*
     * Time Complexity: o(n*m)
     * n = length of string
     * m = number of valid characters
     */
    for (let i = 1; i < length; i++) {
        temp = [];
        for (let k = 0; k < subdomains.length; k++) {
          subdomain = subdomains[k];
          for (let m = 0; m < charset.length; m++) {
            letter = charset[m];
            temp.push(subdomain + letter);
          }
        }
        subdomains = temp
    }

    return subdomains;
}

const subdomains = generateSubdomains(4);
const promises = [];
```

```
/*
 * Iterate through each subdomain, and perform an asynchronous
 * DNS query against each subdomain.
 *
 * This is much more performant than the more common `dns.lookup()`
 * because `dns.lookup()` appears asynchronous from the JavaScript,
 * but relies on the operating system's getaddrinfo(3), which is
 * implemented synchronously.
 */
subdomains.forEach((subdomain) => {
  promises.push(new Promise((resolve, reject) => {
    dns.resolve(`${subdomain}.mega-bank.com`, function (err, ip) {
      return resolve({ subdomain: subdomain, ip: ip });
    });
  }));
});

// after all of the DNS queries have completed, log the results
Promise.all(promises).then(function(results) {
  results.forEach((result) => {
    if (!!result.ip) {
      console.log(result);
    }
  });
});
```

After a short period of waiting, we will see a list of valid subdomains in the terminal:

```
{ subdomain: 'mail', ip: '12.32.244.156' },
{ subdomain: 'admin', ip: '123.42.12.222' },
{ subdomain: 'dev', ip: '12.21.240.117' },
{ subdomain: 'test', ip: '14.34.27.119' },
{ subdomain: 'www', ip: '12.14.220.224' },
{ subdomain: 'shop', ip: '128.127.244.11' },
{ subdomain: 'ftp', ip: '12.31.222.212' },
{ subdomain: 'forum', ip: '14.15.78.136' }
```

Dictionary Attacks

Rather than attempting every possible subdomain, we can speed up the process further by utilizing a *dictionary attack* instead of a brute force attack. Much like a brute force attack, a dictionary attack iterates through a wide array of potential subdomains, but instead of randomly generating them, they are pulled from a list of the most common subdomains.

Dictionary attacks are much faster and will usually find you something of interest. Only the most peculiar and nonstandard subdomains will be hidden from a dictionary attack.

A popular open source DNS scanner called dnscan ships with a list of the most popular subdomains on the internet, based off of millions of subdomains from over 86,000 DNS zone records. According to the subdomain scan data from `dnscan`, the top 25 most common subdomains are as follows:

- `www`
- `mail`
- `ftp`
- `localhost`
- `webmail`
- `smtp`
- `pop`
- `ns1`
- `webdisk`
- `ns2`
- `cpanel`
- `whm`
- `autodiscover`
- `autoconfig`
- `m`
- `imap`
- `test`
- `ns`
- `blog`

- `pop3`
- `dev`
- `www2`
- `admin`
- `forum`
- `news`

The `dnscan` repository on GitHub hosts files containing the top 10,000 subdomains that can be integrated into your recon process thanks to its very open GNU v3 license. You can find `dnscan`'s subdomain lists and source code on GitHub.

We can easily plug a dictionary like `dnscan` into our script. For smaller lists, you can simply copy/paste/hardcode the strings into the script. For large lists, like `dnscan`'s 10,000 subdomain list, we should keep the data separate from the script and pull it in at runtime. This will make it much easier to modify the subdomain list, or make use of other subdomain lists. Most of these lists will be in *.csv* format, making integration into your subdomain recon script very simple:

```
const dns = require('dns');
const csv = require('csv-parser');
const fs = require('fs');

const promises = [];

/*
 * Begin streaming the subdomain data from disk (versus
 * pulling it all into memory at once, in case it is a large file).
 *
 * On each line, call `dns.resolve` to query the subdomain and
 * check if it exists. Store these promises in the `promises` array.
 *
 * When all lines have been read, and all promises have been resolved,
 * then log the subdomains found to the console.
 *
 * Performance Upgrade: if the subdomains list is exceptionally large,
 * then a second file should be opened and the results should be
 * streamed to that file whenever a promise resolves.
 */
```

```
fs.createReadStream('subdomains-10000.txt')
  .pipe(csv())
  .on('data', (subdomain) => {
    promises.push(new Promise((resolve, reject) => {
      dns.resolve(`${subdomain}.mega-bank.com`, function (err, ip) {
        return resolve({ subdomain: subdomain, ip: ip });
      });
    }));
  })
  .on('end', () => {

   // after all of the DNS queries have completed, log the results
   Promise.all(promises).then(function(results) {
     results.forEach((result) => {
       if (!!result.ip) {
         console.log(result);
       }
     });
   });
  });
```

Yes, it is that simple! If you can find a solid dictionary of subdomains (it's just one search away), you can just paste it into the brute force script, and now you have a dictionary attack script to use as well. Because the dictionary approach is much more efficient than the brute force approach, it may be wise to begin with a dictionary and then use a brute force subdomain generation only if the dictionary does not return the results you are seeking.

Summary

When performing recon against a web application, the main goal should be to build a map of the application that can be used later when prioritizing and deploying attack payloads. An initial component of this search is understanding what servers are responsible for keeping an application functioning—hence our search for subdomains attached to the main domain of an application.

Consumer-facing domains, such as the client of a banking website, usually get the most scrutiny. Bugs will be fixed rapidly, as visitors are exposed to them on a daily basis.

Servers that run behind the scenes, like a mail server or admin backdoor, are often riddled with bugs, as they have much less use and exposure. Often, finding one of these “behind-the-scenes” APIs can be a beneficial jump start when searching for vulnerabilities to exploit in an application.

A number of techniques should be used when trying to find subdomains, as one technique may not provide comprehensive results. Once you believe you have performed sufficient reconnaissance and have collected a few subdomains for the domain you are testing against, you can move on to other recon techniques—but you are always welcome to come back and look for more if you are not having luck with more obvious attack vectors.