5

VULNERABILITY SCANNING AND FUZZING

In Chapter 4, we identified hosts on a network and a few running services, including HTTP, FTP, and SSH. Each of these protocols has its own set of tests we could perform. In this chapter, we'll use specialized tools on the discovered services to find out as much as we can about them.

In the process, we'll use bash to run security testing tools, parse their output, and write custom scripts to scale security testing across many URLs. We'll fuzz with tools such as ffuf and Wfuzz, write custom security checks using the Nuclei templating system, extract personally identifiable information (PII) from the output of tools, and create our own quick-and-dirty vulnerability scanners.

Scanning Websites with Nikto

Nikto is a web scanning tool available on Kali. It performs banner grabbing and runs a few basic checks to determine if the web server uses security

headers to mitigate known web vulnerabilities; these vulnerabilities include *cross-site scripting (XSS)*, which is a client-side injection vulnerability targeting web browsers, and *UI redressing* (also known as *clickjacking*), a vulnerability that lets attackers use decoy layers in a web page to hijack user clicks. The security headers indicate to browsers what to do when loading certain resources and opening URLs, protecting the user from falling victim to an attack.

After performing these security checks, Nikto also sends requests to possible endpoints on the server by using its built-in wordlist of common paths. The requests can discover interesting endpoints that could be useful for penetration testers. Let's use Nikto to perform a basic web assessment of the three web servers we've identified on the IP addresses 172.16.10.10 (*p-web-01*), 172.16.10.11 (*p-ftp-01*), and 172.16.10.12 (*p-web-02*).

We'll run a Nikto scan against the web ports we found to be open on the three target IP addresses. Open a terminal and run the following commands one at a time so you can dissect the output for each IP address:

```
$ nikto -host 172.16.10.10 -port 8081
$ nikto -host 172.16.10.11 -port 80
$ nikto -host 172.16.10.12 -port 80
```

The output for 172.16.10.10 on port 8081 shouldn't yield much interesting information about discovered endpoints, but it should indicate that the server doesn't seem to be hardened, as it doesn't use security headers:

```
+ Server: Werkzeug/2.2.3 Python/3.11.1
+ The anti-clickjacking X-Frame-Options header is not present.
+ The X-XSS-Protection header is not defined. This header can hint to the user
agent to protect against some forms of XSS
+ The X-Content-Type-Options header is not set. This could allow the user
agent to render the content of the site in a different fashion to the MIME
type
--snip--
+ Allowed HTTP Methods: OPTIONS, GET, HEAD
+ 7891 requests: 0 error(s) and 4 item(s) reported on remote host
```

Nikto was able to perform a banner grab of the server, as indicated by the line that starts with the word `Server`. It then listed a few missing security headers. These are useful pieces of information but not enough to take over a server just yet.

The IP address 172.16.10.11 on port 80 should give you a similar result, though Nikto also discovered a new endpoint, */backup*, and that directory indexing mode is enabled:

```
+ Server: Apache/2.4.55 (Ubuntu)
--snip--
+ OSVDB-3268: /backup/: Directory indexing found.
+ OSVDB-3092: /backup/: This might be interesting...
```

Directory indexing is a server-side setting that, instead of a web page, lists files located at certain web paths. When enabled, the directory indexing setting lists the content of a directory when an index file is missing (such as *index.html* or *index.php*). Directory indexing is interesting to find because it could highlight sensitive files in an application, such as configuration files with connection strings, local database files (such as SQLite files), and other environmental files. Open the browser in Kali to *http://172.16.10.11/backup* to see the content of this endpoint (Figure 5-1).

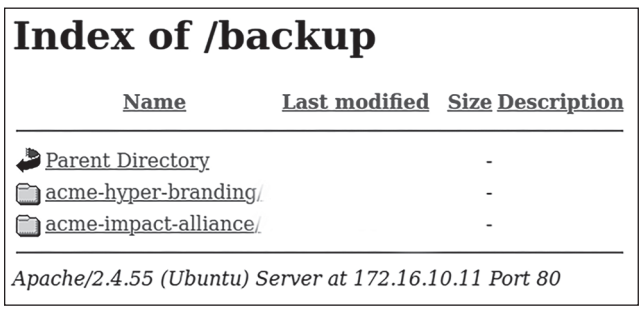

Figure 5-1: Directory indexing found on 172.16.10.11/backup

Directory indexing lets you view files in the browser. You can click directories to open them, click files to download them, and so on. On the web page, you should identify two folders: *acme-hyper-branding* and *acme-impact-alliance*. The *acme-hyper-branding* folder appears to contain a file named *app.py*. Download it to Kali by clicking it so it's available for later inspection.

We'll explore the third IP address in a moment, but first let's use bash automation to take advantage of directory indexing.

Building a Directory Indexing Scanner

What if we wanted to run a scan against a list of URLs to check whether they enable directory indexing, then download all the files they serve? In Listing 5-1, we use bash to carry out such a task.

directory_indexing_scanner.sh

```
#!/bin/bash
FILE="${1}"
OUTPUT_FOLDER="${2}"

❶ if [[ ! -s "${FILE}" ]]; then
    echo "You must provide a non-empty hosts file as an argument."
    exit 1
  fi

  if [[ -z "${OUTPUT_FOLDER}" ]]; then
  ❷ OUTPUT_FOLDER="data"
  fi

  while read -r line; do
  ❸ url=$(echo "${line}" | xargs)
```

```
  if [[ -n "${url}" ]]; then
    echo "Testing ${url} for Directory indexing..."
  ❹ if curl -L -s "${url}" | grep -q -e "Index of /" -e "[PARENTDIR]"; then
      echo -e "\t -!- Found Directory Indexing page at ${url}"
      echo -e "\t -!- Downloading to the \"${OUTPUT_FOLDER}\" folder..."
      mkdir -p "${OUTPUT_FOLDER}"
    ❺ wget -q -r -np -R "index.html*" "${url}" -P "${OUTPUT_FOLDER}"
    fi
  fi
done < <(cat "${FILE}")
```

Listing 5-1: Automatically downloading files available via directory indexing

In this script, we define the `FILE` and `OUTPUT_FOLDER` variables. Their assigned values are taken from the arguments the user passes on the command line (`$1` and `$2`). We then fail and exit the script (`exit 1`) if the `FILE` variable is not of the file type and of length zero (`-s`) ❶. If the file has a length of zero, it means the file is empty.

We then use a `while` loop to read the file at the path assigned to the `FILE` variable. At ❸, we ensure that each whitespace character in each line from the file is removed by piping it to the `xargs` command. At ❹, we use `curl` to make an HTTP GET request and follow any HTTP redirects (using `-L`). We silence verbose output from `curl` (using `-s`) and pipe it to `grep` to find any instances of the strings `Index of /` and `[PARENTDIR]`. These two strings exist in directory indexing pages. You can verify this by viewing the source HTML page at *http://172.16.10.11/backup*.

If we find either string, we call the `wget` command ❺ with the quiet option (`-q`) to silence verbose output, the recursive option (`-r`) to download files recursively from folders, the no-parent option (`-np`) to ensure we download only files at the same level of hierarchy or lower (subfolders), and the reject option (`-R`) to exclude files starting with *index.html*. We then use the target folder option (`-P`) to download the content to the path specified by the user calling the script (the `OUTPUT_FOLDER` variable). If the user didn't provide a destination folder, the script will default to using the *data* folder ❷.

NOTE *You can download this chapter's scripts from* https://github.com/dolevf/Black-Hat-Bash/blob/master/ch05.

The *acme-impact-alliance* folder we downloaded appears to be empty. But is it really? When dealing with web servers, you may run into what seem to be dead ends only to find out that something is hiding there, just not in an obvious place. Take note of the empty folder for now; we'll resume this exploration in a little bit.

Identifying Suspicious robots.txt Entries

After scanning the third IP address, 172.16.10.12 (*p-web-02*), Nikto outputs the following:

```
+ Server: Apache/2.4.54 (Debian)
+ Retrieved x-powered-by header: PHP/8.0.28
--snip--
+ Uncommon header 'link' found, with contents: <http://172.16.10.12/wp-json/>;
rel="https://api.w.org/"
--snip--
+ Entry '/wp-admin/' in robots.txt returned a non-forbidden or redirect HTTP
code (302)
+ Entry '/donate.php' in robots.txt returned a non-forbidden or redirect HTTP
code (200)
+ "robots.txt" contains 17 entries which should be manually viewed.
+ /wp-login.php: Wordpress login found
--snip--
```

Nikto was able to find a lot more information this time! It caught missing security headers (which is extremely common to see in the wild, unfortunately). Next, Nikto found that the server is running on Apache and Debian and that it is powered by PHP, a backend programming language commonly used in web applications.

It also found an uncommon link that points to *http://172.16.10.12/wp-json* and found two suspicious entries in the *robots.txt* file—namely, */wp-admin/* and */donate.php*. The *robots.txt* file is a special file used to indicate to web crawlers (such as Google's search engine) which endpoints to index and which to ignore. Nikto hints that the *robots.txt* file may have more entries than just these two and advises us to inspect it manually.

Finally, it also identified another endpoint at */wp-login.php*, which is a login page for WordPress, a blog platform. Navigate to the main page at *http://172.16.10.12/* to confirm you've identified a blog.

Finding these non-indexed endpoints is useful during a penetration test because you can add them to your list of possible targets to test. When you open this file, you should notice a list of paths:

```
User-agent:  *

Disallow: /cgi-bin/
Disallow: /z/j/
Disallow: /z/c/
Disallow: /stats/
--snip--
Disallow: /manual/*
Disallow: /phpmanual/
Disallow: /category/
Disallow: /donate.php
Disallow: /amount_to_donate.txt
```

We identified some of these endpoints earlier (such as */donate.php* and */wp-admin*), but others we didn't see when scanning with Nikto. In Exercise 5, you'll use bash to automate your exploration of them.

Exercise 5: Exploring Non-indexed Endpoints

Nikto scanning returned a list of non-indexed endpoints. In this exercise, you'll use bash to see whether they really exist on the server. Put together a script that will make an HTTP request to *robots.txt*, return the response, and iterate over each line, parsing the output to extract only the paths. Then the script should make an additional HTTP request to each path and check the status code it returns.

Listing 5-2 is an example script that can get you started. It relies on a useful `curl` feature you'll find handy in your bash scripts: built-in variables you can reference to extract particular values from HTTP requests and responses, such as the size of the request sent (`%{size_request}`) and the size of the headers returned in bytes (`%{size_header}`).

curl_fetch_robots_txt.sh

```
#!/bin/bash
TARGET_URL="http://172.16.10.12"
ROBOTS_FILE="robots.txt"

❶ while read -r line; do
  ❷ path=$(echo "${line}" | awk -F'Disallow: ' '{print $2}')
  ❸ if [[ -n "${path}" ]]; then
      url="${TARGET_URL}${path}"
      status_code=$(curl -s -o /dev/null -w "%{http_code}" "${url}")
      echo "URL: ${url} returned a status code of: ${status_code}"
    fi

❹ done < <(curl -s "${TARGET_URL}/${ROBOTS_FILE}")
```

Listing 5-2: Reading robots.txt *and making requests to individual paths*

At ❶, we read the output from the `curl` command at ❹ line by line. This command makes an HTTP GET request to *http://172.16.10.12/robots.txt*. We then parse each line and grab the second field (which is separated from the others by a space) to extract the path and assign it to the `path` variable ❷. We check that the `path` variable length is greater than zero to ensure we were able to properly parse it ❸.

Then we create a `url` variable, which is a string concatenated from the `TARGET_URL` variable plus each path from the *robots.txt* file, and make an HTTP request to the URL. We use the `-w` (write-out) variable `%{http_code}` to extract only the status code from the response returned by the web server.

To go beyond this script, try using other `curl` variables. You can find the full list of variables at *https://curl.se/docs/manpage.html* or by running the `man curl` command.

Brute-Forcing Directories with dirsearch

The *dirsearch* fast directory brute-forcing tool is used to find hidden paths and files on web servers. Written in Python by Mauro Soria, dirsearch provides features such as built-in web directory wordlists, bring-your-own-dictionary options, and advanced response filtering. We'll use it to try to

identify additional attack vectors and verify that Nikto hasn't missed anything obvious.

First, let's rescan port 8081 on *p-web-01* (172.16.10.10), which yielded no discovered endpoints when scanned by Nikto. The following dirsearch command uses the `-u` (URL) option to specify a base URL from which to start crawling:

```
$ dirsearch -u http://172.16.10.10:8081/

--snip--

Target: http://172.16.10.10:8081/

[00:14:55] Starting:
[00:15:32] 200 -  371B  - /upload
[00:15:35] 200 -   44B  - /uploads
```

Great! This tool was able to pick up two previously unknown endpoints named */upload* and */uploads*. This is why it's important to double- and triple-check your results by using more than one tool and to manually verify the findings; tools sometimes produce false positives or use limited path-list databases. If you navigate to the */upload* page, you should see a file-upload form. Take note of this endpoint because we'll test it in Chapter 6.

Let's also use dirsearch to look for attack vectors in what looked like an empty folder on *p-ftp-01*, at *http://172.16.10.11/backup/acme-impact-alliance*:

```
$ dirsearch -u http://172.16.10.11/backup/acme-impact-alliance/

--snip--
Extensions: php, aspx, jsp, html, js | HTTP method: GET | Threads: 30 | Wordlist size: 10927
Target: http://172.16.10.11/backup/acme-impact-alliance/
--snip--
[22:49:53] Starting:
[22:49:53] 301 -  337B  - /backup/acme-impact-alliance/js  ->  http://172.16.10.11/backup/
acme-impact-alliance/js/
[22:49:53] 301 -  339B  - /backup/acme-impact-alliance/.git  ->  http://172.16.10.11/backup/
acme-impact-alliance/.git/
--snip--
[22:49:53] 200 -   92B  - /backup/acme-impact-alliance/.git/config
--snip--
```

dirsearch inspects responses returned from the web server to identify interesting behaviors that could indicate the existence of an asset. For example, the tool might note whether a certain URL redirects to a new location (specified by an HTTP status code 301) and the response size in bytes. Sometimes you can infer information and observe behaviors solely by inspecting this data.

This time, we've identified a subfolder within the *acme-impact-alliance* folder named *.git*. A folder with this name usually indicates the existence of a Git repository on the server. *Git* is a source code management tool, and in this case, it likely manages code running locally on the remote server.

Use dirsearch again to perform brute forcing against the second directory, */backup/acme-hyper-branding*. Save the results into their own folder, then check them. You should find a Git repository there too.

Exploring Git Repositories

When you find a Git repository, it's often useful to run a specialized Git cloner that pulls the repository and all its associated metadata so you can inspect it locally. For this task, we'll use Gitjacker.

Cloning the Repository

Gitjacker's command is pretty simple. The first argument is a URL, and the `-o` (output) argument takes a folder name into which the data will be saved if Gitjacker succeeds at pulling the repository:

```
$ gitjacker http://172.16.10.11/backup/acme-impact-alliance/ -o acme-impact-alliance-git

--snip--
Target:     http://172.16.10.11/backup/acme-impact-alliance/
Output Dir: acme-impact-alliance-git
Operation complete.

Status:            Success
Retrieved Objects: 3242
--snip--
```

As you can see, the tool returned a successful status and a few thousand objects. At this point, you should have a folder named *acme-impact-alliance-git*:

```
$ ls -la ./acme-impact-alliance-git

--snip--
128 -rw-r--r--  1 kali kali 127309 Mar 17 23:15 comment.php
 96 -rw-r--r--  1 kali kali  96284 Mar 17 23:15 comment-template.php
 16 -rw-r--r--  1 kali kali  15006 Mar 17 23:15 compat.php
  4 drwxr-xr-x  2 kali kali   4096 Mar 17 23:15 customize
--snip--
 12 -rw-r--r--  1 kali kali  10707 Mar 17 23:15 customize.php
  4 -rw-r--r--  1 kali kali    705 Mar 17 23:15 donate.php
  4 -rw-r--r--  1 kali kali    355 Mar 17 23:15 robots.txt
--snip--
```

Notice some familiar filenames in this list? We saw *donate.php* and *robots.txt* earlier, when we scanned the 172.16.10.12 (*p-web-02*) host.

Viewing Commits with git log

When you run into a Git repository, you should attempt a `git log` command to see the history of Git code commits made to the repository, as they may

include interesting data we could use as attackers. In source code management, a *commit* is a snapshot of the code's state that is taken before the code is pushed to the main repository and made permanent. Commit information could include details about who made the commit and a description of the change (such as whether it was a code addition or deletion):

```
$ cd acme-impact-alliance-git
$ git log

commit 3822fd7a063f3890e78051e56bd280f00cc4180c (HEAD -> master)
Author: Kevin Peterson <kpeterson@acme-impact-alliance.com>
--snip--

    commit code
```

We've identified a person who has committed code to the Git repository: Kevin Peterson, at *kpeterson@acme-impact-alliance.com*. Take note of this information because this account could exist in other places found during the penetration test.

Try running Gitjacker again to hijack the Git repository that lives on the second folder, at */backup/acme-hyper-branding*. Then execute another **`git log`** command to see who committed code to this repository, as we did before. The log should reveal the identity of a second person: Melissa Rogers, at *mrogers@acme-hyper-branding.com*.

You may sometimes run into Git repositories with many contributors and many commits. We can use Git's built-in `--pretty=format` option to easily extract all this metadata, like so:

```
$ git log --pretty=format:"%an %ae"
```

The `%ae` (author name) and `%ae` (email) fields are built-in placeholders in Git that allow you to specify values of interest to include in the output. For the list of all available variables, see *https://git-scm.com/docs/pretty-formats#_pretty_formats*.

Filtering git log Information

Even without the pretty formatting, bash can filter `git log` output with a single line:

```
$ git log | grep Author | grep -oP '(?<=Author: ).*' | sort -u | tr -d '<>'
```

This bash code runs `git log`, uses `grep` to search for any lines that start with the word `Author`, and then pipes the results to another `grep` command, which uses regular expressions (`-oP`) to filter anything after the word `Author:` and print only the words that matched. This filtering leaves us with the Git commit author's name and email.

Because the same author could have made multiple commits, we use `sort` to sort the list and use the `-u` option to remove any duplicated lines,

leaving us with a list free of duplicated entries. Finally, since the email is surrounded by the characters <> by default, we trim these characters by using `tr -d '<>'`.

Inspecting Repository Files

The repository contains a file called *app.py*. Let's inspect its contents by viewing it in a text editor. You should see that the file contains web server code written with Python's Flask library:

```
import os, subprocess

from flask import (
    Flask,
    send_from_directory,
    send_file,
    render_template,
    request
)



@app.route('/')

--snip--

@app.route('/files/<path:path>')

--snip--

@app.route('/upload', methods = ['GET', 'POST'])

--snip--

@app.route('/uploads', methods=['GET'])

--snip--

@app.route('/uploads/<path:file_name>', methods=['GET'])

--snip--
```

The interesting parts here are the endpoints that are exposed via `@app.route()`. You can see that the application exposes endpoints such as */*, */files*, */upload*, and */uploads*.

When we scanned the target IP address range with dirsearch and Nikto, we saw two endpoints, named */upload* and */uploads*, on *p-web-01* (172.16.10.10:8081). Because this Python file includes the same endpoints, this source code likely belongs to the application running on the server.

You may be asking yourself why we didn't find the */files* endpoint in our scans. Well, web scanners often rely on response status codes returned by

web servers to determine whether certain endpoints exist. If you run the following `curl` command with the `-I` (HEAD request) option, you'll see that the */files* endpoint returns the HTTP status code 404 Not Found:

```
$ curl -I http://172.16.10.10:8081/files

HTTP/1.1 404 NOT FOUND
--snip--
```

Web scanners interpret these 404 errors as indicating that an endpoint doesn't exist. Yet the reason we get 404 errors here is that, when called directly, */files* doesn't serve any requests. Instead, it serves requests for web paths appended to */files*, such as */files/abc.jpg* or */files/salary.docx*.

Vulnerability Scanning with Nuclei

Nuclei is one of the most impressive open source vulnerability scanners released in recent years. Its advantage over other tools stems from its community-powered templating system, which reduces false positives by matching known patterns against responses it receives from network services and files. It also reduces barriers to writing vulnerability checks, as it doesn't require learning how to code. You can also easily extend it to do custom security checks.

Nuclei naturally supports common network services, such as HTTP, DNS, and network sockets, as well as local file scanning. You can use it to send HTTP requests, DNS queries, and raw bytes over the network. Nuclei can even scan files to find credentials (for example, when you've identified an open Git repository and want to pull it locally to find secrets).

As of this writing, Nuclei has more than 8,000 templates in its database. In this section, we'll introduce Nuclei and how to use it.

Understanding Templates

Nuclei templates are based on YAML files with the following high-level structure:

ID A unique identifier for the template

Metadata Information about the template, such as a description, the author, the severity, and tags (arbitrary labels that can group multiple templates, such as *injection* or *denial of service*)

Protocol The mechanism that the template uses to make its requests; for example, `http` is a protocol type that uses HTTP for web requests

Operators Used for matching patterns against responses received by a template execution (*matchers*) and extracting data (*extractors*), similarly to the filtering performed by tools like `grep`

Here is a simple example of a Nuclei template that uses HTTP to find the default Apache HTML welcome page. Navigate to *http://172.16.10.11/* to see what this page looks like.

```
id: detect-apache-welcome-page

❶ info:
    name: Apache2 Ubuntu Default Page
    author: Dolev Farhi and Nick Aleks
    severity: info
    tags: apache

  http:
    - method: GET
      path:
      ❷ - '{{BaseURL}}'
    ❸ matchers:
        - type: word
          words:
            - "Apache2 Ubuntu Default Page: It works"
          part: body
```

We define the template metadata, such as the template's name, author, severity, and so on ❶. We then instruct Nuclei to use an HTTP client when executing this template ❷. We also declare that the template should use the GET method. Next, we define a variable that will be swapped with the target URL we'll provide to Nuclei on the command line at scan time. Then, we define a single matcher of type `word` ❸ and a search pattern to match against the HTTP response body coming back from the server, defined by `part: body`.

As a result, when Nuclei performs a scan against an IP address that runs some form of a web server, this template will make a GET request to its base URL (`/`) and look for the string `Apache2 ubuntu Default Page: It works` in the response. If it finds this string in the response's body, the check will be considered successful because the pattern matched.

We encourage you to explore Nuclei's templating system at *https://docs.projectdiscovery.io/introduction*, as you can easily use Nuclei with bash to perform continuous assessments.

Writing a Custom Template

Let's write a simple template that finds the Git repositories we discovered earlier, on *p-ftp-01* (172.16.10.11). We'll define multiple `BaseURL` paths to represent the two paths we've identified. Then, using Nuclei's matchers, we'll define a string `ref: refs/heads/master` to match the response body returned by the scanned server:

git-finder.yaml

```
id: detect-git-repository

info:
  name: Git Repository Finder
  author: Dolev Farhi and Nick Aleks
  severity: info
  tags: git
```

```
http:
  - method: GET
    path:
      - '{{BaseURL}}/backup/acme-hyper-branding/.git/HEAD'
      - '{{BaseURL}}/backup/acme-impact-alliance/.git/HEAD'
    matchers:
      - type: word
        words:
          - "ref: refs/heads/master"
        part: body
```

This template works just like the one in the previous example, except this time we provide two paths to check against: */backup/acme-hyper-branding/.git/HEAD* and */backup/acme-impact-alliance/.git/HEAD*. The matcher defines the string we expect to see in the *HEAD* file. You can confirm the match by making a `curl` request to the Git repository at 172.16.10.11:

```
$ curl http://172.16.10.11/backup/acme-hyper-branding/.git/HEAD

ref: refs/heads/master
```

Download this custom Nuclei template from the book's GitHub repository.

Applying the Template

Let's run Nuclei against *p-ftp-01* (172.16.10.11) with the custom template we just wrote. Nuclei stores its built-in templates in the folder *~/.local/nuclei-templates*. First, run the following command to update Nuclei's template database:

```
$ nuclei -ut
```

Next, save the custom template into the folder *~/.local/nuclei-templates/custom* and give it a name such as *git-finder.yaml*.

In the following command, the `-u` (URL) option specifies the address, and `-t` (template) specifies the path to the template:

```
$ nuclei -u 172.16.10.11 -t ~/.local/nuclei-templates/custom/git-finder.yaml

--snip--
[INF] Targets loaded for scan: 1
[INF] Running httpx on input host
[INF] Found 1 URL from httpx
[detect-git-repository] [http] [info] http://172.16.10.11/backup/acme-hyper-branding/.git/HEAD
[detect-git-repository] [http] [info] http://172.16.10.11/backup/acme-impact-alliance/.git/HEAD
```

As you can see, we were able to identify the two Git repositories with the custom template.

Running a Full Scan

When not provided with a specific template, Nuclei will use its built-in templates during the scan. Running Nuclei is noisy, so we recommend tailoring

the execution to a specific target. For instance, if you know a server is running Apache, you could select just the Apache-related templates by specifying the `-tags` option:

```
$ nuclei -tags apache,git -u 172.16.10.11
```

Run **`nuclei -tl`** to get a list of all available templates.

Let's run a full Nuclei scan against the three IP addresses in the 172.16.10.0/24 network by using all its built-in templates:

```
$ nuclei -u 172.16.10.10:8081
$ nuclei -u 172.16.10.11
$ nuclei -u 172.16.10.12

--snip--
[tech-detect:google-font-api] [http] [info] http://172.16.10.10:8081
[tech-detect:python] [http] [info] http://172.16.10.10:8081
[http-missing-security-headers:access-control-allow-origin] [http] [info]
http://172.16.10.10:8081
[http-missing-security-headers:content-security-policy] [http] [info]
http://172.16.10.10:8081
--snip--
```

Nuclei tries to optimize the number of total requests made by using *clustering*. When multiple templates call the same web path (such as */backup*), Nuclei consolidates these into a single request to reduce network overhead. However, Nuclei could still send thousands of requests during a single scan. You can control the number of requests sent by specifying the rate limit option (`-rl`), followed by an integer indicating the number of allowed requests per second.

The full scan results in a lot of findings, so append the output to a file (using `>>`) so that you can examine them one by one. As you'll see, Nuclei can identify vulnerabilities, but it can also fingerprint the target server and the technologies running on it. Nuclei should have highlighted findings seen previously, as well as a few new ones. Here are some of the issues it detected:

- An FTP server with anonymous access enabled on 172.16.10.11 port 21
- A WordPress login page at *172.16.10.12/wp-login.php*
- A WordPress user-enumeration vulnerability (CVE-2017-5487) at *http://172.16.10.12/?rest_route=/wp/v2/users/*

Let's manually confirm these three findings to ensure there are no false positives. Connect to the identified FTP server at 172.16.10.11 by issuing the following `ftp` command. This command will connect to the server by using the *anonymous* user and an empty password:

```
$ ftp ftp://anonymous:@172.16.10.11

Connected to 172.16.10.11.
220 (vsFTPd 3.0.5)
```

```
331 Please specify the password.
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
200 Switching to Binary mode.
```

We were able to connect! Let's issue an `ls` command to verify that we can list files and directories on the server:

```
ftp> ls
229 Entering Extended Passive Mode (|||33817|)
150 Here comes the directory listing.
drwxr-xr-x    1 0        0            4096 Mar 11 05:23 backup
-rw-r--r--    1 0        0           10671 Mar 11 05:22 index.html
226 Directory send OK.
```

We see an *index.html* file and a *backup* folder. This is the same folder that stores the two Git repositories we saw earlier, except now we have access to the FTP server where these files actually live.

Next, open a browser to *http://172.16.10.12/wp-login.php* from your Kali machine. You should see the page in Figure 5-2.

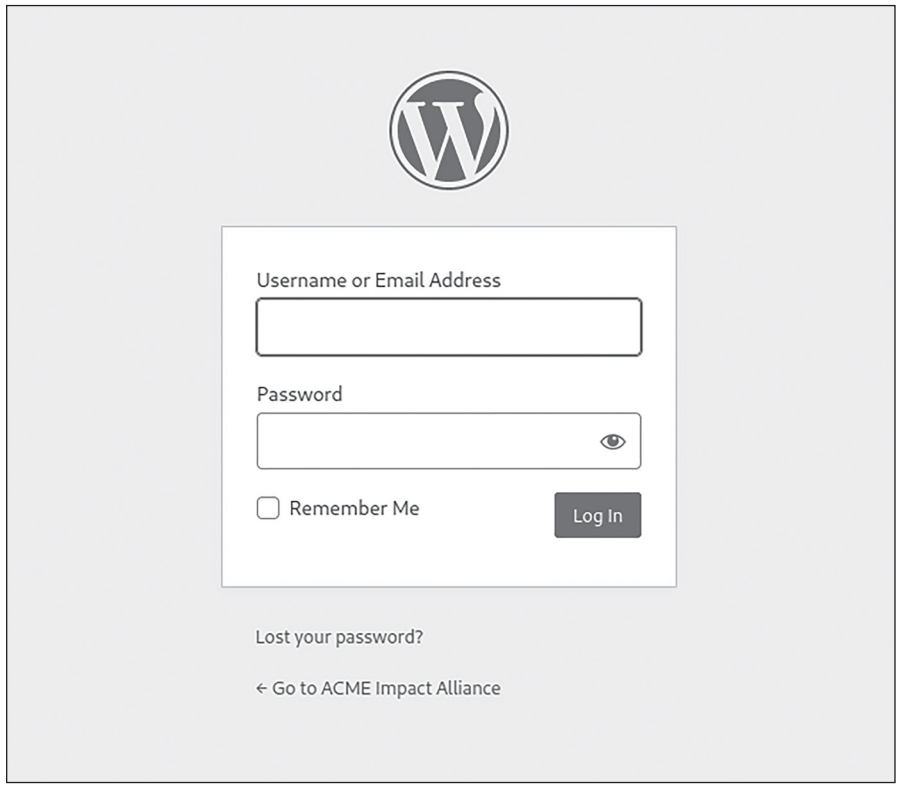

Figure 5-2: The WordPress login page

Finally, verify the third finding: the WordPress user-enumeration vulnerability, which allows you to gather information about WordPress accounts. By default, every WordPress instance exposes an API endpoint that lists WordPress system users. The endpoint usually doesn't require authentication or authorization, so a simple GET request should return the list of users.

We'll use `curl` to send this request and then pipe the response to jq to prettify the JSON output that comes back. The result should be an array of user data:

```
$ curl -s http://172.16.10.12/?rest_route=/wp/v2/users | jq

[
  {
    "id": 1,
    "name": "jtorres",
    "url": "http://172.16.10.12",
    "description": "",
    "link": "http://172.16.10.12/author/jtorres/",
    "slug": "jtorres",
  },
--snip--
]
```

The blog has a single user, *jtorres*. This can be a good target to brute-force later. If this `curl` command had returned many users, you could have parsed only the usernames with jq (Listing 5-3).

```
$ curl -s http://172.16.10.12/?rest_route=/wp/v2/users/ | jq .[].name
```

Listing 5-3: Extracting usernames from an HTTP response

All three findings were true positives, which is great news for us. Table 5-1 recaps the users we've identified so far.

Table 5-1: Identity Information Gathered from Repositories and WordPress

Source	Name	Email
acme-impact-alliance Git repository	Kevin Peterson	*kpeterson@acme-impact-alliance.com*
acme-hyper-branding Git repository	Melissa Rogers	*mrogers@acme-hyper-branding.com*
WordPress account	J. Torres	*jtorres@acme-impact-alliance.com*

Because the *jtorres* account was found on the ACME Impact Alliance website and we already know the email scheme the website uses, it's pretty safe to assume that the *jtorres* email is *jtorres@acme-impact-alliance.com*.

Exercise 6: Parsing Nuclei's Findings

Nuclei's scan output is a little noisy and can be difficult to parse with bash, but not impossible. Nuclei allows you to pass a `-silent` parameter to show only the findings in the output. Before you write a script to parse it, consider Nuclei's output format:

[*template*] [*protocol*] [*severity*] *url* [*extractor*]

Each field is enclosed in square brackets `[]` and separated by spaces. The *template* field is a template name (taken from the name of the template file); the *protocol* shows the protocol, such as HTTP; and the *severity* shows the severity of the finding (informational, low, medium, high, or critical). The fourth field is the URL or IP address, and the fifth field is metadata extracted by the template's logic using extractors.

Now you should be able to parse this information with bash. Listing 5-4 shows an example script that runs Nuclei, filters for a specific severity of interest, parses the interesting parts, and emails you the results.

nuclei-notifier.sh

```
#!/bin/bash
EMAIL_TO="security@blackhatbash.com"
EMAIL_FROM="nuclei-automation@blackhatbash.com"

for ip_address in "$@"; do
  echo "Testing ${ip_address} with Nuclei..."
❶ result=$(nuclei -u "${ip_address}" -silent -severity medium,high,critical)
  if [[ -n "${result}" ]]; then
  ❷ while read -r line; do
      template=$(echo "${line}" | awk '{print $1}' | tr -d '[]')
      url=$(echo "${line}" | awk '{print $4}')
      echo "Sending an email with the findings ${template} ${url}"
      sendemail -f "${EMAIL_FROM}" \
              ❸ -t "${EMAIL_TO}" \
                -u "[Nuclei] Vulnerability Found!" \
                -m "${template} - ${url}"

  ❹ done <<< "${result}"
  fi
done
```

Listing 5-4: Scanning with Nuclei and sending yourself the results

Let's dissect the code to better understand what it's doing. We use a `for` loop to iterate through values in the `$@` variable, a special value you learned about in Chapter 1 that contains the arguments passed to the script on the command line. We assign each argument to the `ip_address` variable.

Next, we run a Nuclei scan, passing it the `-severity` argument to scan for vulnerabilities categorized as either medium, high, or critical, and save the output to the `result` variable ❶. At ❷, we read the output passed to the `while` loop at ❹ line by line. From each line, we extract the first field, using the `tr -d '[]'` command to remove the `[]` characters for a cleaner output. We also extract the fourth field from each line, which is where Nuclei

stores the vulnerable URL. At ❸, we send an email containing the relevant information.

To run this script, save it to a file and pass the IP addresses to scan on the command line:

```
$ nuclei-notifier.sh 172.16.10.10:8081 172.16.10.11 172.16.10.12 172.16.10.13
```

To make this script your own, try having Nuclei output JSON data by using the `-j` option. Then pipe this output to jq, as shown in Chapter 4.

Fuzzing for Hidden Files

Now that we've identified the potential location of files, let's use fuzzing tools to find hidden files on *p-web-01* (*http://172.16.10.10:8081/files*). *Fuzzers* generate semi-random data to use as part of a payload. When sent to an application, these payloads can trigger anomalous behavior or reveal covert information. You can use fuzzers against web servers to find hidden paths or against local binaries to find vulnerabilities such as buffer overflows or DoS.

Creating a Wordlist of Possible Filenames

Fuzzing tools in the context of web application enumeration work best when fed custom wordlists tailored to your target. These lists could contain the name of the company, the individuals you've identified, relevant locations, and so on. These tailored wordlists can help you identify user accounts to attack, network and application services, valid domain names, covert files, email addresses, and web paths, for example.

Let's use bash to write a custom wordlist containing potential filenames of interest (Listing 5-5).

```
$ echo -e acme-hyper-branding-{0..100}.{txt,csv,pdf,jpg}"\n" | sed 's/ //g' > files_wordlist.txt
```

Listing 5-5: Using brace expansion to create multiple files with various extensions

This command creates files with probable file extensions tailored to our target's name, ACME Hyper Branding. It uses `echo` with brace expansion `{0..100}` to create arbitrary strings ranging from 0 to 100 and then appends these to the company name. We also use brace expansion to create multiple file extension types, such as *.txt*, *.csv*, *.pdf*, and *.jpg*. The `-e` option, for `echo`, enables us to interpret backslash (\) escapes. This means that `\n` will be interpreted as a newline. We then pipe this output to the `sed` command to remove all whitespace from the output for a cleaner list.

Use `head` to view the created files:

```
$ head files_wordlist.txt

acme-hyper-branding-0.txt
acme-hyper-branding-0.csv
```

```
acme-hyper-branding-0.pdf
acme-hyper-branding-0.jpg
acme-hyper-branding-1.txt
acme-hyper-branding-1.csv
acme-hyper-branding-1.pdf
acme-hyper-branding-1.jpg
acme-hyper-branding-2.txt
acme-hyper-branding-2.csv
```

As you can see, this command's output follows the format *acme-hyper-branding-<some_number>.<some_extension>*.

Fuzzing with ffuf

ffuf (an acronym for *Fuzz Faster U Fool*) is a versatile and blazing-fast web fuzzing tool. We'll use ffuf to discover potential files under the */files* endpoint that could contain interesting data.

The following `ffuf` command uses the `-c` (color) option to highlight the results in the terminal, the `-w` (wordlist) option to specify a custom wordlist, the `-u` (URL) option to specify a path, and the full URL to the endpoint to fuzz. We run ffuf against *p-web-01* (172.16.10.10):

```
$ ffuf -c -w files_wordlist.txt -u http://172.16.10.10:8081/files/FUZZ

:: Method           : GET
:: URL              : http://172.16.10.10:8081/files/FUZZ
:: Wordlist         : FUZZ: files_wordlist.txt
:: Follow redirects : false
:: Calibration      : false
:: Timeout          : 10
:: Threads          : 40
:: Matcher          : Response status: 200,204,301,302,307,401,403,405,500

________________________________________________

acme-hyper-branding-5.csv [Status: 200, Size: 432, Words: 31, Lines: 9, Duration: 32ms]
:: Progress: [405/405] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] :: Errors: 0 ::
```

Note that the word `FUZZ` at the end of the URL is a placeholder that tells the tool where to inject the words from the wordlist. In essence, it will swap the word `FUZZ` with each line from our file.

According to the output, ffuf identified that the path *http://172.16.10.10:8081/files/acme-hyper-branding-5.csv* returned a status code of HTTP 200 OK. If you look closely at the output, you should see that the fuzzer sent 405 requests in less than a second, which is pretty impressive.

Fuzzing with Wfuzz

Wfuzz is another web fuzzing tool similar to ffuf. In fact, ffuf is based on Wfuzz. Let's use Wfuzz to perform the same type of wordlist-based scan

(`-w`) and then use its filtering capabilities to show only files that receive a response status code of 200 OK (`--sc 200`):

```
$ wfuzz --sc 200 -w files_wordlist.txt http://172.16.10.10:8081/files/FUZZ

--snip--
Target: http://172.16.10.10:8081/files/FUZZ
Total requests: 405

=====================================================================
ID          Response   Lines    Word       Chars       Payload
=====================================================================

000000022: 200        8 L      37 W       432 Ch      "acme-hyper-branding-5.csv"

Total time: 0
Processed Requests: 405
Filtered Requests: 404
Requests/sec.: 0
```

Next, let's use the `wget` command to download the identified file:

```
$ wget http://172.16.10.10:8081/files/acme-hyper-branding-5.csv
$ cat acme-hyper-branding-5.csv

no, first_name, last_name, designation, email
1, Jacob, Taylor, Founder, jtayoler@acme-hyper-branding.com
2, Sarah, Lewis, Executive Assistance, slewis@acme-hyper-branding.com
3, Nicholas, Young, Influencer, nyoung@acme-hyper-branding.com
4, Lauren, Scott, Influencer, lscott@acme-hyper-branding.com
5, Aaron,Peres, Marketing Lead, aperes@acme-hyper-branding.com
6, Melissa, Rogers, Marketing Lead, mrogers@acme-hyper-branding.com
```

We've located a table of PII, including first and last names, titles, and email addresses. Take notes of every detail we've managed to extract in this chapter; you never know when it will come in handy.

Note that fuzzers can cause unintentional DoS conditions, especially if they're optimized for speed. You may encounter applications running on low-powered servers that will crash if you run a highly capable fuzzer against them, so make sure you have explicit permission from the company you're working with to perform such activities.

Assessing SSH Servers with Nmap's Scripting Engine

Nmap contains many NSE scripts to test for vulnerabilities and misconfigurations. All Nmap scripts live in the */usr/share/nmap/scripts* path. When you run Nmap with the `-A` flag, it will blast all NSE scripts at the target, as well as enable operating system detection, version detection, script scanning, and traceroute. This is probably the noisiest scan you can do with Nmap, so never use it when you need to be covert.

In Chapter 4, we identified a server running OpenSSH on *p-jumpbox-01* (172.16.10.13). Let's use an NSE script tailored to SSH servers to see what we can discover about the supported authentication methods:

```
$ nmap --script=ssh-auth-methods 172.16.10.13

Starting Nmap ( https://nmap.org ) at 03-19 01:53 EDT
--snip--
PORT   STATE SERVICE
22/tcp open  ssh
| ssh-auth-methods:
|   Supported authentication methods:
|     publickey
|_    password

Nmap done: 1 IP address (1 host up) scanned in 0.26 seconds
```

The *ssh-auth-methods* NSE script enumerates the authentication methods offered by the SSH server. If *password* is one of them, this means that the server accepts passwords as an authentication mechanism. SSH servers that allow password authentication are prone to brute-force attacks. In Chapter 7, we'll perform a brute-force attack against SSH servers.

Exercise 7: Combining Tools to Find FTP Issues

The goal of this exercise is to write a script that calls several security tools, parses their output, and passes the output to other tools to act on it. Orchestrating multiple tools in this way is a common task in penetration testing, so we encourage you to get comfortable with building such workflows.

Your script should do the following:

1. Accept one or more IP addresses on the command line.
2. Run a port scanner against the IP addresses; which port scanner you use is completely up to you.
3. Identify open ports. If any of them are FTP ports (21/TCP), the script should pass the address to the vulnerability scanner in step 4.
4. Use Nuclei to scan the IP addresses and ports. Try applying templates dedicated to finding issues in FTP servers. Search the Nuclei templates folder */home/kali/.local/nuclei-templates* for FTP-related templates, or use the `-tags ftp` Nuclei flag.
5. Scan the IP addresses with Nmap. Use NSE scripts that find vulnerabilities in FTP servers, which you can search for in the */usr/share/nmap/scripts* folder. For example, try *ftp-anon.nse*.
6. Parse and write the results to a file, in a format of your choice. The file should include a description of the vulnerability, the relevant IP address and port, the timestamp at which it was found, and the name of the tool that detected the issue. There is no hard requirement about

how to present the data; one option is to use an HTML table. If you need an example table, download *vulnerability_table.html* from the book's GitHub repository and open it in a browser. Alternatively, you could write the results to a CSV file.

As you should know by now, there is more than one way to write such a script. Only the end result matters, so craft the script as you see fit.

Summary

In this chapter, we wrapped up reconnaissance activities by performing vulnerability scanning and fuzzing. We also verified the vulnerabilities we discovered, weeding out potential false positives.

Along the way, we used bash scripting to perform several tasks. We scanned for vulnerabilities, wrote custom scripts that can perform recursive downloads from misconfigured web servers, extracted sensitive information from Git repositories, and more. We also created custom wordlists using clever bash scripting and orchestrated the execution of multiple security tools to generate a report.

Let's recap what we've identified so far, from a reconnaissance perspective:

- Hosts running multiple services (HTTP, FTP, and SSH) and their versions
- A web server running WordPress with a login page enabled and a few vulnerabilities, such as user enumeration and an absence of HTTP security headers
- A web server with a revealing *robots.txt* file containing paths to custom upload forms and a donation page
- An anonymous, login-enabled FTP server
- Multiple open Git repositories
- OpenSSH servers that allow password-based logins

In the next chapter, we'll use the information identified in this chapter to establish an initial foothold by exploiting vulnerabilities and taking over servers.