Retire.js is a tool that detects outdated JavaScript libraries and Node.js packages. You can use it to check for outdated technologies on a site.

Writing Your Own Recon Scripts

You've probably realized by now that good recon is an extensive process. But it doesn't have to be time-consuming or hard to manage. We've already discussed several tools that use the power of automation to make the process easier.

Sometimes you may find it handy to write your own scripts. A *script* is a list of commands designed to be executed by a program. They're used to automate tasks such as data analysis, web-page generation, and system administration. For us bug bounty hunters, scripting is a way of quickly and efficiently performing recon, testing, and exploitation. For example, you could write a script to scan a target for new subdomains, or enumerate potentially sensitive files and directories on a server. Once you've learned how to script, the possibilities are endless.

This section covers bash scripts in particular—what they are and why you should use them. You'll learn how to use bash to simplify your recon process and even write your own tools. I'll assume that you have basic knowledge of how programming languages work, including variables, conditionals, loops, and functions, so if you're not familiar with these concepts, please take an introduction to coding class at Codecademy (*https://www.codecademy.com/*) or read a programming book.

Bash scripts, or any type of shell script, are useful for managing complexities and automating recurrent tasks. If your commands involve multiple input parameters, or if the input of one command depends on the output of another, entering it all manually could get complicated quickly and increase the chance of a programming mistake. On the other hand, you might have a list of commands that you want to execute many, many times. Scripts are useful here, as they save you the trouble of typing the same commands over and over again. Just run the script each time and be done with it.

Understanding Bash Scripting Basics

Let's write our first script. Open any text editor to follow along. The first line of every shell script you write should be the *shebang line*. It starts with a hash mark (`#`) and an exclamation mark (`!`), and it declares the interpreter to use for the script. This allows the plaintext file to be executed like a binary. We'll use it to indicate that we're using bash.

Let's say we want to write a script that executes two commands; it should run Nmap and then Dirsearch on a target. We can put the commands in the script like this:

```
#!/bin/bash
nmap scanme.nmap.org
/PATH/TO/dirsearch.py -u scanme.nmap.org -e php
```

This script isn't very useful; it can scan only one site, *scanme.nmap.org*. Instead, we should let users provide input arguments to the bash script so they can choose the site to scan. In bash syntax, `$1` represents the first argument passed in, `$2` is the second argument, and so on. Also, `$@` represents all arguments passed in, while `$#` represents the total number of arguments. Let's allow users to specify their targets with the first input argument, assigned to the variable `$1`:

```
#!/bin/bash
nmap $1
/PATH/TO/dirsearch.py -u $1 -e php
```

Now the commands will execute for whatever domain the user passes in as the first argument.

Notice that the third line of the script includes */PATH/TO/dirsearch.py*. You should replace */PATH/TO/* with the absolute path of the directory where you stored the Dirsearch script. If you don't specify its location, your computer will try to look for it in the current directory, and unless you stored the Dirsearch file in the same directory as your shell script, bash won't find it.

Another way of making sure that your script can find the commands to use is through the `PATH` variable, an environmental variable in Unix systems that specifies where executable binaries are found. If you run this command to add Dirsearch's directory to your `PATH`, you can run the tool from anywhere without needing to specify its absolute path:

```
export PATH="PATH_TO_DIRSEARCH:$PATH"
```

After executing this command, you should be able to use Dirsearch directly:

```
#!/bin/bash
nmap $1
dirsearch.py -u $1 -e php
```

Note that you will have to run the `export` command again after you restart your terminal for your `PATH` to contain the path to Dirsearch. If you don't want to export `PATH` over and over again, you can add the `export` command to your *~/.bash_profile* file, a file that stores your bash preferences and configuration. You can do this by opening *~/.bash_profile* with your favorite text editor and adding the `export` command to the bottom of the file.

The script is complete! Save it in your current directory with the filename *recon.sh*. The *.sh* extension is the conventional extension for shell scripts. Make sure your terminal's working directory is the same as the one where you've stored your script by running the command **`cd /location/of/your/script`**. Execute the script in the terminal with this command:

```
$ ./recon.sh
```

You might see a message like this:

```
permission denied: ./recon.sh
```

This is because the current user doesn't have permission to execute the script. For security purposes, most files aren't executable by default. You can correct this behavior by adding executing rights for everyone by running this command in the terminal:

```
$ chmod +x recon.sh
```

The `chmod` command edits the permissions for a file, and `+x` indicates that we want to add the permission to execute for all users. If you'd like to grant executing rights for the owner of the script only, use this command instead:

```
$ chmod 700 recon.sh
```

Now run the script as we did before. Try passing in *scanme.nmap.org* as the first argument. You should see the output of the Nmap and Dirsearch printed out:

```
$ ./recon.sh scanme.nmap.org
Starting Nmap 7.60 ( https://nmap.org )
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.062s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2f
Not shown: 992 closed ports
PORT      STATE    SERVICE
22/tcp    open     ssh
25/tcp    filtered smtp
80/tcp    open     http
135/tcp   filtered msrpc
139/tcp   filtered netbios-ssn
445/tcp   filtered microsoft-ds
9929/tcp  open     nping-echo
31337/tcp open     Elite
Nmap done: 1 IP address (1 host up) scanned in 2.16 seconds

Extensions: php | HTTP method: get | Threads: 10 | Wordlist size: 6023
Error Log: /Users/vickieli/tools/dirsearch/logs/errors.log
Target: scanme.nmap.org
[11:14:30] Starting:
[11:14:32] 403 -  295B  - /.htaccessOLD2
[11:14:32] 403 -  294B  - /.htaccessOLD
[11:14:33] 301 -  316B  - /.svn  ->  http://scanme.nmap.org/.svn/
[11:14:33] 403 -  298B  - /.svn/all-wcprops
[11:14:33] 403 -  294B  - /.svn/entries
[11:14:33] 403 -  297B  - /.svn/prop-base/
[11:14:33] 403 -  296B  - /.svn/pristine/
[11:14:33] 403 -  315B  - /.svn/text-base/index.php.svn-base
[11:14:33] 403 -  297B  - /.svn/text-base/
[11:14:33] 403 -  293B  - /.svn/props/
[11:14:33] 403 -  291B  - /.svn/tmp/
[11:14:55] 301 -  318B  - /images  ->  http://scanme.nmap.org/images/
[11:14:56] 200 -    7KB - /index
[11:14:56] 200 -    7KB - /index.html
```

```
[11:15:08] 403 -  296B  - /server-status/
[11:15:08] 403 -  295B  - /server-status
[11:15:08] 301 -  318B  - /shared  ->  http://scanme.nmap.org/shared/
Task Completed
```

Saving Tool Output to a File

To analyze the recon results later, you may want to save your scripts' output in a separate file. This is where input and output redirection come into play. *Input redirection* is using the content of a file, or the output of another program, as the input to your script. *Output redirection* is redirecting the output of a program to another location, such as to a file or another program. Here are some of the most useful redirection operators:

PROGRAM **>** *FILENAME* Writes the program's output into the file with that name. (It will clear any content from the file first. It will also create the file if the file does not already exist.)

PROGRAM **>>** *FILENAME* Appends the output of the program to the end of the file, without clearing the file's original content.

PROGRAM **<** *FILENAME* Reads from the file and uses its content as the program input.

PROGRAM1 **|** *PROGRAM2* Uses the output of *PROGRAM1* as the input to *PROGRAM2*.

We could, for example, write the results of the Nmap and Dirsearch scans into different files:

```
#!/bin/bash
echo "Creating directory $1_recon." ❶
mkdir $1_recon ❷
nmap $1 > $1_recon/nmap ❸
echo "The results of nmap scan are stored in $1_recon/nmap."
/PATH/TO/dirsearch.py -u $1 -e php ❹ --simple-report=$1_recon/dirsearch
echo "The results of dirsearch scan are stored in $1_recon/dirsearch."
```

The `echo` command ❶ prints a message to the terminal. Next, `mkdir` creates a directory with the name *DOMAIN_recon* ❷. We store the results of `nmap` into a file named *nmap* in the newly created directory ❸. Dirsearch's `simple-report` flag ❹ generates a report in the designated location. We store the results of Dirsearch to a file named *dirsearch* in the new directory.

You can make your script more manageable by introducing variables to reference files, names, and values. Variables in bash can be assigned using the following syntax: *VARIABLE_NAME=VARIABLE_VALUE*. Note that there should be no spaces around the equal sign. The syntax for referencing variables is *$VARIABLE_NAME*. Let's implement these into the script:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon ❶
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
```

```
nmap $DOMAIN > $DIRECTORY/nmap
echo "The results of nmap scan are stored in $DIRECTORY/nmap."
$PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php –simple-report=$DIRECTORY/dirsearch ❷
echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
```

We use `${DOMAIN}_recon` instead of `$DOMAIN_recon` ❶ because, otherwise, bash would recognize the entirety of `DOMAIN_recon` as the variable name. The curly brackets tell bash that `DOMAIN` is the variable name, and `_recon` is the plaintext we're appending to it. Notice that we also stored the path to Dirsearch in a variable to make it easy to change in the future ❷.

Using redirection, you can now write shell scripts that run many tools in a single command and save their outputs in separate files.

Adding the Date of the Scan to the Output

Let's say you want to add the current date to your script's output, or select which scans to run, instead of always running both Nmap and Dirsearch. If you want to write tools with more functionalities like this, you have to understand some advanced shell scripting concepts.

For example, a useful one is *command substitution*, or operating on the output of a command. Using `$()` tells Unix to execute the command surrounded by the parentheses and assign its output to the value of a variable. Let's practice using this syntax:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
TODAY=$(date) ❶
echo "This scan was created on $TODAY" ❷
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
nmap $DOMAIN > $DIRECTORY/nmap
echo "The results of nmap scan are stored in $DIRECTORY/nmap."
$PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
```

At ❶, we assign the output of the `date` command to the variable `TODAY`. The `date` command displays the current date and time. This lets us output a message indicating the day on which we performed the scan ❷.

Adding Options to Choose the Tools to Run

Now, to selectively run only certain tools, you need to use conditionals. In bash, the syntax of an `if` statement is as follows. Note that the conditional statement ends with the `fi` keyword, which is `if` backward:

```
if [ condition 1 ]
then
  # Do if condition 1 is satisfied
elif [ condition 2 ]
then
```

```
  # Do if condition 2 is satisfied, and condition 1 is not satisfied
else
  # Do something else if neither condition is satisfied
fi
```

Let's say that we want users to be able to specify the scan `MODE`, as such:

```
$ ./recon.sh scanmme.nmap.org MODE
```

We can implement this functionality like this:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
TODAY=$(date)
echo "This scan was created on $TODAY"
DIRECTORY=${DOMAIN}_recon
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
if [ $2 == "nmap-only" ] ❶
then
  nmap $DOMAIN > $DIRECTORY/nmap ❷
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
elif [ $2 == "dirsearch-only" ] ❸
then
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php –simple-report=$DIRECTORY/dirsearch ❹
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
else ❺
  nmap $DOMAIN > $DIRECTORY/nmap ❻
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
fi
```

If the user specifies `nmap-only` ❶, we run `nmap` only and store the results to a file named *nmap* ❷. If the user specifies `dirsearch-only` ❸, we execute and store the results of Dirsearch only ❹. If the user specifies neither ❺, we run both scans ❻.

Now you can make your tool run only the Nmap or Dirsearch commands by specifying one of these in the command:

```
$ ./recon.sh scanme.nmap.org nmap-only
$ ./recon.sh scanme.nmap.org dirsearch-only
```

Running Additional Tools

What if you want the option of retrieving information from the crt.sh tool, as well? For example, you want to switch between these three modes or run all three recon tools at once:

```
$ ./recon.sh scanme.nmap.org nmap-only
$ ./recon.sh scanme.nmap.org dirsearch-only
$ ./recon.sh scanme.nmap.org crt-only
```

We could rewrite the `if-else` statements to work with three options: first, we check if `MODE` is `nmap-only`. Then we check if `MODE` is `dirsearch-only`, and finally if `MODE` is `crt-only`. But that's a lot of `if-else` statements, making the code complicated.

Instead, let's use bash's `case` statements, which allow you to match several values against one variable without going through a long list of `if-else` statements. The syntax of `case` statements looks like this. Note that the statement ends with `esac`, or `case` backward:

```
case $VARIABLE_NAME in
  case1)
    Do something
    ;;
  case2)
    Do something
    ;;
  caseN)
    Do something
    ;;
  *)
    Default case, this case is executed if no other case matches.
    ;;
esac
```

We can improve our script by implementing the functionality with `case` statements instead of multiple `if-else` statements:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
TODAY=$(date)
echo "This scan was created on $TODAY"
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
case $2 in
  nmap-only)
    nmap $DOMAIN > $DIRECTORY/nmap
    echo "The results of nmap scan are stored in $DIRECTORY/nmap."
    ;;
  dirsearch-only)
    $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
    echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
    ;;
  crt-only)
    curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt ❶
    echo "The results of cert parsing is stored in $DIRECTORY/crt."
    ;;
  *)
    nmap $DOMAIN > $DIRECTORY/nmap
    echo "The results of nmap scan are stored in $DIRECTORY/nmap."
    $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
    echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
```

```
    curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt
    echo "The results of cert parsing is stored in $DIRECTORY/crt."
    ;;
esac
```

The `curl` command ❶ downloads the content of a page. We use it here to download data from crt.sh. And `curl`'s `-o` option lets you specify an output file. But notice that our code has a lot of repetition! The sections of code that run each type of scan repeat twice. Let's try to reduce the repetition by using functions. The syntax of a bash function looks like this:

```
FUNCTION_NAME()
{
  DO_SOMETHING
}
```

After you've declared a function, you can call it like any other shell command within the script. Let's add functions to the script:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
TODAY=$(date)
echo "This scan was created on $TODAY"
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
nmap_scan() ❶
{
  nmap $DOMAIN > $DIRECTORY/nmap
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
}
dirsearch_scan() ❷
{
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
}
crt_scan() ❸
{
  curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt
  echo "The results of cert parsing is stored in $DIRECTORY/crt."
}
case $2 in ❹
  nmap-only)
    nmap_scan
    ;;
  dirsearch-only)
    dirsearch_scan
    ;;
  crt-only)
    crt_scan
    ;;
  *)
    nmap_scan
```

```
    dirsearch_scan
    crt_scan
    ;;
esac
```

You can see that we've simplified our code. We created three functions, `nmap_scan` ❶, `dirsearch_scan` ❷, and `crt_scan` ❸. We put the `scan` and `echo` commands in these functions so we can call them repeatedly without writing the same code over and over ❹. This simplification might not seem like much here, but reusing code with functions will save you a lot of headaches when you write more complex programs.

Keep in mind that all bash variables are *global* except for input parameters like `$1`, `$2`, and `$3`. This means that variables like `$DOMAIN`, `$DIRECTORY`, and `$PATH_TO_DIRSEARCH` become available throughout the script after we've declared them, even if they're declared within functions. On the other hand, parameter values like `$1`, `$2`, and `$3` can refer only to the values the function is called with, so you can't use a script's input arguments within a function, like this:

```
nmap_scan()
{
  nmap $1 > $DIRECTORY/nmap
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
}
nmap_scan
```

Here, the `$1` in the function refers to the first argument that `nmap_scan` was called with, not the argument our *recon.sh* script was called with. Since `nmap_scan` wasn't called with any arguments, `$1` is blank.

Parsing the Results

Now we have a tool that performs three types of scans and stores the results into files. But after the scans, we'd still have to manually read and make sense of complex output files. Is there a way to speed up this process too?

Let's say you want to search for a certain piece of information in the output files. You can use *Global Regular Expression Print (grep)* to do that. This command line utility is used to perform searches in text, files, and command outputs. A simple `grep` command looks like this:

```
grep password file.txt
```

This tells `grep` to search for the string `password` in the file *file.txt*, then print the matching lines in standard output. For example, we can quickly search the Nmap output file to see if the target has port 80 open:

```
$ grep 80 TARGET_DIRECTORY/nmap
80/tcp open http
```

You can also make your search more flexible by using regular expressions in your search string. A *regular expression*, or *regex*, is a special string

that describes a search pattern. It can help you display only specific parts of the output. For example, you may have noticed that the output of the Nmap command looks like this:

```
Starting Nmap 7.60 ( https://nmap.org )
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.065s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2f
Not shown: 992 closed ports
PORT STATE SERVICE
22/tcp open ssh
25/tcp filtered smtp
80/tcp open http
135/tcp filtered msrpc
139/tcp filtered netbios-ssn
445/tcp filtered microsoft-ds
9929/tcp open nping-echo
31337/tcp open Elite
Nmap done: 1 IP address (1 host up) scanned in 2.43 seconds
```

You might want to trim the irrelevant messages from the file so it looks more like this:

```
PORT STATE SERVICE
22/tcp open ssh
25/tcp filtered smtp
80/tcp open http
135/tcp filtered msrpc
139/tcp filtered netbios-ssn
445/tcp filtered microsoft-ds
9929/tcp open nping-echo
31337/tcp open Elite
```

Use this command to filter out the messages at the start and end of Nmap's output and keep only the essential part of the report:

```
grep -E "^\S+\s+\S+\s+\S+$" DIRECTORY/nmap > DIRECTORY/nmap_cleaned
```

The `-E` flag tells `grep` you're using a regex. A regex consists of two parts: constants and operators. *Constants* are sets of strings, while *operators* are symbols that denote operations over these strings. These two elements together make regex a powerful tool of pattern matching. Here's a quick overview of regex operators that represent characters:

`\d` matches any digit.

`\w` matches any character.

`\s` matches any whitespace, and `\S` matches any non-whitespace.

`.` matches with any single character.

`\` escapes a special character.

`^` matches the start of the string or line.

`$` matches the end of the string or line.

Several operators also specify the number of characters to match:

* matches the preceding character zero or more times.

+ matches the preceding character one or more times.

{3} matches the preceding character three times.

{1, 3} matches the preceding character one to three times.

{1, } matches the preceding character one or more times.

[*abc*] matches one of the characters within the brackets.

[*a*-*z*] matches one of the characters within the range of *a* to *z*.

(*a*|*b*|*c*) matches either *a* or *b* or *c*.

Let's take another look at our regex expression here. Remember how \s matches any whitespace, and \S matches any non-whitespace? This means \s+ would match any whitespace one or more characters long, and \S+ would match any non-whitespace one or more characters long. This regex pattern specifies that we should extract lines that contain three strings separated by two whitespaces:

```
"^\S+\s+\S+\s+\S+$"
```

The filtered output will look like this:

```
PORT STATE SERVICE
22/tcp open ssh
25/tcp filtered smtp
80/tcp open http
135/tcp filtered msrpc
139/tcp filtered netbios-ssn
445/tcp filtered microsoft-ds
9929/tcp open nping-echo
31337/tcp open Elite
```

To account for extra whitespaces that might be in the command output, let's add two more optional spaces around our search string:

```
"^\s*\S+\s+\S+\s+\S+\s*$"
```

You can use many more advanced regex features to perform more sophisticated matching. However, this simple set of operators serves well for our purposes. For a complete guide to regex syntax, read RexEgg's cheat sheet (*https://www.rexegg.com/regex-quickstart.html*).

Building a Master Report

What if you want to produce a master report from all three output files? You need to parse the JSON file from crt.sh. You can do this with jq, a command line utility that processes JSON. If we examine the JSON output file from crt.sh, we can see that we need to extract the name_value field of each certificate item to extract domain names. This command does just that:

```
$ jq -r ".[] | .name_value" $DOMAIN/crt
```

The -r flag tells jq to write the output directly to standard output rather than format it as JSON strings. The .[] iterates through the array within the JSON file, and .name_value extracts the name_value field of each item. Finally, $DOMAIN/crt is the input file to the jq command. To learn more about how jq works, read its manual (*https://stedolan.github.io/jq/manual/*).

To combine all output files into a master report, write a script like this:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
nmap_scan()
{
  nmap $DOMAIN > $DIRECTORY/nmap
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
}
dirsearch_scan()
{
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
}
crt_scan()
{
  curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt
  echo "The results of cert parsing is stored in $DIRECTORY/crt."
}
case $2 in
  nmap-only)
    nmap_scan
    ;;
  dirsearch-only)
    dirsearch_scan
    ;;
  crt-only)
    crt_scan
    ;;
  *)
    nmap_scan
    dirsearch_scan
    crt_scan
    ;;
esac
echo "Generating recon report from output files..."
TODAY=$(date)
echo "This scan was created on $TODAY" > $DIRECTORY/report ❶
echo "Results for Nmap:" >> $DIRECTORY/report
grep -E "^\s*\S+\s+\S+\s+\S+\s*$" $DIRECTORY/nmap >> $DIRECTORY/report ❷
echo "Results for Dirsearch:" >> $DIRECTORY/report
cat $DIRECTORY/dirsearch >> $DIRECTORY/report ❸
echo "Results for crt.sh:" >> $DIRECTORY/report
jq -r ".[] | .name_value" $DIRECTORY/crt >> $DIRECTORY/report ❹
```

First, we create a new file named *report* and write today's date into it ❶ to keep track of when the report was generated. We then append the results of the `nmap` and `dirsearch` commands into the report file ❷. The `cat` command prints the contents of a file to standard output, but we can also use it to redirect the content of the file into another file ❸. Finally, we extract domain names from the crt.sh report and append it to the end of the report file ❹.

Scanning Multiple Domains

What if we want to scan multiple domains at once? When reconning a target, we might start with several of the organization's domain names. For example, we know that Facebook owns both *facebook.com* and *fbcdn.net*. But our current script allows us to scan only one domain at a time. We need to write a tool that can scan multiple domains with a single command, like this:

```
./recon.sh facebook.com fbcdn.net nmap-only
```

When we scan multiple domains like this, we need a way to distinguish which arguments specify the scan `MODE` and which specify target domains. As you've already seen from the tools I introduced, most tools allow users to modify the behavior of a tool by using command line *options* or *flags*, such as `-u` and `--simple-report`.

The `getopts` tool parses options from the command line by using single-character flags. Its syntax is as follows, where *`OPTSTRING`* specifies the option letters that `getopts` should recognize. For example, if it should recognize the options `-m` and `-i`, you should specify `mi`. If you want an option to contain argument values, the letter should be followed by a colon, like this: `m:i`. The *`NAME`* argument specifies the variable name that stores the option letter.

```
getopts OPTSTRING NAME
```

To implement our multiple-domain scan functionality, we can let users use an `-m` flag to specify the scan mode and assume that all other arguments are domains. Here, we tell `getopts` to recognize an option if the option flag is `-m` and that this option should contain an input value. The `getopts` tool also automatically stores the value of any options into the `$OPTARG` variable. We can store that value into a variable named `MODE`:

```
getopts "m:" OPTION
MODE=$OPTARG
```

Now if you run the shell script with an `-m` flag, the script will know that you're specifying a scan `MODE`! Note that `getopts` stops parsing arguments when it encounters an argument that doesn't start with the `-` character, so you'll need to place the scan mode before the domain arguments when you run the script:

```
./recon.sh -m nmap-only facebook.com fbcdn.net
```

Next, we'll need a way to read every domain argument and perform scans on them. Let's use loops! Bash has two types of loops: the `for` loop and the `while` loop. The `for` loop works better for our purposes, as we already know the number of values we are looping through. In general, you should use `for` loops when you already have a list of values to iterate through. You should use `while` loops when you're not sure how many values to loop through but want to specify the condition in which the execution should stop.

Here's the syntax of a `for` loop in bash. For every item in *`LIST_OF_VALUES`*, bash will execute the code between `do` and `done` once:

```
for i in LIST_OF_VALUES
do
  DO SOMETHING
done
```

Now let's implement our functionality by using a `for` loop:

```
❶ for i in "${@:$OPTIND:$#}"
do
  # Do the scans for $i
done
```

We create an array ❶ that contains every command line argument, besides the ones that are already parsed by `getopts`, which stores the index of the first argument after the options it parses into a variable named `$OPTIND`. The characters `$@` represent the array containing all input arguments, while `$#` is the number of command line arguments passed in. `"${@:OPTIND:}"` slices the array so that it removes the `MODE` argument, like `nmap-only`, making sure that we iterate through only the domains part of our input. Array slicing is a way of extracting a subset of items from an array. In bash, you can slice arrays by using this syntax (note that the quotes around the command are necessary):

```
"${INPUT_ARRAY:START_INDEX:END_INDEX}"
```

The `$i` variable represents the current item in the argument array. We can then wrap the loop around the code:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
nmap_scan()
{
  nmap $DOMAIN > $DIRECTORY/nmap
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
}
dirsearch_scan()
{
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
}
crt_scan()
{
```

```
  curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt
  echo "The results of cert parsing is stored in $DIRECTORY/crt."
}
getopts "m:" OPTION
MODE=$OPTARG

for i in "${@:$OPTIND:$#}" ❶
do

  DOMAIN=$i
  DIRECTORY=${DOMAIN}_recon
  echo "Creating directory $DIRECTORY."
  mkdir $DIRECTORY

  case $MODE in
    nmap-only)
      nmap_scan
      ;;
    dirsearch-only)
      dirsearch_scan
      ;;
    crt-only)
      crt_scan
      ;;
    *)
      nmap_scan
      dirsearch_scan
      crt_scan
      ;;
  esac
  echo "Generating recon report for $DOMAIN..."
  TODAY=$(date)
  echo "This scan was created on $TODAY" > $DIRECTORY/report
    if [ -f $DIRECTORY/nmap ];then ❷
    echo "Results for Nmap:" >> $DIRECTORY/report
    grep -E "^\s*\S+\s+\S+\s+\S+\s*$" $DIRECTORY/nmap >> $DIRECTORY/report
  fi
    if [ -f $DIRECTORY/dirsearch ];then ❸
    echo "Results for Dirsearch:" >> $DIRECTORY/report
    cat $DIRECTORY/dirsearch >> $DIRECTORY/report
  fi
    if [ -f $DIRECTORY/crt ];then ❹
    echo "Results for crt.sh:" >> $DIRECTORY/report
    jq -r ".[] | .name_value" $DIRECTORY/crt >> $DIRECTORY/report
  fi
  done ❺
```

The `for` loop starts with the `for` keyword ❶ and ends with the `done` keyword ❺. Notice that we also added a few lines in the report section to see if we need to generate each type of report. We check whether the output file of an Nmap scan, a Dirsearch scan, or a crt.sh scan exist so we can determine if we need to generate a report for that scan type ❷ ❸ ❹.

The brackets around a condition mean that we're passing the contents into a `test` command: `[ -f $DIRECTORY/nmap ]` is equivalent to `test -f $DIRECTORY/nmap`.

The `test` command evaluates a conditional and outputs either `true` or `false`. The `-f` flag tests whether a file exists. But you can test for more conditions! Let's go through some useful test conditions. The `-eq` and `-ne` flags test for equality and inequality, respectively. This returns `true` if `$3` is equal to `1`:

```
if [ $3 -eq 1 ]
```

This returns `true` if `$3` is not equal to `1`:

```
if [ $3 -ne 1 ]
```

The `-gt`, `-ge`, `-lt`, and `le` flags test for greater than, greater than or equal to, less than, and less than or equal to, respectively:

```
if [ $3 -gt 1 ]
if [ $3 -ge 1 ]
if [ $3 -lt 1 ]
if [ $3 -le 1 ]
```

The `-z` and `-n` flags test whether a string is empty. These conditions are both true:

```
if [ -z "" ]
if [ -n "abc" ]
```

The `-d`, `-f`, `-r`, `-w`, and `-x` flags check for directory and file statuses. You can use them to check the existence and permissions of a file before your shell script operates on them. For instance, this command returns `true` if */bin* is a directory that exists:

```
if [ -d /bin]
```

This one returns `true` if */bin/bash* is a file that exists:

```
if [ -f /bin/bash ]
```

And this one returns `true` if */bin/bash* is a readable file:

```
if [ -r /bin/bash ]
```

or a writable file:

```
if [ -w /bin/bash ]
```

or an executable file:

```
if [ -x /bin/bash ]
```

You can also use && and || to combine test expressions. This command returns true if both expressions are true:

```
if [ $3 -gt 1 ] && [ $3 -lt 3 ]
```

And this one returns true if at least one of them is true:

```
if [ $3 -gt 1 ] || [ $3 -lt 0 ]
```

You can find more comparison flags in the test command's manual by running man test. (If you aren't sure about the commands you're using, you can always enter man followed by the command name in the terminal to access the command's manual file.)

Writing a Function Library

As your codebase gets larger, you should consider writing a *function library* to reuse code. We can store all the commonly used functions in a separate file called *scan.lib*. That way, we can call these functions as needed for future recon tasks:

```
#!/bin/bash
nmap_scan()
{
  nmap $DOMAIN > $DIRECTORY/nmap
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
}
dirsearch_scan()
{
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
}
crt_scan()
{
  curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt
  echo "The results of cert parsing is stored in $DIRECTORY/crt."
}
```

In another file, we can source the library file in order to use all of its functions and variables. We source a script via the source command, followed by the path to the script:

```
#!/bin/bash
source ./scan.lib
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
getopts "m:" OPTION
MODE=$OPTARG
for i in "${@:$OPTIND:$#}"
do
  DOMAIN=$i
  DIRECTORY=${DOMAIN}_recon
  echo "Creating directory $DIRECTORY."
  mkdir $DIRECTORY
```

```
  case $MODE in
    nmap-only)
      nmap_scan
      ;;
    dirsearch-only)
      dirsearch_scan
      ;;
    crt-only)
      crt_scan
      ;;
    *)
      nmap_scan
      dirsearch_scan
      crt_scan
      ;;
  esac
  echo "Generating recon report for $DOMAIN..."
  TODAY=$(date)
  echo "This scan was created on $TODAY" > $DIRECTORY/report
  if [ -f $DIRECTORY/nmap ];then
    echo "Results for Nmap:" >> $DIRECTORY/report
    grep -E "^\s*\S+\s+\S+\s+\S+\s*$" $DIRECTORY/nmap >> $DIRECTORY/report
  fi
  if [ -f $DIRECTORY/dirsearch ];then
    echo "Results for Dirsearch:" >> $DIRECTORY/report
    cat $DIRECTORY/dirsearch >> $DIRECTORY/report
  fi
  if [ -f $DIRECTORY/crt ];then
    echo "Results for crt.sh:" >> $DIRECTORY/report
    jq -r ".[] | .name_value" $DIRECTORY/crt >> $DIRECTORY/report
  fi
done
```

Using a library can be super useful when you're building multiple tools that require the same functionalities. For example, you might build multiple networking tools that all require DNS resolution. In this case, you can simply write the functionality once and use it in all of your tools.

Building Interactive Programs

What if you want to build an interactive program that takes user input during execution? Let's say that if users enter the command line option, `-i`, you want the program to enter an interactive mode that allows you to specify domains to scan as you go:

```
./recon.sh -i -m nmap-only
```

For that, you can use `read`. This command reads user input and stores the input string into a variable:

```
echo "Please enter a domain!"
read $DOMAIN
```

These commands will prompt the user to enter a domain, then store the input inside a variable named `$DOMAIN`.

To prompt a user repeatedly, we need to use a `while` loop, which will keep printing the prompt asking for an input domain until the user exits the program. Here's the syntax of a `while` loop. As long as the *`CONDITION`* is true, the `while` loop will execute the code between `do` and `done` repeatedly:

```
while CONDITION
do
  DO SOMETHING
done
```

We can use a `while` loop to repeatedly prompt the user for domains until the user enters `quit`:

```
while [ $INPUT != "quit" ];do
  echo "Please enter a domain!"
  read INPUT
  if [ $INPUT != "quit" ];then
    scan_domain $INPUT
    report_domain $INPUT
  fi
done
```

We also need a way for users to actually invoke the `-i` option, and our `getopts` command isn't currently doing that. We can use a `while` loop to parse options by using `getopts` repeatedly:

```
while getopts "m:i" OPTION; do
  case $OPTION in
    m)
        MODE=$OPTARG
        ;;
    i)
        INTERACTIVE=true
        ;;
  esac
done
```

Here, we specify a `while` loop that gets command line options repeatedly. If the option flag is `-m`, we set the `MODE` variable to the scan mode that the user has specified. If the option flag is `-i`, we set the `$INTERACTIVE` variable to `true`. Then, later in the script, we can decide whether to invoke the interactive mode by checking the value of the `$INTERACTIVE` variable. Putting it all together, we get our final script:

```
#!/bin/bash
source ./scan.lib

while getopts "m:i" OPTION; do
  case $OPTION in
    m)
      MODE=$OPTARG
```

```
      ;;
    i)
      INTERACTIVE=true
      ;;
  esac
done

scan_domain(){
  DOMAIN=$1
  DIRECTORY=${DOMAIN}_recon
  echo "Creating directory $DIRECTORY."
  mkdir $DIRECTORY
  case $MODE in
    nmap-only)
      nmap_scan
      ;;
    dirsearch-only)
      dirsearch_scan
      ;;
    crt-only)
      crt_scan
      ;;
    *)
      nmap_scan
      dirsearch_scan
      crt_scan
      ;;
  esac
}
report_domain(){
  DOMAIN=$1
  DIRECTORY=${DOMAIN}_recon
  echo "Generating recon report for $DOMAIN..."
 TODAY=$(date)
  echo "This scan was created on $TODAY" > $DIRECTORY/report
  if [ -f $DIRECTORY/nmap ];then
   echo "Results for Nmap:" >> $DIRECTORY/report
    grep -E "^\s*\S+\s+\S+\s+\S+\s*$" $DIRECTORY/nmap >> $DIRECTORY/report
  fi
  if [ -f $DIRECTORY/dirsearch ];then
    echo "Results for Dirsearch:" >> $DIRECTORY/report
    cat $DIRECTORY/dirsearch >> $DIRECTORY/report
  fi
  if [ -f $DIRECTORY/crt ];then
    echo "Results for crt.sh:" >> $DIRECTORY/report
    jq -r ".[] | .name_value" $DIRECTORY/crt >> $DIRECTORY/report
  fi
}
if [ $INTERACTIVE ];then ❶
  INPUT="BLANK"
  while [ $INPUT != "quit" ];do ❷
    echo "Please enter a domain!"
    read INPUT
    if [ $INPUT != "quit" ];then ❸
      scan_domain $INPUT
```

```
      report_domain $INPUT
    fi
  done
else
  for i in "${@:$OPTIND:$#}";do
    scan_domain $i
    report_domain $i

  done
fi
```

In this program, we first check if the user has selected the interactive mode by specifying the `-i` option ❶. We then repeatedly prompt the user for a domain by using a `while` loop ❷. If the user input is not the keyword `quit`, we assume that they entered a target domain, so we scan and produce a report for that domain. The `while` loop will continue to run and ask the user for domains until the user enters `quit`, which will cause the `while` loop to exit and the program to terminate ❸.

Interactive tools can help your workflow operate more smoothly. For example, you can build testing tools that will let you choose how to proceed based on preliminary results.

Using Special Variables and Characters

You're now equipped with enough bash knowledge to build many versatile tools. This section offers more tips that concern the particularities of shell scripts.

In Unix, commands return `0` on success and a positive integer on failure. The variable `$?` contains the exit value of the last command executed. You can use these to test for execution successes and failures:

```
#!/bin/sh
chmod 777 script.sh
if [ "$?" -ne "0" ]; then
  echo "Chmod failed. You might not have permissions to do that!"
fi
```

Another special variable is `$$`, which contains the current process's ID. This is useful when you need to create temporary files for the script. If you have multiple instances of the same script or program running at the same time, each might need its own temporary files. In this case, you can create temporary files named */tmp/script_name_$$* for every one of them.

Remember that we talked about variable scopes in shell scripts earlier in this chapter? Variables that aren't input parameters are global to the entire script. If you want other programs to use the variable as well, you need to export the variable:

```
export VARIABLE_NAME=VARIABLE_VALUE
```

Let's say that in one of your scripts you set the variable `VAR`:

```
VAR="hello!"
```

If you don't export it or source it in another script, the value gets destroyed after the script exits. But if you export `VAR` in the first script and run that script before running a second script, the second script will be able to read `VAR`'s value.

You should also be aware of special characters in bash. In Unix, the wildcard character * stands for *all*. For example, this command will print out all the filenames in the current directory that have the file extension *.txt*:

```
$ ls *.txt
```

Backticks (`) indicate command substitution. You can use both backticks and the `$()` command substitution syntax mentioned earlier for the same purpose. This `echo` command will print the output of the `whoami` command:

```
echo `whoami`
```

Most special characters, like the wildcard character or the single quote, aren't interpreted as special when they are placed in double quotes. Instead, they're treated as part of a string. For example, this command will echo the string `"abc '*' 123"`:

```
$ echo "abc '*' 123"
```

Another important special character is the backslash (\), the escape character in bash. It tells bash that a certain character should be interpreted literally, and not as a special character.

Certain special characters, like double quotes, dollar sign, backticks, and backslashes remain special even within double quotes, so if you want bash to treat them literally, you have to escape them by using a backslash:

```
$ echo "\" is a double quote. \$ is a dollar sign. \` is a backtick. \\ is a backslash."
```

This command will echo:

```
" is a double quote. $ is a dollar sign. ` is a backtick. \ is a backslash.
```

You can also use a backslash before a newline to indicate that the line of code has not ended. For example, this command

```
chmod 777 \
script.sh
```

is the same as this one:

```
chmod 777 script.sh
```

Congratulations! You can now write bash scripts. Bash scripting may seem scary at first, but once you've mastered it, it will be a powerful addition to your hacking arsenal. You'll be able to perform better recon, conduct more efficient testing, and have a more structured hacking workflow.

If you plan on implementing a lot of automation, it's a good idea to start organizing your scripts from the start. Set up a directory of scripts and sort your scripts by their functionality. This will become the start of developing your own hacking methodology. When you've collected a handful of scripts that you use on a regular basis, you can use scripts to run them automatically. For example, you might categorize your scripts into recon scripts, fuzzing scripts, automated reporting, and so on. This way, every time you find a script or tool you like, you can quickly incorporate it into your workflow in an organized fashion.

Scheduling Automatic Scans

Now let's take your automation to the next level by building an alert system that will let us know if something interesting turns up in our scans. This saves us from having to run the commands manually and comb through the results over and over again.

We can use cron jobs to schedule our scans. *Cron* is a job scheduler on Unix-based operating systems. It allows you to schedule jobs to run periodically. For example, you can run a script that checks for new endpoints on a particular site every day at the same time. Or you can run a scanner that checks for vulnerabilities on the same target every day. This way, you can monitor for changes in an application's behavior and find ways to exploit it.

You can configure Cron's behavior by editing files called *crontabs*. Unix keeps different copies of crontabs for each user. Edit your own user's crontab by running the following:

```
crontab -e
```

All crontabs follow this same syntax:

```
A B C D E command_to_be_executed
A: Minute (0 – 59)
B: Hour (0 – 23)
C: Day (1 – 31)
D: Month (1 – 12)
E: Weekday (0 – 7) (Sunday is 0 or 7, Monday is 1...)
```

Each line specifies a command to be run and the time at which it should run, using five numbers. The first number, from 0 to 59, specifies the minute when the command should run. The second number specifies the hour, and ranges from 0 to 23. The third and fourth numbers are the day and month the command should run. And the last number is the weekday when the command should run, which ranges from 0 to 7. Both 0 and 7 mean that the command should run on Sundays; 1 means the command should run on Mondays; and so on.

For example, you can add this line to your crontab to run your recon script every day at 9:30 PM:

```
30 21 * * * ./scan.sh
```

You can also batch-run the scripts within directories. The `run-parts` command in crontabs tells Cron to run all the scripts stored in a directory. For example, you can store all your recon tools in a directory and scan your targets periodically. The following line tells Cron to run all scripts in my security directory every day at 9:30 PM:

```
30 21 * * * run-parts /Users/vickie/scripts/security
```

Next, `git diff` is a command that outputs the difference between two files. You need to install the Git program to use it. You can use `git diff` to compare scan results at different times, which quickly lets you see if the target has changed since you last scanned it:

```
git diff SCAN_1 SCAN_2
```

This will help you identify any new domains, subdomains, endpoints, and other new assets of a target. You could write a script like this to notify you of new changes on a target every day:

```
#!/bin/bash
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon
echo "Checking for new changes about the target: $DOMAIN.\n Found these new things."
git diff <SCAN AT TIME 1> <SCAN AT TIME 2>
```

And schedule it with Cron:

```
30 21 * * * ./scan_diff.sh facebook.com
```

These automation techniques have helped me quickly find new JavaScript files, endpoints, and functionalities on targets. I especially like to use this technique to discover subdomain takeover vulnerabilities automatically. We'll talk about subdomain takeovers in Chapter 20.

Alternatively, you can use GitHub to track changes. Set up a repository to store your scan results at *https://github.com/new/*. GitHub has a Notification feature that will tell you when significant events on a repository occur. It's located at Settings ▸ Notifications on each repository's page. Provide GitHub with an email address that it will use to notify you about changes. Then, in the directory where you store scan results, run these commands to initiate `git` inside the directory:

```
git init
git remote add origin https://PATH_TO_THE_REPOSITORY
```

Lastly, use Cron to scan the target and upload the files to GitHub periodically:

```
30 21 * * * ./recon.sh facebook.com
40 21 * * * git add *; git commit -m "new scan"; git push -u origin master
```

GitHub will then send you an email about the files that changed during the new scan.

A Note on Recon APIs

Many of the tools mentioned in this chapter have APIs that allow you to integrate their services into your applications and scripts. We'll talk about APIs more in Chapter 24, but for now, you can think of APIs as endpoints you can use to query a service's database. Using these APIs, you can query recon tools from your script and add the results to your recon report without visiting their sites manually.

For example, Shodan has an API (*https://developer.shodan.io/*) that allows you to query its database. You can access a host's scan results by accessing this URL: *https://api.shodan.io/shodan/host/{ip}?key={YOUR_API_KEY}*. You could configure your bash script to send requests to this URL and parse the results. LinkedIn also has an API (*https://www.linkedin.com/developers/*) that lets you query its database. For example, you can use this URL to access information about a user on LinkedIn: *https://api.linkedin.com/v2/people/{PERSON ID}*. The Censys API (*https://censys.io/api*) allows you to access certificates by querying the endpoint *https://censys.io/api/v1*.

Other tools mentioned in this chapter, like BuiltWith, Google search, and GitHub search, all have their own API services. These APIs can help you discover assets and content more efficiently by integrating third-party tools into your recon script. Note that most API services require you to create an account on their website to obtain an *API key*, which is how most API services authenticate their users. You can find information about how to obtain the API keys of popular recon services at *https://github.com/lanmaster53/recon-ng-marketplace/wiki/API-Keys/*.

Start Hacking!

Now that you've conducted extensive reconnaissance, what should you do with the data you've collected? Plan your attacks by using the information you've gathered! Prioritize your tests based on the functionality of the application and its technology.

For example, if you find a feature that processes credit card numbers, you could first look for vulnerabilities that might leak the credit card numbers, such as IDORs (Chapter 10). Focus on sensitive features such as credit cards and passwords, because these features are more likely to contain critical vulnerabilities. During your recon, you should be able to get a good idea of what the company cares about and the sensitive data it's protecting. Go after those specific pieces of information throughout your bug-hunting process to maximize the business impact of the issues you discover. You can also focus your search on bugs or vulnerabilities that affect that particular tech stack you uncovered, or on elements of the source code you were able to find.

And don't forget, recon isn't a one-time activity. You should continue to monitor your targets for changes. Organizations modify their system, technologies, and codebase constantly, so continuous recon will ensure that you always know what the attack surface looks like. Using a combination of bash, scheduling tools, and alerting tools, build a recon engine that does most of the work for you.