Sprinkler: A Multi-Service Password Sprayer

Prompt Eua-anant, Sam Ederington

Introduction

Have you ever talked with someone and thought, “I bet I could guess their social media account password?” Well, if you have, then a password sprayer is your best tool. First, though, you should verify with them before trying, because without that this is illegal![1]

In a nutshell, a password sprayer...

- Sends a bunch of usernames and passwords to a remote server **rapidly**
- Then tells you which ones grant you access

Why should you care?

- Hackers use password sprayers to **gain unauthorized access** to accounts with weak passwords
- If you own a server that allows users to authenticate themselves, you want to know how a password sprayer works, and how to protect against password-spraying attacks.
- A password sprayer can help you learn about password-spraying tactics and see how easy it is to gain unauthorized access to someone’s account on your server.

What we accomplished

- We made Sprinkler, a password sprayer written in C, that supports the following services: ssh, http-get, and http-post

[1] https://www.ftc.gov/legal-library/browse/rules/identity-theft-assumption-deterrence-act-text

- Sprinkler has extra options available, such as connecting via TLS, delay between attempts, input regex that determines login success/failure, input login parameter names and values, verbose mode, and help pages.
- We created a Makefile that (1) compiles the source files, (2) links with third-party libraries, and (3) allows the user to install it in their system.
- Spraying login credentials to someone else's server requires permission, otherwise, it would be gravely unethical and illegal. Accordingly, we set up our own Apache2 target servers on Ubuntu compatible with all the services above, so we can test Sprinkler freely and ethically.

Introduction to Services

Before going in-depth on how Sprinkler works, here's an introduction to each service Sprinkler offers, and how they differ from each other.

SSH (**S**ecure **sh**ell)

A protocol that allows you to gain access to the OS of a remote server. Once you've successfully entered the login credentials, you get a shell where you can type in OS commands, and they'll be executed on the remote server![2] Moreover, SSH uses a combination of public-key cryptography, symmetric encryption, message authentication algorithm, and hash algorithm to ensure that (1) all communication between you and the remote server is secure, meaning that it won't be intelligible to third parties who intercept the communication, and (2) ensure data integrity, that is, it can detect whether the messages get corrupted or modified.[3]

HTTP-GET

In the context of password spraying, this refers to HTTP/HTTPS server that accepts login credentials via a GET request. However, while there are different authentication schemes by which credentials are submitted through a GET request (eg. Basic, Digest, Negotiate)[4], Sprinkler only supports one authentication scheme—Basic, also known as basic auth.

In normal contexts (not password spraying), here's how basic auth works:

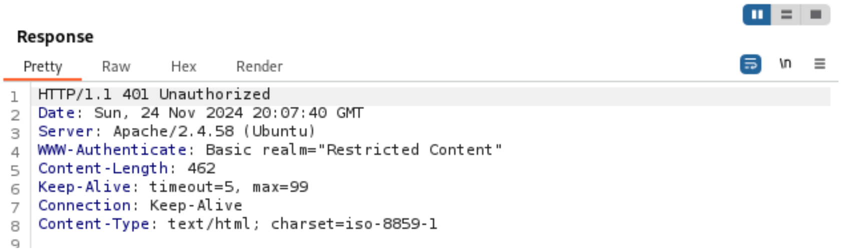
1. User's server sends a GET request to the target server

[2] https://www.cloudflare.com/learning/access-management/what-is-ssh/#:~:text=The%20Secure%20Shell%20(SSH)%20protocol%20is%20a%20method%20for%20securely,and%20encrypt%20connections%20between%20devices.

[3] https://datatracker.ietf.org/doc/html/rfc4253#section-1

[4] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/WWW-Authenticate#directives

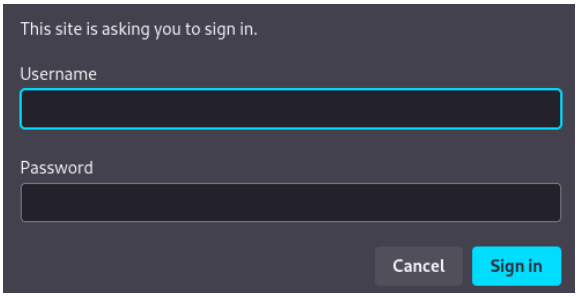
2. The target server sends the user a 401 Unauthorized response code, and one of the response headers is WWW-Authenticate, with a value of ‘Basic realm = …’
    a. ‘Basic’ indicates that the server uses basic auth
    b. The ‘realm’ variable is optional and indicates the name of the group of resources protected by the same authentication scheme and/or authorization database.[5] The user may reuse the same login credentials for all resources in the same realm.[6]



```
HTTP/1.1 401 Unauthorized
Date: Sun, 24 Nov 2024 20:07:40 GMT
Server: Apache/2.4.58 (Ubuntu)
WWW-Authenticate: Basic realm="Restricted Content"
Content-Length: 462
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1
```

Server’s response with 401 Unauthorized

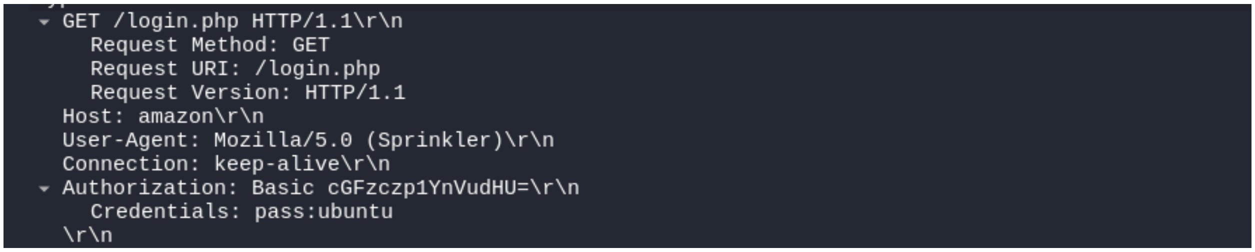
3. This pop-up appears:



This site is asking you to sign in.

4. Once the user typed their credentials, the browser concatenates **username** and **password** into the string “**username**:**password**” and encodes it into base64, ensuring that all characters are printable ASCII characters.[7][8]
5. The browser prepares a new GET request, and this time it adds the encoded text to the request header “Authorization”

[5] https://datatracker.ietf.org/doc/html/rfc7235#section-2.2
[6] Ibid.
[7] https://datatracker.ietf.org/doc/html/rfc7617#section-1
[8] https://developer.mozilla.org/en-US/docs/Glossary/Base64



```
▾ GET /login.php HTTP/1.1\r\n
    Request Method: GET
    Request URI: /login.php
    Request Version: HTTP/1.1
  Host: amazon\r\n
  User-Agent: Mozilla/5.0 (Sprinkler)\r\n
  Connection: keep-alive\r\n
▾ Authorization: Basic cGFzczp1YnVudHU=\r\n
    Credentials: pass:ubuntu
  \r\n
```

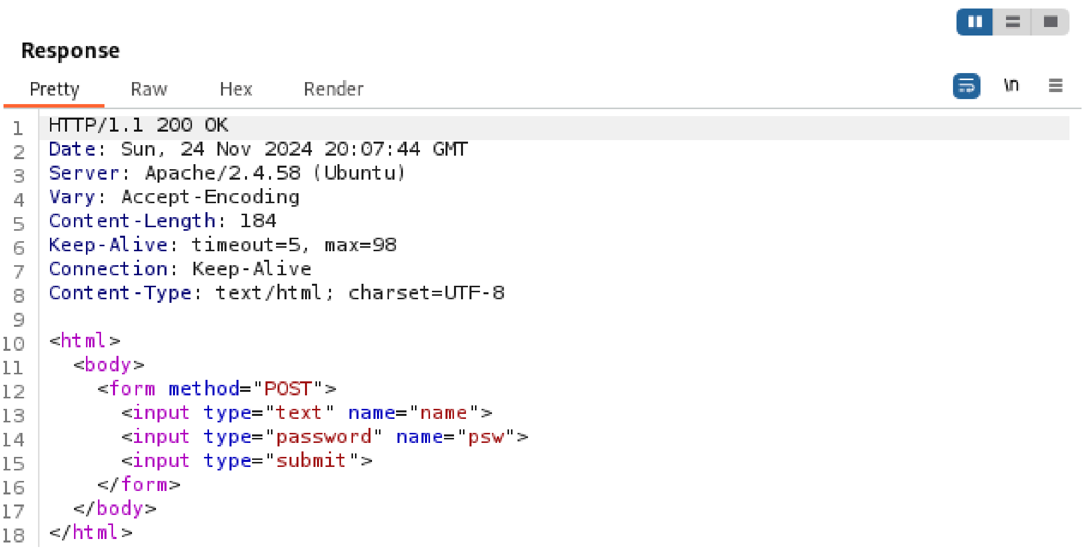
6. Send this new GET request to server.
7. The server's response to the user depends on whether the credentials are correct or not.

For password sprayers, the process starts at step 4, but instead of typing the credentials into the browser, the user inputs password and username files to a password sprayer, which encodes and formats them properly in a GET request and sends them to the server.

HTTP-POST

This refers to HTTP/HTTPS that uses form-based authentication. Here's how it works:

1. The user sends a GET request to the server
2. Instead of sending an ugly pop-up like basic auth, the server sends a customizable login form, containing the variable names it expects the user to use.[9]



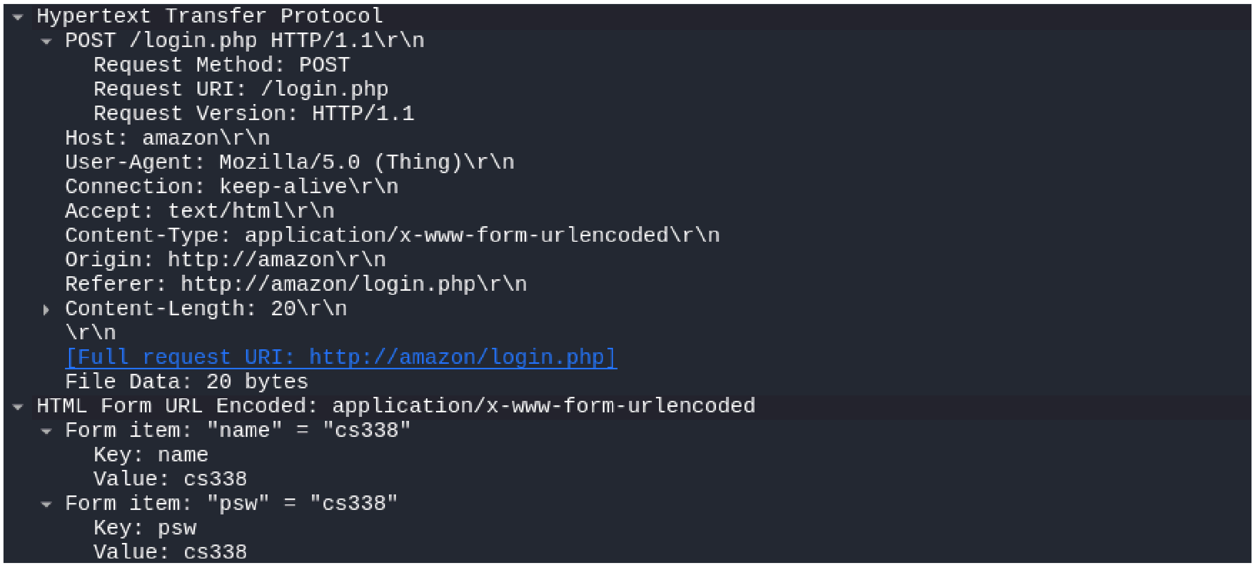
```
HTTP/1.1 200 OK
Date: Sun, 24 Nov 2024 20:07:44 GMT
Server: Apache/2.4.58 (Ubuntu)
Vary: Accept-Encoding
Content-Length: 184
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

<html>
  <body>
    <form method="POST">
      <input type="text" name="name">
      <input type="password" name="psw">
      <input type="submit">
    </form>
  </body>
</html>
```

In the image above, the login form has the variables "name" and "psw" corresponding to the username and password.

[9] https://docs.oracle.com/cd/E15217_01/doc.1014/e12488/v2form.htm

3. The user types in the username and password in the login form and click submit.
4. The user's browser prepares a POST request and appends the username and password in the body of the request:



```
▾ Hypertext Transfer Protocol
  ▾ POST /login.php HTTP/1.1\r\n
      Request Method: POST
      Request URI: /login.php
      Request Version: HTTP/1.1
    Host: amazon\r\n
    User-Agent: Mozilla/5.0 (Thing)\r\n
    Connection: keep-alive\r\n
    Accept: text/html\r\n
    Content-Type: application/x-www-form-urlencoded\r\n
    Origin: http://amazon\r\n
    Referer: http://amazon/login.php\r\n
  ▸ Content-Length: 20\r\n
    \r\n
    [Full request URI: http://amazon/login.php]
    File Data: 20 bytes
▾ HTML Form URL Encoded: application/x-www-form-urlencoded
  ▾ Form item: "name" = "cs338"
      Key: name
      Value: cs338
  ▾ Form item: "psw" = "cs338"
      Key: psw
      Value: cs338
```

Unlike basic auth, the credentials are in the body (space below the headers), and separated by an '&' (NOTE, this syntax where variable values are separated by '&', and non-alphanumeric characters are percent-encoded, is called `application/x-www-form-urlencoded` and is not the only encoding type out there.)[10]

5. User's browser sends the POST request to the server.
6. The server processes and stores the variables sent by the user. Unlike GET requests, a POST request usually means that the variables will be stored on the server. Moreover, sending the same POST request repeatedly does not guarantee the same effects (the standard term for this characteristic is *non-idempotent*)[11].
7. The server's response depends on whether the credentials are correct.

[10] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST
[11] Ibid.

Service	Description
ssh (**s**ecure **sh**ell)	For executing OS commands on a remote server from your local machine.[12]
http-get	Sending login credentials through a GET request to a basic auth server.
http-post	This concerns HTTP/HTTPS servers that send the user a login form, and expect the user to send a POST request, containing login credentials as form variables.[13]

What is Sprinkler?

As mentioned in the introduction, Sprinkler is a password sprayer written in C, and supports the services ssh, http-get, and http-post as defined above. It also supports extra options such as connecting via TLS, delay between attempts, input regex that determines login success/failure, input login parameter names and values, verbose mode, and help pages.

NOTE: usually, a password-spraying attack is one where the attacker sends the same weak password to multiple accounts on the same server.[14] This is to avoid the lockout that occurs when too many login requests are sent for the same account.[15] However, the current version of Sprinkler proceeds on an account-by-account basis, spraying multiple passwords to the same account and then moving on to the next. This is also how Hydra, an industrial-grade password sprayer, works by default.[16] Another consideration is that, unlike Hydra, Sprinkler does not support synchronous I/O multiplexing, since the socket type is non-blocking.[17] This is a major blow to Sprinkler's execution speed, but Sprinkler compensates for it through other mechanisms (see the sprinkler_recv section).

[12] https://www.cloudflare.com/learning/access-management/what-is-ssh/#:~:text=The%20Secure%20Shell%20(SSH)%20protocol%20is%20a%20method%20for%20securely,and%20encrypt%20connections%20between%20devices.

[13] https://docs.oracle.com/cd/E15217_01/doc.1014/e12488/v2form.htm

[14] https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/password-spraying/?srsltid=AfmBOopcG3Y20wof86JLzbqqP3s27arl30o2YtwQMf2d7OZ8COZMeIwl

[15] Ibid.

[16] https://github.com/vanhauser-thc/thc-hydra

[17] https://beej.us/guide/bgnet/html/index-wide.html#blocking

How Does Sprinkler work?

Roughly, Sprinkler works like this:

1. User inputs options/arguments through command line
2. Sprinkler processes the command line and checks for invalidities
3. For each username, then for each password, send the login credentials to the server

   According to the service specified, Sprinkler…

   a. Prepares login request so it has the right format
   b. Sends login request to server
   c. Receives server's response
   d. Determines whether it's a login success or failure

4. If a login success is detected, stop spraying for the current username and move on to the next one.
5. Prints out all the successful combinations of usernames and passwords

However, there's much more going on behind the scenes. So here's a more detailed explanation of steps 1-3

1. **User input options/arguments through command line**

   The user input the following mandatory options/argument through the command line

   a. Username or username file (**-u username** or **-U FILE**)
   b. Password or password file (**-p password** or **-P FILE**)
   c. Target server's port (**-s PORT**)
   d. Target server's IP address or domain name
   e. Service

   The user can also input extra options such as

   - Set a delay time between logins (**-d DELAY**)
   - Prints out status report and/or all login attempts (**-v** or **-V**)
   - Connect via TLS (**-S**)
   - Input a regex that determines a login success/failure based on whether the regex is found in target's response (**-r REGEX**)
   - Specify form parameter names and values (if applicable) for servers that use form-based authentication (**-i PARAMETERS**)

2. **Sprinkler processes the command line**

   In the sprinkler.c file, Sprinkler parses the command line using C's <unistd> library function **getopt**. Essentially, a single call to **getopt** returns the nearest option character in the command line, starting with the first one. **getopt** also sets the variable *optarg* if there is an argument next to the option. Accordingly, Sprinkler has a while loop where each iteration gets the option, checks what option it is, whether an argument is supplied, and initialises appropriate variables.

   While processing each option/argument, Sprinkler also checks its validity by using **getopt**'s feature: if the user types in an argument to an option that does not expect one, or there's no argument for an option that requires an argument, then **getopt** returns a recognisable marker. If multiple invalidities exist in the command line, Sprinkler detects as many as it can, prints the error messages for each invalidity, and exits.

   The following invalidities are detected during this process:
   - The five mandatory inputs (see above) are not provided
   - Mismatch between actual vs. expected number of arguments
   - Port is not a number or <= 0
   - Username or password files do not exist
   - Delay time is not a number or <= 0
   - Input regex does not start with S= or F=
   - Unknown option provided
   - Service unsupported

3. **Sprinkler prepares and send login request for the service specified, and processes the target server's response**

Once the service is identified, Sprinkler either calls the function from sprinkler-ssh.c or sprinkler-http.c, depending on the service.

This is the step where each service is wildly different: each expects a different login request format and returns different things, so Sprinkler needs to take that into account. Here is a breakdown of what's going on for each service.

SSH

Sprinkler uses Libssh library functions to perform the following tasks:

a) Create a **ssh_session** struct by calling ssh_new(). This struct is for storing necessary information used for connecting to an ssh server.
b) Call ssh_options_set() to add the necessary information to the **ssh_session** object, including target's IP, target's port, and what username to log in as.
c) Connect to the server by calling ssh_connect, which accepts the **ssh_session** from the previous step.
d) Sends the password to the target. This is done by calling the function user_auth_password
e) The return value indicates whether it's a login success or failure.

HTTP-GET

First, in order to send or receive messages from the target server, Sprinkler sets up a network socket.

What is that?

In UNIX systems, in order to send messages to a remote server, there must be a file to write messages to. Once the message is written to the file, the OS will send the file contents to the network card, which passes it on to the router, then from the router to the modem, and so on until it reaches the remote server. But given that there are lots of files on the system, how does the OS know which file is meant for communicating with the remote server? Answer: the OS assigns each file a file descriptor, which is just a number that's associated with an open file.[18] When Sprinkler sends a message to the server, it also passes in a file descriptor to the *send* function, so that the OS knows which file to write the messages to. All of this applies to receiving messages from a remote server as well.

A socket, then, is just an endpoint (or for UNIX systems, a file) where messages intended for the remote server are written to.

How does Sprinkler set up a socket and connect to server?

[18] https://beej.us/guide/bgnet/html/index-wide.html#what-is-a-socket

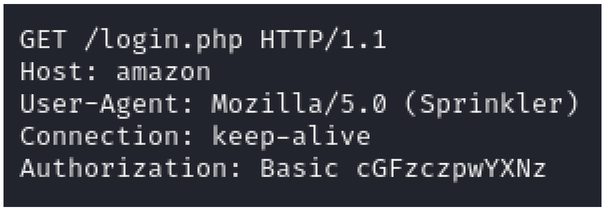
First, sprinkler use C's <netdb> library function *getaddrinfo* to get the necessary information for setting up a socket. This includes what internet protocol the target server accepts, whether stream or datagram protocol is used, what specific protocol it uses, and the IP address. Sprinkler requires that the target server use IPv4 and one of the streaming protocols (TCP is a popular one)[19].

Next, Sprinkler uses C's <sys/socket> library functions *socket* and *connect* to set up a stream socket and connect to the target server respectively.

Formatting login request

Sprinkler assumes that the server uses basic auth authentication scheme and prepares the request buffer:

a) Set up all the required headers in the GET request
b) Concatenates the username and password in to "username:password"
c) Encode the text into base64.
d) Add the encoded text to the "Authorization" header. So far, the request looks like this:



```
GET /login.php HTTP/1.1
Host: amazon
User-Agent: Mozilla/5.0 (Sprinkler)
Connection: keep-alive
Authorization: Basic cGFzczpwYXNz
```

For the "Authorization" header, the 'Basic' refers to basic auth, and the string after is the base64-encoded text

e) Use C's <sys/socket> library function to send the request to server.
f) Receive the server's response (see the section sprinkler_recv for details).
g) Process the server's response. By default, if the server sends a 200 OK, then Sprinkler deems that the login was successful.

HTTP-POST

Like HTTP-GET, Sprinkler also sets up a socket and connects to the server. But the next steps are wildly different.

[19] https://beej.us/guide/bgnet/html/index-wide.html#two-types-of-internet-sockets

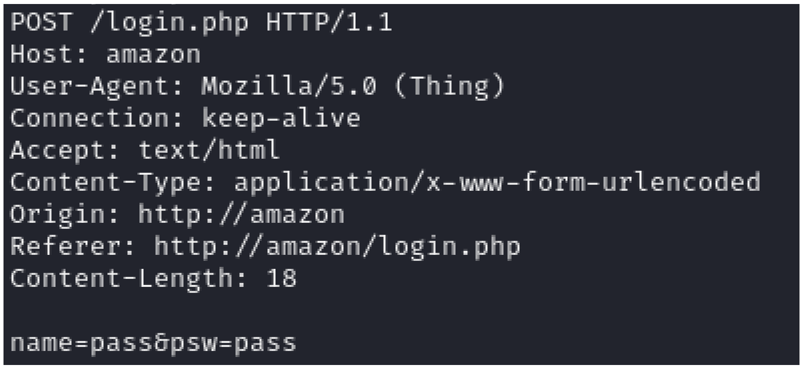
By default:

a) Sprinkler sends a GET request to the server:
b) The server sends a login form
c) Sprinkler parses the login form.
   i) Get the name and value fields for each <input> tag
   ii) For the name field, if the following regex is found in the field…
       "usr|user|email|.*name"
       …then the field is a parameter name that corresponds to the username.
   iii) In contrast, if the following regex is found in the name field…
       "pass|pw|psw|ps"
       Then the field is a parameter name that corresponds to the password.

       (NOTE: for steps i-iii, the regex uses POSIX extended regex and ignores case)

   iv) If there is a value field present, this is neither the username nor password, but Sprinkler appends the name and value to the body of the POST request

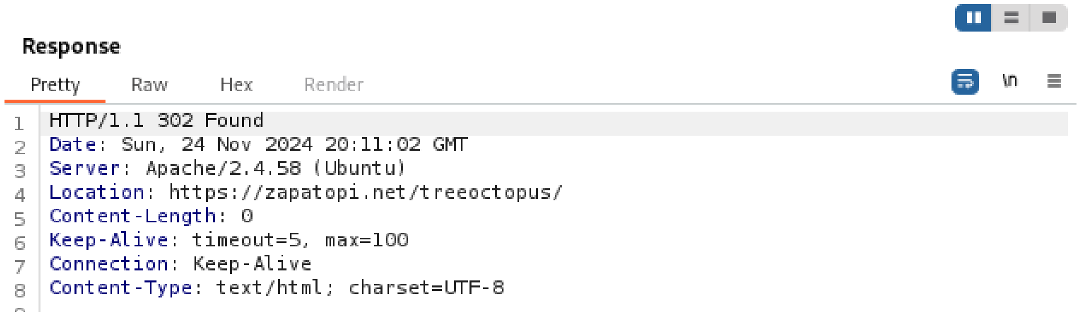
d) The actual username and password are added in the body of the POST request next to their form parameter names.
e) Sprinkler calculates the Content-Length, which is how many bytes does the body of the request contain, and add that to the Content-Length header. At this step, this is what the POST request looks like.



```
POST /login.php HTTP/1.1
Host: amazon
User-Agent: Mozilla/5.0 (Thing)
Connection: keep-alive
Accept: text/html
Content-Type: application/x-www-form-urlencoded
Origin: http://amazon
Referer: http://amazon/login.php
Content-Length: 18

name=pass&psw=pass
```

f) Sprinkler sends the request to the server
g) Receives the server's response (see the sprinkler_recv section for details)
h) Check if the response indicates a redirect to a different page, then Sprinkler deems a login success. A redirect is indicated by the Location header value that's different from the login page. For instance:



```
HTTP/1.1 302 Found
Date: Sun, 24 Nov 2024 20:11:02 GMT
Server: Apache/2.4.58 (Ubuntu)
Location: https://zapatopi.net/treeoctopus/
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

Sprinkler interprets this response as login success

sprinkler_recv - how it works

The sprinkler_recv function is used for receiving the target server's response for http-get and http-post. It utilizes C's <sys/socket> library function *recv*. Since the server can send the response in multiple packets, one call to recv does **not** guarantee that it receives all of the server's response.[20] One approach to ensure that the entire response is received is to make repeated calls to recv until the target server closes the connection. This approach is used by Hydra.[21] However, closing and re-establishing a connection for every login attempt wastes a lot of time. To circumvent this issue, when Sprinkler sends login credentials, it sets the "Connection" header to 'keep-alive', which tells the target server not to close the connection after every login attempt.

As such, sprinkler_recv proceeds by the following steps

1. Accepts three arguments
    a. char *buf - the char array to store server's response
    b. int *bufSize - how many bytes to store in buf
    c. int receiveAll - whether it should receive all of server's response (receiveAll = 1) or receive at most bufSize bytes (receiveAll = 0)
2. Call recv until the marker that separates the header and the body is found (\r\n\r\n).
3. **If** Content-Length header is found, the value is how many bytes the body contains.[22] Use this to find how many bytes have not been received.

[20] https://pubs.opengroup.org/onlinepubs/007904975/functions/recv.html
[21] https://github.com/vanhauser-thc/thc-hydra/blob/master/hydra-http.c
[22] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Length

Else if the value of the Transfer-Encoding header is “chunked,” this means that the Content-Length header is not provided. Instead, the end of the response is indicated by “\r\n0\r\n”[23]

Else, exit the program, since Sprinkler doesn’t know how to handle other forms of Transfer-Encoding.

4. If there are more bytes to receive, then call recv until one of the following is true
   a. “\r\n0\r\n” is found **and** chunked encoding
   b. All Content-Length bytes are received **and** not chunked encoding
   c. receiveAll = 0 **and** the buffer is full
5. If receiveAll = 0 and the buffer is full, then call recv to read the remaining bytes in the socket into a trash array that will be discarded once the function ends. Specifically, call recv until one of these is true
   a. “\r\n0\r\n” is found **and** chunked encoding
   b. All Content-Length bytes are received **and** not chunked encoding

Important remarks

- For all calls to recv, if an error is returned, then close and re-establish the connection, re-send the request buffer, and call recv again. If after multiple re-attempts, recv still returns an error, then exit the program.
- There are multiple reasons why recv might return an error. One possible reason is that the server recognizes that too many requests are coming from the same IP address, and therefore cuts off the connection to prevent password spraying.
- If the TLS option is enabled, then sprinkler_recv calls the <ssl.lib> function *SSL_read_ex* instead of recv.

[23] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding#directives

Extra Options

Sprinkler supports the following global options (options that apply to all services)

-d DELAY

- DELAY how many seconds should Sprinkler wait between each login attempt
- Why use this at all? Some servers block the user if too many login requests are sent in a limited amount of time. The delay option is a way to circumvent this issue.
- Delay reads the amount of time inputted after the option in seconds and waits for it before sending the login request.

-v or -V

- Verbose mode.
- -v prints out status reports, such as when the remote server closes the connection, or when an array needs to be realloc'ed
- -V prints out status reports and the result of all login attempts sent to the server

Sprinkler also provides the following service-specific options

-S

- For http-get and http-post only
- Connect to target server via TLS. Briefly, TLS is a protocol that uses public-key cryptography to exchange symmetric keys and encrypt all communications with those symmetric keys. Moreover, TLS allows the servers to authenticate their identities to each other by providing a digital certificate and proving that the private key is legitimate. The upshot is that the two servers can be confident in each other's identity, and all communications between the two servers are unintelligible to third parties.
- Sprinkler uses OpenSSL's library functions to perform a TLS handshake and send/receive messages from the target server. Sprinkler does not waste time in validating the target's certificate.

-r REGEX

- For http-get and http-post only
- Supply a regex (POSIX extended regex) to determine a login success (S=REGEX) or failure (F=REGEX) if regex pattern is found in target's response.

- Why use this? Sometimes, the server’s response for login success or failure is different from what is expected by default. For example, an HTTP server that uses basic auth needs not return a 200 OK for login successes.
- Examples:

```
sprinkler -u user -p pass -s 80 -r 'F=/login/[^\r\n]*\r\n\r\n' 127.0.0.1 http-get
sprinkler -u user -p pass -s 80 -r 'S=Welcome back!' 127.0.0.1 http-get
```

-i PARAMETERS

- For http-post only
- Supply parameter names and values that go in the request body, using the following syntax:
  'param1=value1&param2=value2&...'
- If a parameter corresponds to the username, set value to ^USER^
  If a parameter corresponds to the password, set value to ^PASS^
- Why use this? Some parameters are interpreted by the target server as usernames or passwords. Sprinkler tries to figure out what those params are, by searching for a regex pattern (see the **HTTP-POST** section above) but in cases where the login form parameters elude the regex, the user should use -i to specify what the parameters for username and password are.
- Examples:

```
sprinkler u user -p pass -s 443 -S -i ‘email=^USER^&passwd=^PASS^’ 127.0.0.1 http-post
sprinkler u user -p pass -s 443 -S -i ‘usr=^USER^&ps=^PASS^&id=whatever’ 127.0.0.1 http-post
```

MakeFile

Upon typing ‘Make’ in the terminal, our MakeFile detects the OS and architecture of the machine and compiles all the source files (sprinkler.c , sprinkler-ssh.c , sprinkler-http.c) into one executable, as well as linking with third-party libraries Libssh, Libssl, and Libcrypto (part of OpenSSL).

To install Sprinkler, the command ‘Make install’ copies the executable and the help page files to /usr/local/bin. This works for MacOS and Linux systems, but note that other OS may use different directories for storing such programs. The user may edit the directory in ‘install’ target, and change how sprinkler.c gets the help pages.

Setting Up The Target Servers

TL;DR

1. Ubuntu installation of Linux on a virtual machine as a target for ssh.
2. Apache2 server on the Ubuntu virtual machine, with the directory protected by basic authentication as a target for http-get.
3. Apache2 server on the Ubuntu virtual machine, with a login.php file requiring a POST request from the client to enter in login information before checking it.

Here's some more details.

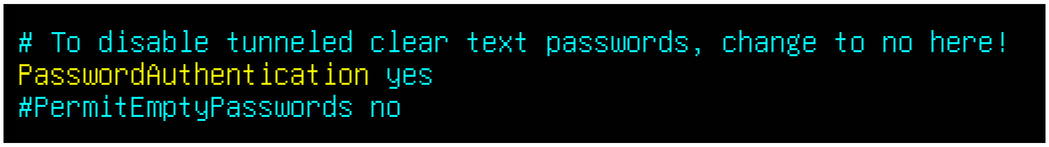
Ubuntu is an installation of Linux commonly used for server hosting and with some of the necessary files already installed. That makes it an ideal choice for a target.

After installation, we first want to set up ssh compatibility. We'll need root permissions to install one thing and configure the settings to get it working. Run

```
sudo apt install ssh
```

Or your equivalent if using a different file manager.

Next we need to configure the login. For the target, it is simple. Again, we'll need root permissions to do this. Open /etc/ssh/sshd_config with editing permissions and find #Password Authentication and get rid of the comment. This will enable you to ssh from your home machine onto the Ubuntu virtual machine using the password you use to login to your Ubuntu user. It should look like so:



```
# To disable tunneled clear text passwords, change to no here!
PasswordAuthentication yes
#PermitEmptyPasswords no
```

Then, start ssh up with

```
sudo systemctl start ssh
```

And your ssh target is up and running! By default, your ssh service will be hosted on port 22. With your ip address for the virtual machine, you can ssh into it.

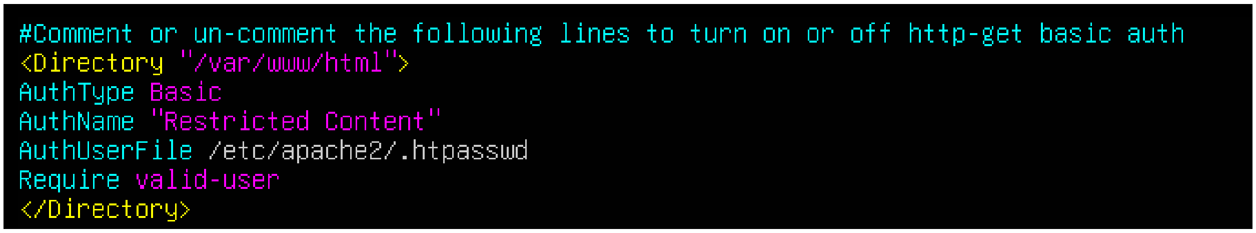
To set up a website for Sprinkler to target, we want to use apache2. It's a hosting service that comes with most of the necessary components pre installed on ubuntu. Let's get the missing ones out of the way first with:

```
sudo apt install apache2
sudo apt install lib apache2-mod-php
```

Then, start it up with

```
sudo systemctl start apache2
```

Now we have a website running, but it isn't protected by anything. To put up basic authentication protections, we'll want to go into the configuration file, by default at /etc/apache2/apache2.conf with root permissions and add the following at the top[24]:



```
#Comment or un-comment the following lines to turn on or off http-get basic auth
<Directory "/var/www/html">
AuthType Basic
AuthName "Restricted Content"
AuthUserFile /etc/apache2/.htpasswd
Require valid-user
</Directory>
```

When those lines are uncommented (# is comment), then basic authentication will be on. This code simply tells the server to restrict the directory's content behind a basic auth login. In this case, the whole website. However, the password file isn't set up yet. To set it up, run the following:

```
sudo htpasswd /etc/apache2/.htpasswd ubuntu
```

This creates a user called ubuntu to the htpasswd file, and creates it if the file doesn't exist. You can change the name if you'd like. You'll be prompted to enter in a corresponding password for that login. We used ubuntu for our target, to keep things simple. To enable our changes so far, run:

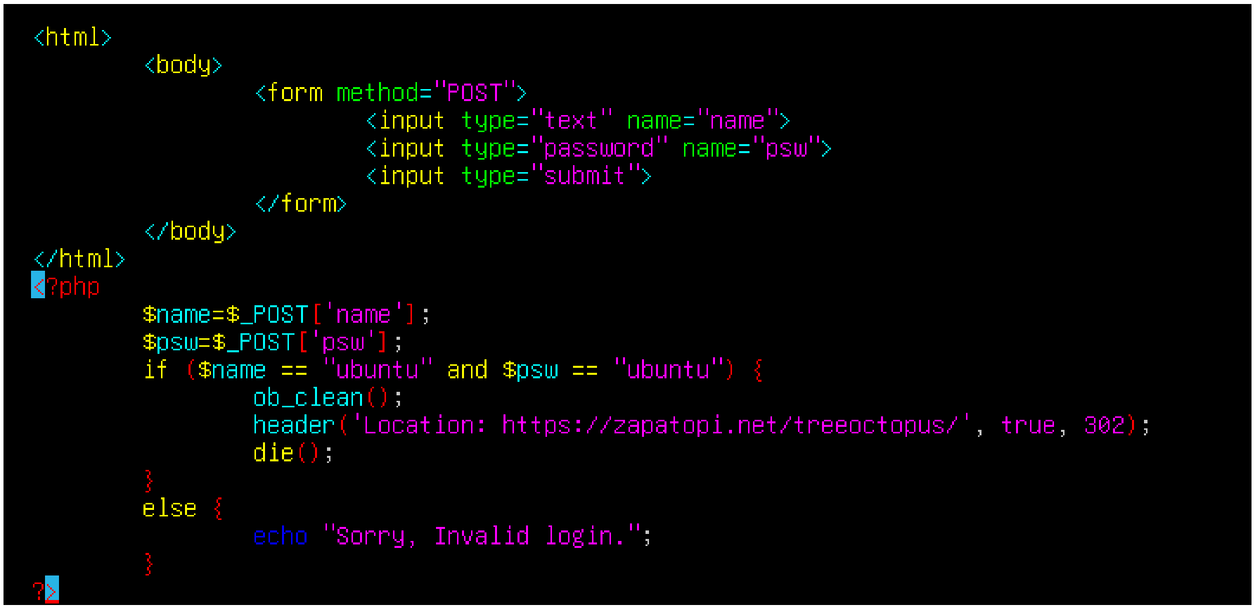
```
sudo systemctl restart apache2
```

[24] https://httpd.apache.org/docs/2.4/mod/mod_auth_basic.html

Now if you open up the website you are hosting, you'll find it protected by basic authentication and asking you to login. To try this, you need to find the ip address of your virtual machine; one way to find your ip address on Ubuntu is:

```
ip a
```

Now that basic auth is setup, we can test our http-get service. For http-post, we'll need to add a file to our server. Create a .php file in /var/www/html/ with the following abomination:



```
<html>
        <body>
                <form method="POST">
                        <input type="text" name="name">
                        <input type="password" name="psw">
                        <input type="submit">
                </form>
        </body>
</html>
<?php
        $name=$_POST['name'];
        $psw=$_POST['psw'];
        if ($name == "ubuntu" and $psw == "ubuntu") {
                ob_clean();
                header('Location: https://zapatopi.net/treeoctopus/', true, 302);
                die();
        }
        else {
                echo "Sorry, Invalid login.";
        }
?>
```

We called ours login.php. The html at the top displays a login form, and tells the user to send their login information via a post request. The php code beneath it interprets the login and if valid redirects the user to https://zapatopi.net/treeoctopus/, a choice to make sure we knew when we had a valid login, which can be replaced with any link you want. Redirecting a successful login to elsewhere in the site is typical of a valid login. Otherwise, it apologizes and gives the "Sorry, Invalid login." message. This is hilariously vulnerable code that doesn't even really work as an authentication page. It is, however, an amazingly fast *login* page. It very quickly differentiates between accepted usernames and passwords, and takes action on that difference. It does not provide authentication to other parts of the site like a normal one authentication page would.

To switch to http-post authentication, turn off the basic authentication. So go back to /etc/apache2/apache2.conf with permissions to edit it, and put a # before each of those lines we added. Then, restart apache2 with:

```
sudo systemctl restart apache2
```

Http-post is now set up as well! With all of our targets set up, you can practice attacking them as you desire. You can edit the .php page however you like, and it will automatically be updated from the user's experience. This is because the server reads and operates the .php file every time a user visits. Editing the apache2.conf file, however, requires restarting the server. This is because it changes the rules of how the server engages with the user.

References

Cited in this document

‘Base64 - MDN Web Docs Glossary: Definitions of Web-Related Terms | MDN’, 4 November 2024. https://developer.mozilla.org/en-US/docs/Glossary/Base64.

‘Beej’s Guide to Network Programming’. Accessed 25 November 2024. https://beej.us/guide/bgnet/html/index-wide.html.

cloudflare. ‘What Is SSH? | Secure Shell (SSH) Protocol’, n.d. https://www.cloudflare.com/learning/access-management/what-is-ssh/.

Federal Trade Commission. ‘Identity Theft and Assumption Deterrence Act’, 12 August 2013. https://www.ftc.gov/legal-library/browse/rules/identity-theft-assumption-deterrence-act-text.

Fielding, Roy T., and Julian Reschke. ‘Hypertext Transfer Protocol (HTTP/1.1): Authentication’. Request for Comments. Internet Engineering Task Force, June 2014. https://datatracker.ietf.org/doc/rfc7235/.

‘Form-Based Authentication’. Accessed 25 November 2024. https://docs.oracle.com/cd/E15217_01/doc.1014/e12488/v2form.htm.

Hauser Heuse, Marc van. ‘Hydra’. C, March 2021. https://github.com/vanhauser-thc/thc-hydra.

Lonvick, Chris M., and Tatu Ylonen. ‘The Secure Shell (SSH) Transport Layer Protocol’. Request for Comments. Internet Engineering Task Force, January 2006. https://datatracker.ietf.org/doc/rfc4253/.

‘Mod_auth_basic - Apache HTTP Server Version 2.4’. Accessed 25 November 2024. https://httpd.apache.org/docs/2.4/mod/mod_auth_basic.html.

‘POST - HTTP | MDN’, 8 October 2024. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST.

‘Recv’. Accessed 25 November 2024. https://pubs.opengroup.org/onlinepubs/007904975/functions/recv.html.

‘WWW-Authenticate - HTTP | MDN’, 5 November 2024. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/WWW-Authenticate.

All resources essential for this project's completion

Hydra

Hauser Heuse, Marc van. 'Hydra'. C, March 2021. https://github.com/vanhauser-thc/thc-hydra.

Setting up target servers

'Committer SSH Access - Apache Infrastructure Website'. Accessed 25 November 2024. https://infra.apache.org/user-ssh.html.

Ubuntu Server. 'How to Install and Configure PHP'. Accessed 25 November 2024. https://documentation.ubuntu.com/server/how-to/web-services/install-php/.

Ubuntu Server. 'How to Install Apache2'. Accessed 25 November 2024. https://documentation.ubuntu.com/server/how-to/web-services/install-apache2/.

Accessed 25 November 2024. https://ubuntu.com/download/server#release-notes-lts.

Command line processing

'Getopt (The GNU C Library)'. Accessed 25 November 2024. https://www.gnu.org/software/libc/manual/html_node/Getopt.html.

'Such Programming - Command Line Arguments in C'. Accessed 25 November 2024. https://suchprogramming.com/command-line-c/.

SSH

'Libssh – The SSH Library!' Accessed 25 November 2024. https://www.libssh.org/.

'Libssh: The Tutorial'. Accessed 25 November 2024. https://api.libssh.org/stable/libssh_tutorial.html.

Socket programming

'Beej's Guide to Network Programming'. Accessed 25 November 2024. https://beej.us/guide/bgnet/html/index-wide.html.

'Recv'. Accessed 25 November 2024. https://pubs.opengroup.org/onlinepubs/007904975/functions/recv.html.

TLS

'Authentication Test'. Accessed 25 November 2024. https://authenticationtest.com/.

Library. 'Library'. Accessed 25 November 2024. https://openssl-library.org/.

'Ossl-Guide-Tls-Client-Block - OpenSSL Documentation'. Accessed 25 November 2024. https://docs.openssl.org/3.2/man7/ossl-guide-tls-client-block/.

HTTP formatting

'Content-Length - HTTP | MDN', 30 October 2024. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Length.

'GET - HTTP | MDN', 12 September 2024. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET.

'POST - HTTP | MDN', 8 October 2024. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST.

'Regular Expressions (The GNU C Library)'. Accessed 25 November 2024. https://www.gnu.org/software/libc/manual/html_node/Regular-Expressions.html.

ryyst. 'Answer to "How Do I Base64 Encode (Decode) in C?"' Stack Overflow, 21 July 2011. https://stackoverflow.com/a/6782480.

'Transfer-Encoding - HTTP | MDN', 25 September 2024. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding.

'WWW-Authenticate - HTTP | MDN', 5 November 2024. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/WWW-Authenticate.

MakeFile

Makefile Tutorial. 'Makefile Tutorial by Example'. Accessed 25 November 2024. https://makefiletutorial.com.

'Phony Targets (GNU Make)'. Accessed 25 November 2024. https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html.

Robinson, Trevor. 'Answer to "OS Detecting Makefile"'. Stack Overflow, 23 August 2012. https://stackoverflow.com/a/12099167.