# Week 3 Penetration Testing Report

**Full Name: Chirag Suthar**
**Program: HCPT**
**Date: 03-03-2025**

## Introduction

This report hereby describes the proceedings and results of a Black Box security assessment conducted against **Week 3 Labs**. The report hereby lists the findings and corresponding best practice mitigation actions and recommendations.

## 1. Objective

The objective of the assessment was to uncover vulnerabilities in the **Week 3 Labs** and provide a final security assessment report comprising vulnerabilities, remediation strategy, and recommendation guidelines to help mitigate the identified vulnerabilities and risks during the activity.

## 2. Scope

This section defines the scope and boundaries of the project.

| | |
|---|---|
| **Application Name** | **Cross Origin Resource Sharing, Cross-Site Request Forgery** |

## 3. Summary

Outlined is a Black Box Application Security assessment for **Week 3 Labs**.

**Total number of Sub-labs: 16 Sub-labs**

| High | Medium | Low |
|---|---|---|
| 5 | 4 | 4 |

**High** - **Number of Sub-labs with hard difficulty level**

**Medium** - **Number of Sub-labs with medium difficulty level,**

**Low** - **Number of Sub-labs with Easy difficulty level**

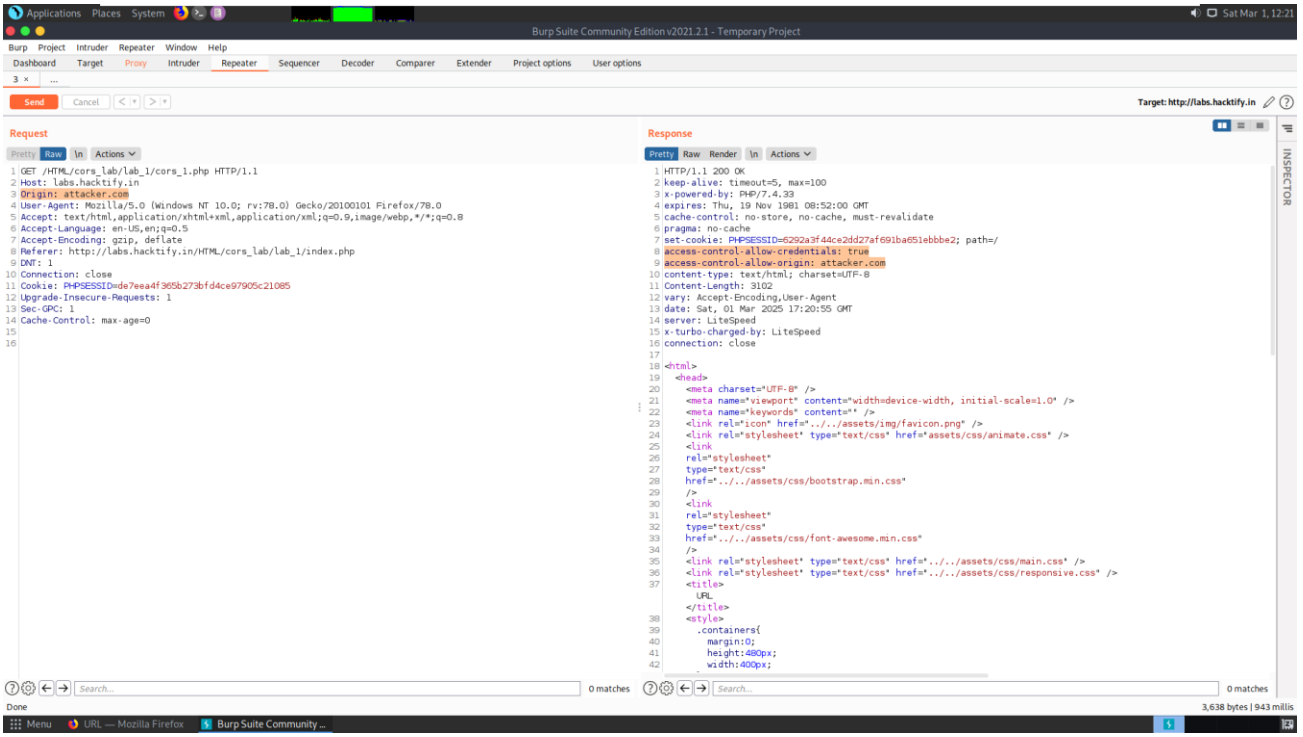
# 1. Cross Origin Resource Sharing (CORS)

## 1.1. CORS With Arbitrary Origin

| Reference | Risk Rating |
|---|---|
| CORS With Arbitrary Origin | Low |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy. However, it also provides potential for cross-domain based attacks, if a website's CORS policy is poorly configured and implemented. The CORS protocol uses some HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. Many modern websites use CORS to allow access from subdomains and trusted third parties. Sometimes because of mistakes of developers' attackers can use misconfiguration to exploit vulnerability. | |
| **How It Was Discovered** | |
| I found this vulnerability by manually writing Origin payload by intercepting it in the Burp Suite. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/cors_lab/lab_1/index.php | |
| **Consequences of not Fixing the Issue** | |
| Attackers would treat many victims to visit the attacker's website, if victim is logged in, then his personal information is recorded on attacker's server. Attacker can perform any action in the user's account, bypassing CSRF tokes | |
| **Suggested Countermeasures** | |
| 1. **Proper configuration of cross-domain requests** : If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header.<br>2. **Only allow trusted sites** : Dynamically reflecting origins from cross-domain requests without validation is readily exploitable and should be voided.<br>3. **Avoid whitelisting null** : Avoid using the header Access-Control-Allow-Origin: null . Cross-domain resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.<br>4. **Avoid wildcards in internal networks** : Avoid using wildcards in internal networks. Trusting network configuration alone to protect internal resources is not sufficient when internal browsers can access untrusted external domains. | |
| **References** | |
| 1. https://portswigger.net/web-security/cors<br>2. https://owasp.org/www-community/attacks/CORS_OriginHeaderScrutiny<br>3. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS | |
| **Payload Used** | |
| origin:attacker.com | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.

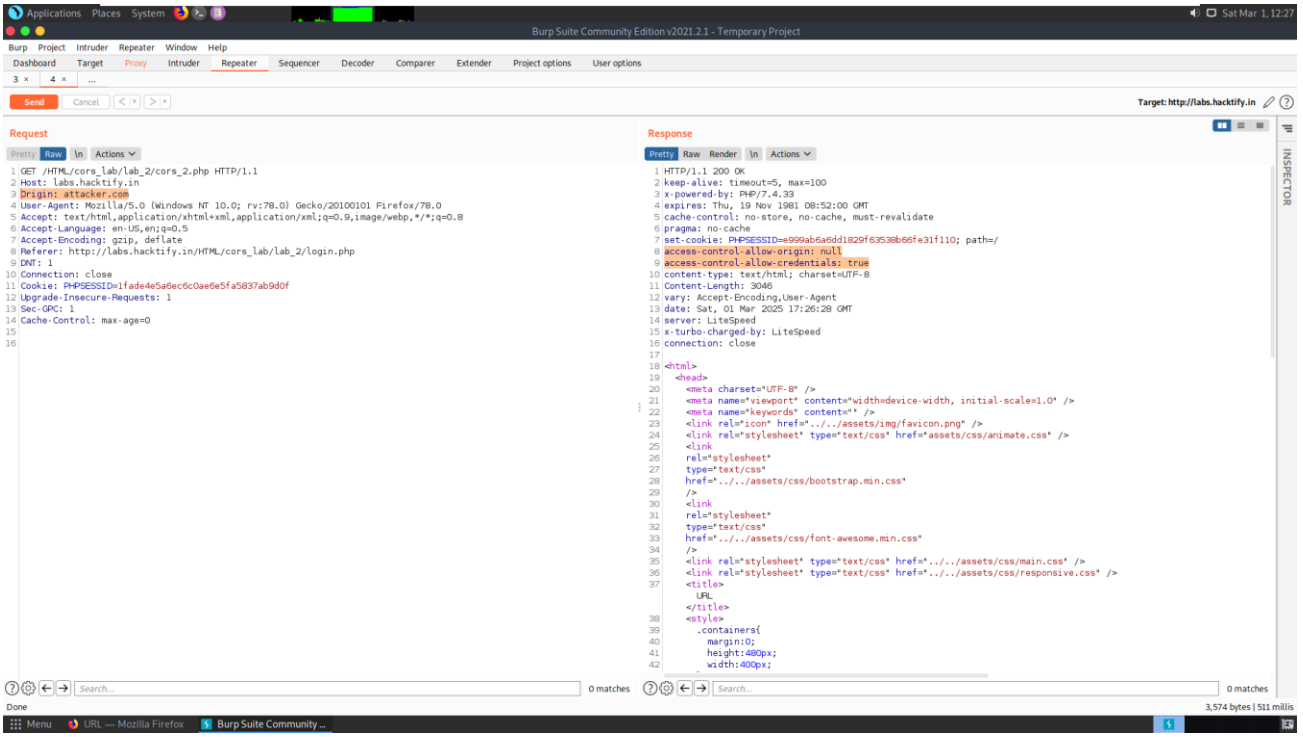


## 1.2. CORS with Null Origin

| Reference | Risk Rating |
|---|---|
| CORS With Null Origin | Low |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy. However, it also provides potential for cross-domain based attacks, if a website's CORS policy is poorly configured and implemented. The CORS protocol uses some HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. Many modern websites use CORS to allow access from subdomains and trusted third parties. Sometimes because of mistakes of developers' attackers can use misconfiguration to exploit vulnerability. | |
| **How It Was Discovered** | |
| I found this vulnerability by manually writing Origin payload by intercepting it in the Burp Suite. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/cors_lab/lab_2/index.php | |
| **Consequences of not Fixing the Issue** | |
| Attackers would treat many victims to visit the attacker's website, if victim is logged in, then his personal information is recorded on attacker's server. Attacker can perform any action in the user's account, bypassing CSRF tokes | |
| **Suggested Countermeasures** | |
| 1. **Proper configuration of cross-domain requests** : If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header.<br>2. **Only allow trusted sites** : Dynamically reflecting origins from cross-domain requests without validation is readily exploitable and should be voided.<br>3. **Avoid whitelisting null** : Avoid using the header Access-Control-Allow-Origin: null . Cross-domain resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.<br>4. **Avoid wildcards in internal networks** : Avoid using wildcards in internal networks. Trusting network configuration alone to protect internal resources is not sufficient when internal browsers can access untrusted external domains. | |
| **References** | |
| 1. https://portswigger.net/web-security/cors<br>2. https://owasp.org/www-community/attacks/CORS_OriginHeaderScrutiny<br>3. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS | |
| **Payload Used** | |
| origin:attacker.com | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.

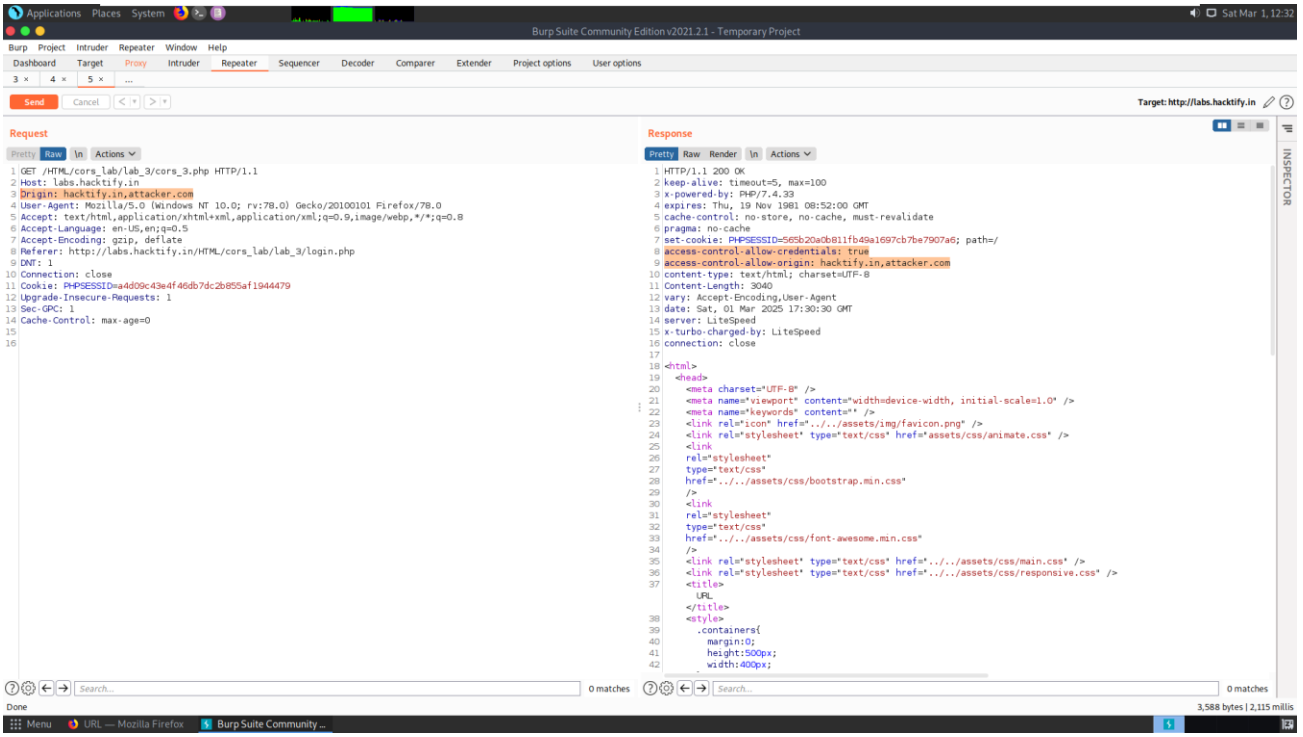


## 1.3. CORS with prefix match

| Reference | Risk Rating |
|---|---|
| CORS With prefix match | Medium |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy. However, it also provides potential for cross-domain based attacks, if a website's CORS policy is poorly configured and implemented. The CORS protocol uses some HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. Many modern websites use CORS to allow access from subdomains and trusted third parties. Sometimes because of mistakes of developers' attackers can use misconfiguration to exploit vulnerability. | |
| **How It Was Discovered** | |
| I found this vulnerability by manually writing Origin payload by intercepting it in the Burp Suite. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/cors_lab/lab_3/index.php | |
| **Consequences of not Fixing the Issue** | |
| Attackers would treat many victims to visit the attacker's website, if victim is logged in, then his personal information is recorded on attacker's server. Attacker can perform any action in the user's account, bypassing CSRF tokes | |
| **Suggested Countermeasures** | |
| 1. **Proper configuration of cross-domain requests** : If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header.<br>2. **Only allow trusted sites** : Dynamically reflecting origins from cross-domain requests without validation is readily exploitable and should be voided.<br>3. **Avoid whitelisting null** : Avoid using the header Access-Control-Allow-Origin: null . Cross-domain resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.<br>4. **Avoid wildcards in internal networks** : Avoid using wildcards in internal networks. Trusting network configuration alone to protect internal resources is not sufficient when internal browsers can access untrusted external domains. | |
| **References** | |
| 1. https://portswigger.net/web-security/cors<br>2. https://owasp.org/www-community/attacks/CORS_OriginHeaderScrutiny<br>3. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS | |
| **Payload Used** | |
| origin:hacktify.in,attacker.com | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.

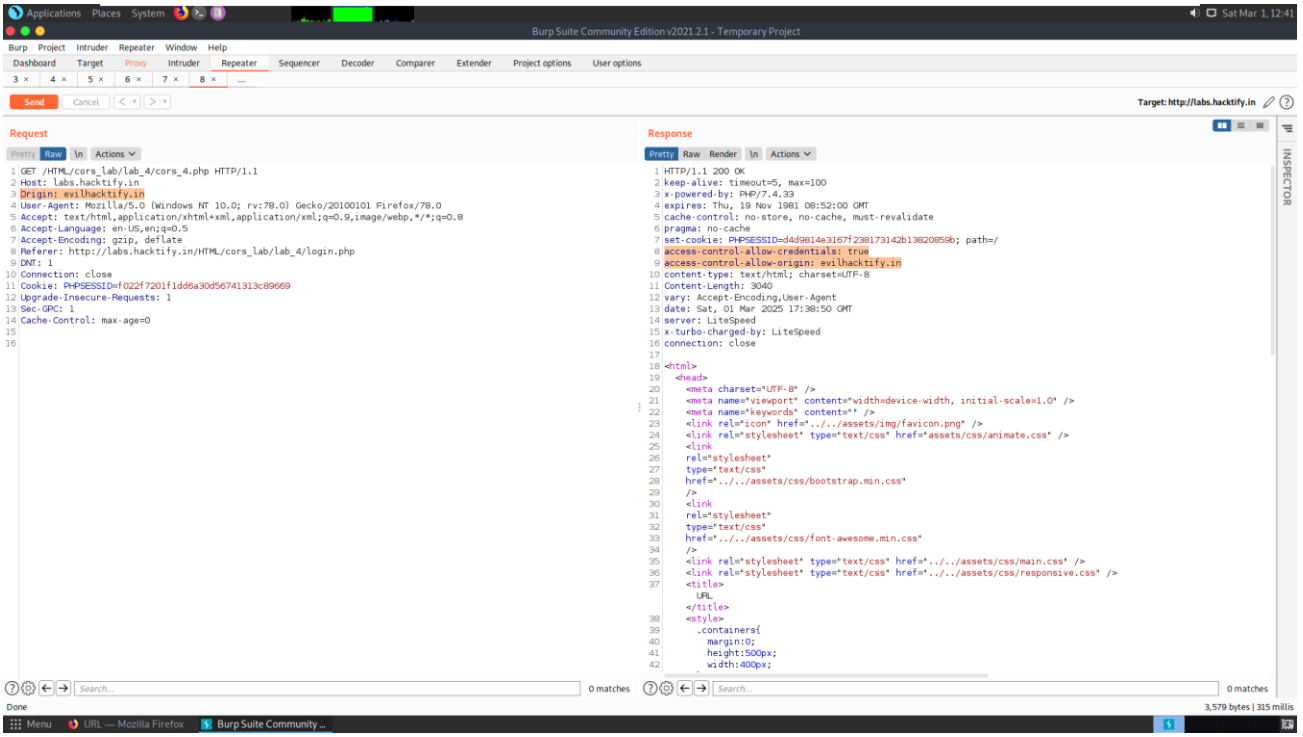


## 1.4. CORS with suffix match

| Reference | Risk Rating |
|---|---|
| CORS With suffix match | Medium |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy. However, it also provides potential for cross-domain based attacks, if a website's CORS policy is poorly configured and implemented. The CORS protocol uses some HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. Many modern websites use CORS to allow access from subdomains and trusted third parties. Sometimes because of mistakes of developers' attackers can use misconfiguration to exploit vulnerability. | |
| **How It Was Discovered** | |
| I found this vulnerability by manually writing Origin payload by intercepting it in the Burp Suite. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/cors_lab/lab_4/index.php | |
| **Consequences of not Fixing the Issue** | |
| Attackers would treat many victims to visit the attacker's website, if victim is logged in, then his personal information is recorded on attacker's server. Attacker can perform any action in the user's account, bypassing CSRF tokes | |
| **Suggested Countermeasures** | |
| 1. **Proper configuration of cross-domain requests** : If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header.<br>2. **Only allow trusted sites** : Dynamically reflecting origins from cross-domain requests without validation is readily exploitable and should be voided.<br>3. **Avoid whitelisting null** : Avoid using the header Access-Control-Allow-Origin: null . Cross-domain resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.<br>4. **Avoid wildcards in internal networks** : Avoid using wildcards in internal networks. Trusting network configuration alone to protect internal resources is not sufficient when internal browsers can access untrusted external domains. | |
| **References** | |
| 1. https://portswigger.net/web-security/cors<br>2. https://owasp.org/www-community/attacks/CORS_OriginHeaderScrutiny<br>3. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS | |
| **Payload Used** | |
| origin:evilhacktify.com | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.

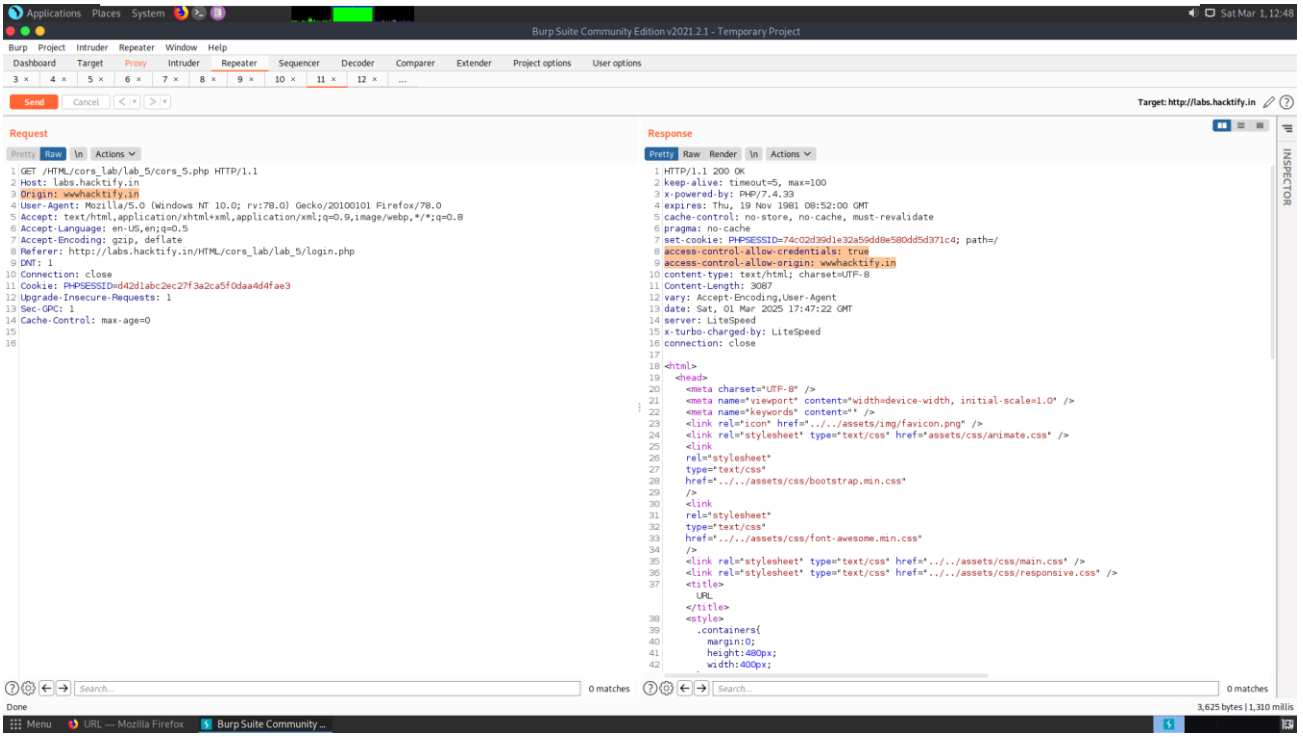


## 1.5. CORS with Escape dot

| Reference | Risk Rating |
|---|---|
| CORS with Escape dot | High |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy. However, it also provides potential for cross-domain based attacks, if a website's CORS policy is poorly configured and implemented. The CORS protocol uses some HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. Many modern websites use CORS to allow access from subdomains and trusted third parties. Sometimes because of mistakes of developers' attackers can use misconfiguration to exploit vulnerability. | |
| **How It Was Discovered** | |
| I found this vulnerability by manually writing Origin payload by intercepting it in the Burp Suite. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/cors_lab/lab_5/index.php | |
| **Consequences of not Fixing the Issue** | |
| Attackers would treat many victims to visit the attacker's website, if victim is logged in, then his personal information is recorded on attacker's server. Attacker can perform any action in the user's account, bypassing CSRF tokes | |
| **Suggested Countermeasures** | |
| 1. **Proper configuration of cross-domain requests** : If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header.<br>2. **Only allow trusted sites** : Dynamically reflecting origins from cross-domain requests without validation is readily exploitable and should be voided.<br>3. **Avoid whitelisting null** : Avoid using the header Access-Control-Allow-Origin: null . Cross-domain resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.<br>4. **Avoid wildcards in internal networks** : Avoid using wildcards in internal networks. Trusting network configuration alone to protect internal resources is not sufficient when internal browsers can access untrusted external domains. | |
| **References** | |
| 1. https://portswigger.net/web-security/cors<br>2. https://owasp.org/www-community/attacks/CORS_OriginHeaderScrutiny<br>3. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS | |
| **Payload Used** | |
| origin:wwwhacktify.in | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.

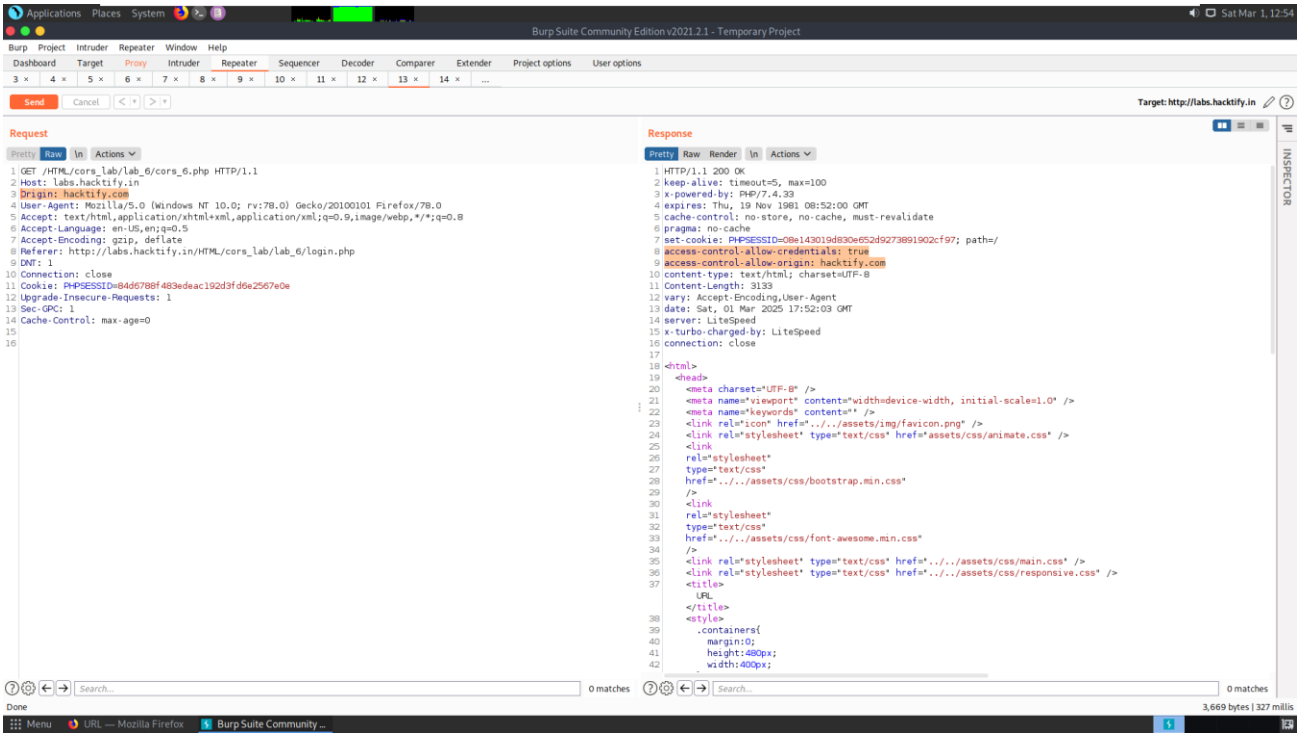


## 1.6. CORS with Substring match

| Reference | Risk Rating |
|---|---|
| CORS with Substring match | High |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy. However, it also provides potential for cross-domain based attacks, if a website's CORS policy is poorly configured and implemented. The CORS protocol uses some HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. Many modern websites use CORS to allow access from subdomains and trusted third parties. Sometimes because of mistakes of developers' attackers can use misconfiguration to exploit vulnerability. | |
| **How It Was Discovered** | |
| I found this vulnerability by manually writing Origin payload by intercepting it in the Burp Suite. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/cors_lab/lab_6/index.php | |
| **Consequences of not Fixing the Issue** | |
| Attackers would treat many victims to visit the attacker's website, if victim is logged in, then his personal information is recorded on attacker's server. Attacker can perform any action in the user's account, bypassing CSRF tokes | |
| **Suggested Countermeasures** | |
| 1. **Proper configuration of cross-domain requests** : If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header.<br>2. **Only allow trusted sites** : Dynamically reflecting origins from cross-domain requests without validation is readily exploitable and should be voided.<br>3. **Avoid whitelisting null** : Avoid using the header Access-Control-Allow-Origin: null . Cross-domain resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.<br>4. **Avoid wildcards in internal networks** : Avoid using wildcards in internal networks. Trusting network configuration alone to protect internal resources is not sufficient when internal browsers can access untrusted external domains. | |
| **References** | |
| 1. https://portswigger.net/web-security/cors<br>2. https://owasp.org/www-community/attacks/CORS_OriginHeaderScrutiny<br>3. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS | |
| **Payload Used** | |
| origin:hacktify.com | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.

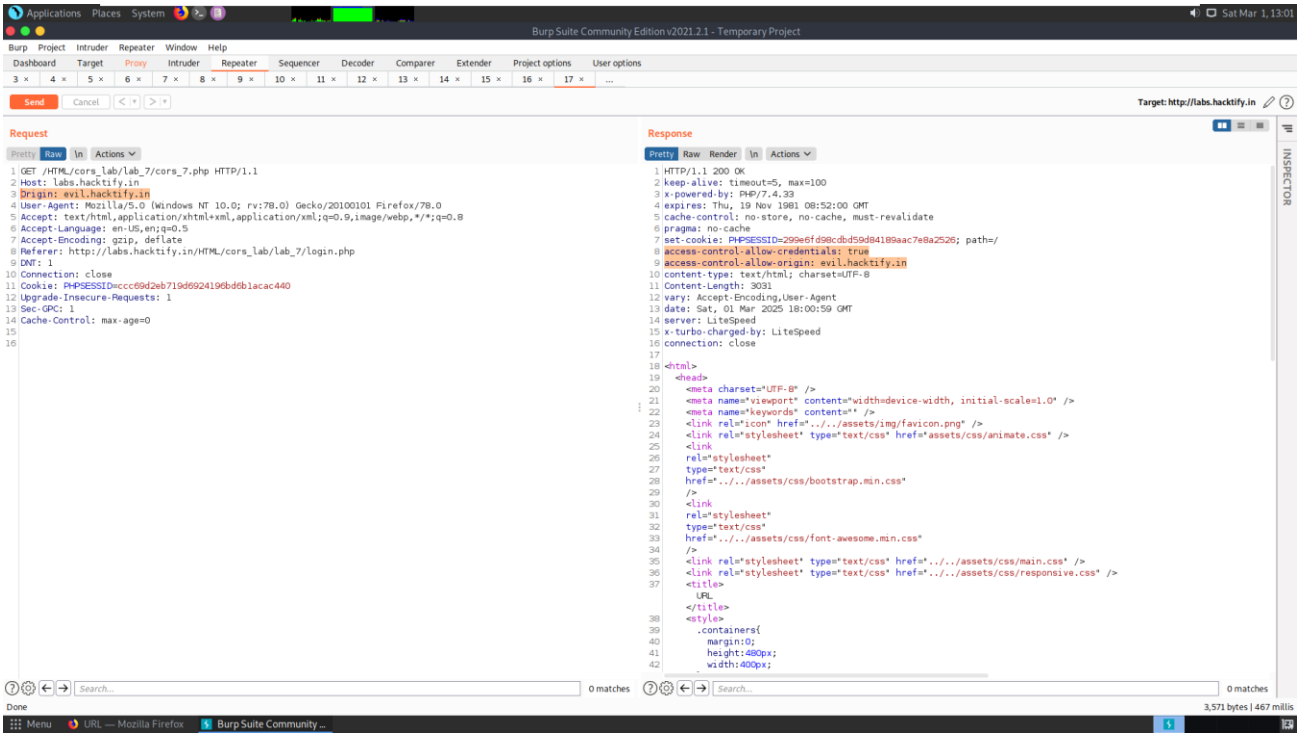


## 1.7. CORS with Arbitrary Subdomain

| Reference | Risk Rating |
|---|---|
| CORS with Arbitrary Subdomain | High |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy. However, it also provides potential for cross-domain based attacks, if a website's CORS policy is poorly configured and implemented. The CORS protocol uses some HTTP headers that define trusted web origins and associated properties such as whether authenticated access is permitted. Many modern websites use CORS to allow access from subdomains and trusted third parties. Sometimes because of mistakes of developers' attackers can use misconfiguration to exploit vulnerability. | |
| **How It Was Discovered** | |
| I found this vulnerability by manually writing Origin payload by intercepting it in the Burp Suite. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/cors_lab/lab_7/index.php | |
| **Consequences of not Fixing the Issue** | |
| Attackers would treat many victims to visit the attacker's website, if victim is logged in, then his personal information is recorded on attacker's server. Attacker can perform any action in the user's account, bypassing CSRF tokes | |
| **Suggested Countermeasures** | |
| 1. **Proper configuration of cross-domain requests** : If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header.<br>2. **Only allow trusted sites** : Dynamically reflecting origins from cross-domain requests without validation is readily exploitable and should be voided.<br>3. **Avoid whitelisting null** : Avoid using the header Access-Control-Allow-Origin: null . Cross-domain resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.<br>4. **Avoid wildcards in internal networks** : Avoid using wildcards in internal networks. Trusting network configuration alone to protect internal resources is not sufficient when internal browsers can access untrusted external domains. | |
| **References** | |
| 1. https://portswigger.net/web-security/cors<br>2. https://owasp.org/www-community/attacks/CORS_OriginHeaderScrutiny<br>3. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS | |
| **Payload Used** | |
| origin:evil.hacktify.in | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.



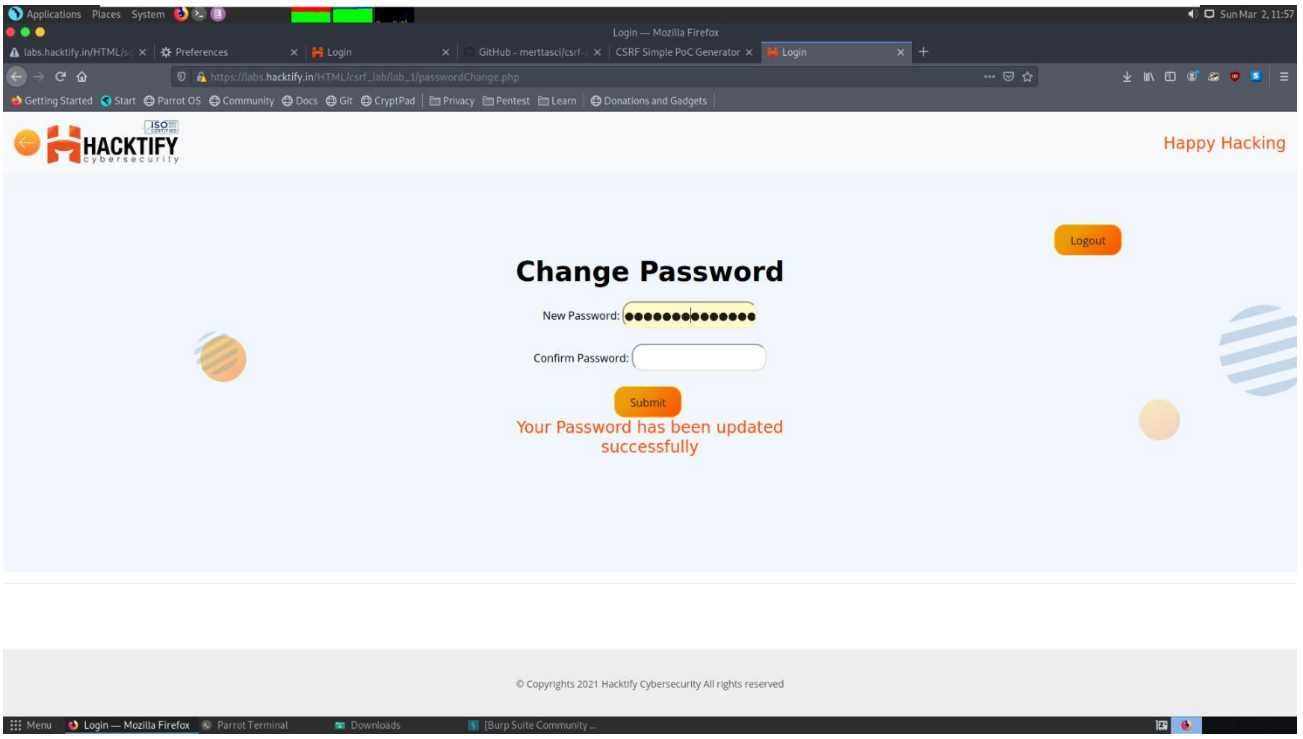
## 2. Cross-Site Request Forgery (CSRF)

### 2.1. Eassyy CSRF

| Reference | Risk Rating |
|---|---|
| Eassyy CSRF | Low |
| **Tools Used** | |
| Google chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application. | |
| **How It Was Discovered** | |
| I found this vulnerability by generating CSRF POCs of forged URL and sending it to the victim. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/csrf_lab/lab_1/passwordChange.php | |
| **Consequences of not Fixing the Issue** | |
| In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. Depending on the nature of the action, the attacker might be able to gain full control over the user's account. If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality | |
| **Suggested Countermeasures** | |
| 1. **Anti-CSRF Tokens**: Use a token that is associated with a particular user and can be found as a hidden value in every state changing form which is present on the web application. This token, called a CSRF Token or a Synchronizer Token.<br>2. **Same Site Cookies**: CSRF attacks are only possible since Cookies are always sent with any requests that are sent to a particular origin, which is related to that Cookie. Due to the nature of a CSRF attack, a flag can be set against a Cookie, turning it into a same-site Cookie. A same-site Cookie is a Cookie which can only be sent, if the request is being made from the same origin that is related to the Cookie being sent. | |
| **References** | |
| 1. https://portswigger.net/web-security/csrf<br>2. https://owasp.org/www-community/attacks/csrf<br>3. https://www.acunetix.com/websitesecurity/csrf-attacks/ | |
| **Payload Used** | |
| Generated CSRF POC using Burp Suite and changed the password to "abcd" followed by sending the URL to victim user. Logged in into the system as victim user, followed by hitting submit button from it. | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.

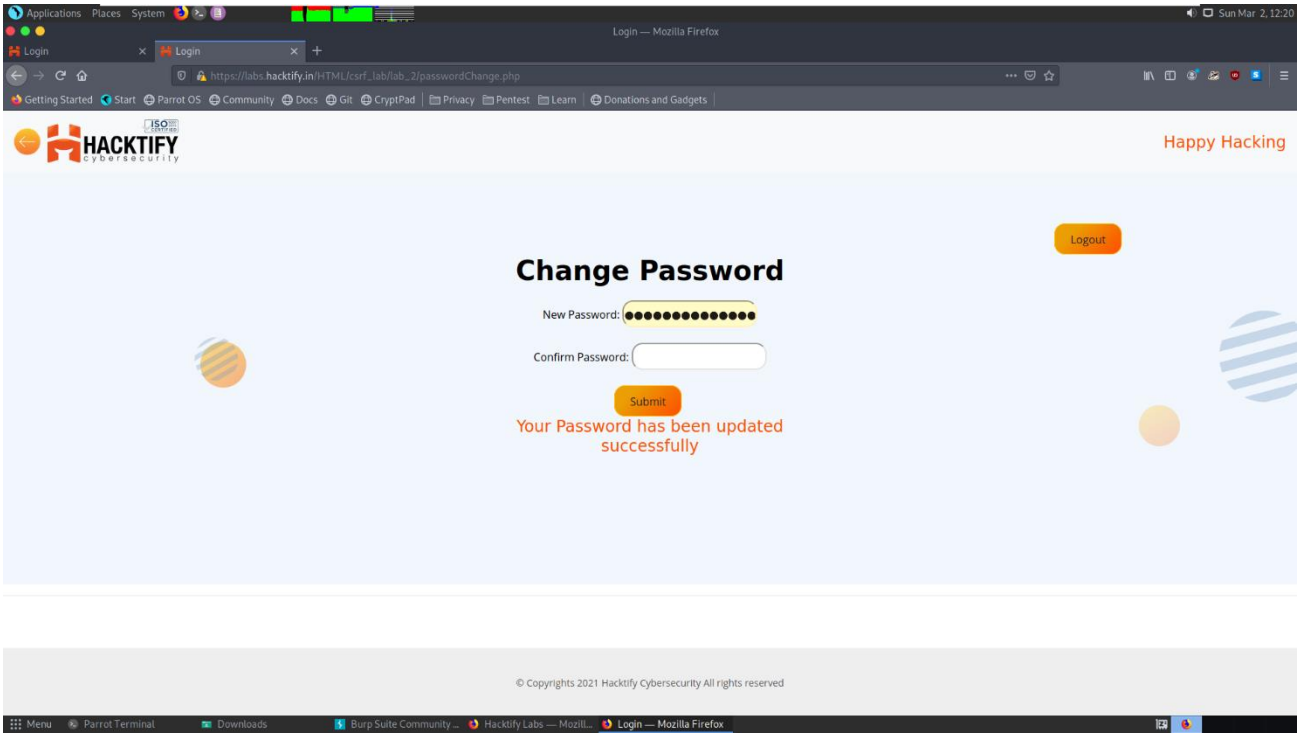


## 2.2. Always Validate Tokens

| Reference | Risk Rating |
|---|---|
| Always Validate Tokens | Medium |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application. | |
| **How It Was Discovered** | |
| I found this vulnerability by generating CSRF POCs of forged URL and sending it to the victim. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/csrf_lab/lab_2/passwordChange.php | |
| **Consequences of not Fixing the Issue** | |
| In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. Depending on the nature of the action, the attacker might be able to gain full control over the user's account. If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality | |
| **Suggested Countermeasures** | |
| 1. **Anti-CSRF Tokens**: Use a token that is associated with a particular user and can be found as a hidden value in every state changing form which is present on the web application. This token, called a CSRF Token or a Synchronizer Token.<br>2. **Same Site Cookies**: CSRF attacks are only possible since Cookies are always sent with any requests that are sent to a particular origin, which is related to that Cookie. Due to the nature of a CSRF attack, a flag can be set against a Cookie, turning it into a same-site Cookie. A same-site Cookie is a Cookie which can only be sent, if the request is being made from the same origin that is related to the Cookie being sent. | |
| **References** | |
| 1. https://portswigger.net/web-security/csrf<br>2. https://owasp.org/www-community/attacks/csrf<br>3. https://www.acunetix.com/websitesecurity/csrf-attacks/ | |
| **Payload Used** | |
| Generated CSRF POC using Burp Suite and changed the password to "abcd" followed by sending the URL to victim user. Logged in into the system as victim user, followed by hitting submit button from it. | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.

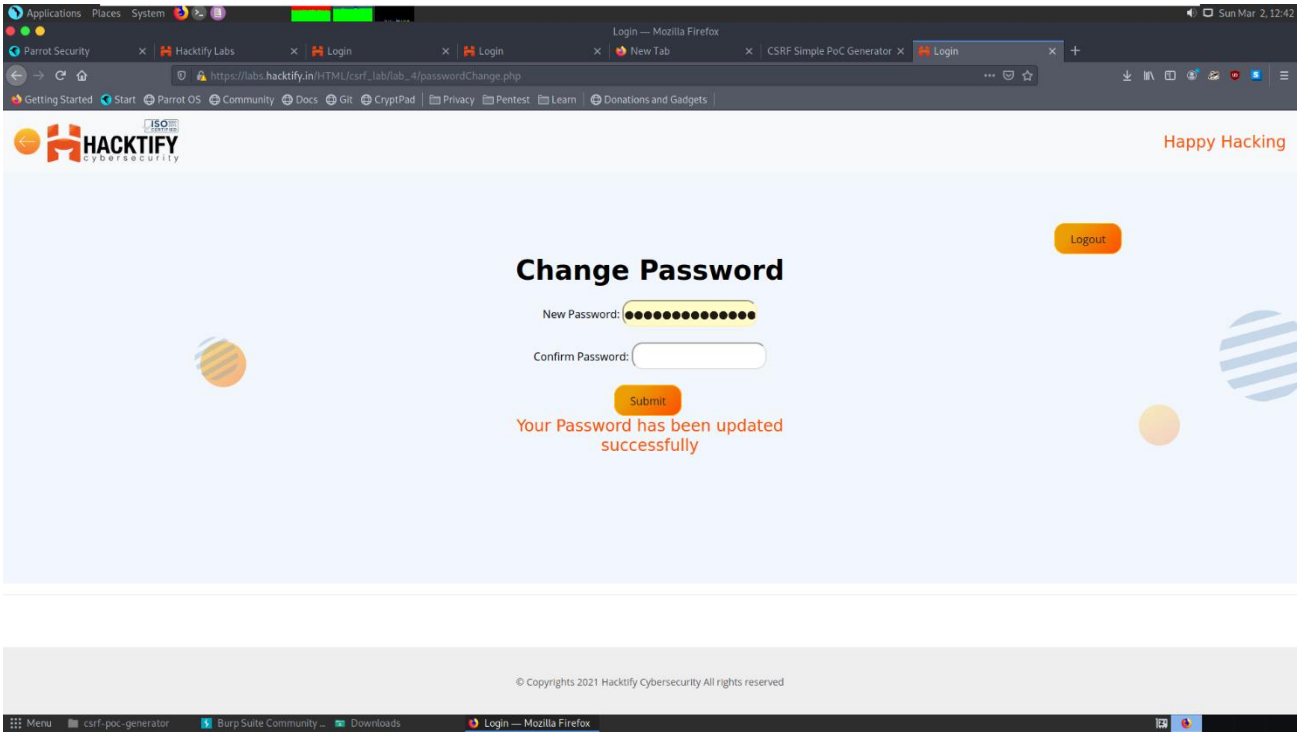


## 2.3. I hate when someone uses my tokens!

| Reference | Risk Rating |
|---|---|
| I hate when someone uses my tokens! | Medium |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application. | |
| **How It Was Discovered** | |
| I found this vulnerability by generating CSRF POCs of forged URL and sending it to the victim. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/csrf_lab/lab_4/passwordChange.php | |
| **Consequences of not Fixing the Issue** | |
| In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. Depending on the nature of the action, the attacker might be able to gain full control over the user's account. If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality | |
| **Suggested Countermeasures** | |
| 1. **Anti-CSRF Tokens**: Use a token that is associated with a particular user and can be found as a hidden value in every state changing form which is present on the web application. This token, called a CSRF Token or a Synchronizer Token.<br>2. **Same Site Cookies**: CSRF attacks are only possible since Cookies are always sent with any requests that are sent to a particular origin, which is related to that Cookie. Due to the nature of a CSRF attack, a flag can be set against a Cookie, turning it into a same-site Cookie. A same-site Cookie is a Cookie which can only be sent, if the request is being made from the same origin that is related to the Cookie being sent. | |
| **References** | |
| 1. https://portswigger.net/web-security/csrf<br>2. https://owasp.org/www-community/attacks/csrf<br>3. https://www.acunetix.com/websitesecurity/csrf-attacks/ | |
| **Payload Used** | |
| Generated CSRF POC using Burp Suite and changed the password to "abcd" followed by sending the URL to victim user. Logged in into the system as victim user, followed by hitting submit button from it. | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.

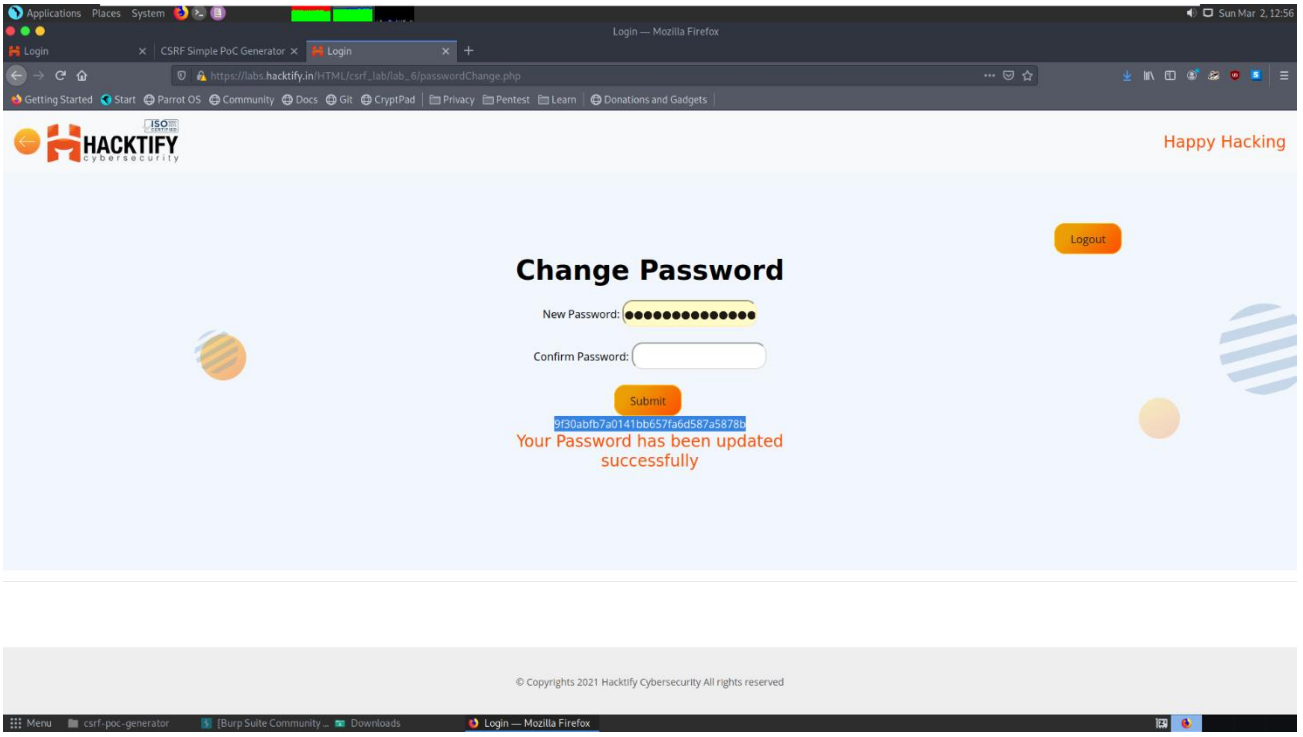


## 2.4. GET Me or POST ME

| Reference | Risk Rating |
|---|---|
| GET Me or POST ME | Low |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application. | |
| **How It Was Discovered** | |
| I found this vulnerability by generating CSRF POCs of forged URL and sending it to the victim. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/csrf_lab/lab_6/passwordChange.php | |
| **Consequences of not Fixing the Issue** | |
| In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. Depending on the nature of the action, the attacker might be able to gain full control over the user's account. If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality | |
| **Suggested Countermeasures** | |
| 1. **Anti-CSRF Tokens**: Use a token that is associated with a particular user and can be found as a hidden value in every state changing form which is present on the web application. This token, called a CSRF Token or a Synchronizer Token.<br>2. **Same Site Cookies**: CSRF attacks are only possible since Cookies are always sent with any requests that are sent to a particular origin, which is related to that Cookie. Due to the nature of a CSRF attack, a flag can be set against a Cookie, turning it into a same-site Cookie. A same-site Cookie is a Cookie which can only be sent, if the request is being made from the same origin that is related to the Cookie being sent. | |
| **References** | |
| 1. https://portswigger.net/web-security/csrf<br>2. https://owasp.org/www-community/attacks/csrf<br>3. https://www.acunetix.com/websitesecurity/csrf-attacks/ | |
| **Payload Used** | |
| Generated CSRF POC using Burp Suite and changed the password to "abcd" followed by sending the URL to victim user. Logged in into the system as victim user, followed by hitting submit button from it. | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.

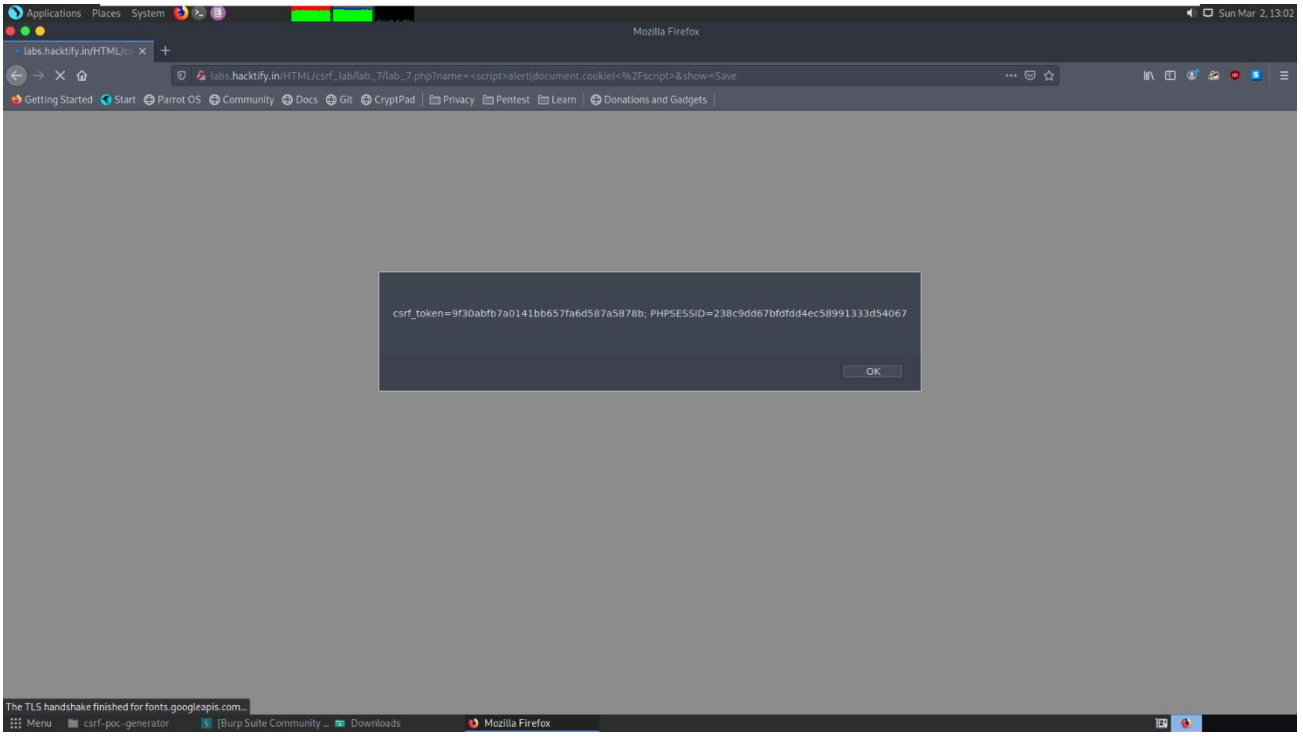


## 2.5. XSS the saviour

| Reference | Risk Rating |
|---|---|
| XSS the saviour | Low |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application. | |
| **How It Was Discovered** | |
| I found this vulnerability by generating CSRF POCs of forged URL and sending it to the victim. | |
| **Vulnerable URLs** | |
| http://labs.hacktify.in/HTML/csrf_lab/lab_7/lab_7.php?name=%3Cscript%3Ealert%28document.cookie%29%3C%2Fscript%3E&show=Save | |
| **Consequences of not Fixing the Issue** | |
| In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. Depending on the nature of the action, the attacker might be able to gain full control over the user's account. If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality | |
| **Suggested Countermeasures** | |
| 1. **Anti-CSRF Tokens**: Use a token that is associated with a particular user and can be found as a hidden value in every state changing form which is present on the web application. This token, called a CSRF Token or a Synchronizer Token.<br>2. **Same Site Cookies**: CSRF attacks are only possible since Cookies are always sent with any requests that are sent to a particular origin, which is related to that Cookie. Due to the nature of a CSRF attack, a flag can be set against a Cookie, turning it into a same-site Cookie. A same-site Cookie is a Cookie which can only be sent, if the request is being made from the same origin that is related to the Cookie being sent. | |
| **References** | |
| 1. https://portswigger.net/web-security/csrf<br>2. https://owasp.org/www-community/attacks/csrf<br>3. https://www.acunetix.com/websitesecurity/csrf-attacks/ | |
| **Payload Used** | |
| <script>alert(document.cookie)</script> | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.



## 2.6. rm -rf token

| Reference | Risk Rating |
|---|---|
| rm -rf token | Low |
| **Tools Used** | |
| Google Chrome and Burp Suite | |
| **Vulnerability Description** | |
| Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application. | |
| **How It Was Discovered** | |
| I found this vulnerability by generating CSRF POCs of forged URL and sending it to the victim. | |
| **Vulnerable URLs** | |
| https://labs.hacktify.in/HTML/csrf_lab/lab_8/passwordChange.php | |
| **Consequences of not Fixing the Issue** | |
| In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. Depending on the nature of the action, the attacker might be able to gain full control over the user's account. If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality | |
| **Suggested Countermeasures** | |
| 1. **Anti-CSRF Tokens**: Use a token that is associated with a particular user and can be found as a hidden value in every state changing form which is present on the web application. This token, called a CSRF Token or a Synchronizer Token.<br>2. **Same Site Cookies**: CSRF attacks are only possible since Cookies are always sent with any requests that are sent to a particular origin, which is related to that Cookie. Due to the nature of a CSRF attack, a flag can be set against a Cookie, turning it into a same-site Cookie. A same-site Cookie is a Cookie which can only be sent, if the request is being made from the same origin that is related to the Cookie being sent. | |
| **References** | |
| 1. https://portswigger.net/web-security/csrf<br>2. https://owasp.org/www-community/attacks/csrf<br>3. https://www.acunetix.com/websitesecurity/csrf-attacks/ | |
| **Payload Used** | |
| Generated CSRF POC using Burp Suite and changed the password to "abcd" followed by sending the URL to victim user. Logged in into the system as victim user, followed by hitting submit button from it. | |

## Proof of Concept

This section contains proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.