

# DETAILED GUIDE ON

`https://site.com/bgin?redirect=`

← **SECURE SITE**

`https://site.com/redirect?url=`

**PHISHING ZONE** →

**OPEN REDIRECT**

[www.hackingarticles.in](http://www.hackingarticles.in)





## Contents

Introduction .....	3
What is the URL?.....	3
Introduction to Open Redirect .....	3
Open redirection Impact.....	4
Open Redirection Exploitation .....	5
Basic Redirection.....	5
Encoded Redirection .....	7
URL Encoded Redirection.....	7
Double URL Encoded Redirection .....	9
Base64 Redirection .....	13
URL Redirection with Hash values .....	15
Redirection with Hash values using Salt .....	17
Redirection over inside a Web-Page .....	21
DOM-based Open Redirect .....	23
So, what is the sink?.....	23
Mitigation Steps .....	26





## Introduction

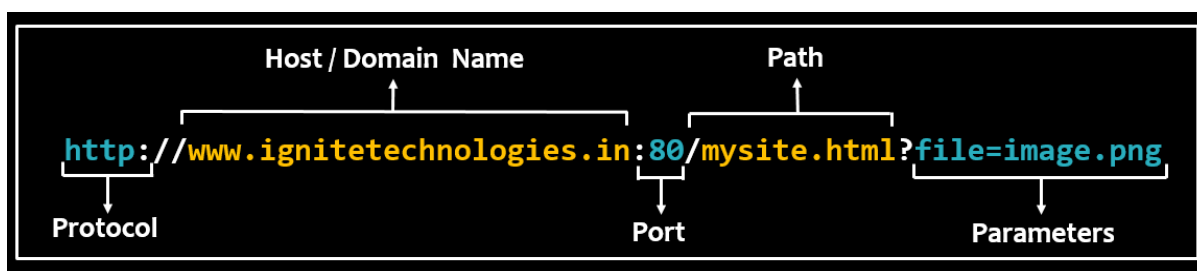
**URLs** are commonly referred to as a **web address**, which **determines the exact location of a web resource over the internet**. But what if this URL gets redirected and takes you to the place where you never expected to? Today, in this article, we'll take a tour on **Open Redirection** and will learn how an **attacker** can **deface a website** by simply **redirecting its URL to a malicious one**.

## What is the URL?

A URL is an abbreviation to **Uniform Resource Locator**, which is nothing but an **address to a unique resource or a file over the Internet**. These resources can be anything, an HTML page, an image, or a CSS document.

A URL is composed of different segments – including a **protocol**, a **domain name**, or a **path** – which **instructs the web browser about how to fetch a resource** and thus are even managed by the webserver. However, it is the **responsibility of the developer to validate the resources and its associated URLs and even the user-input parameters**.

A basic URL is structured in a way like:



## Introduction to Open Redirect

Have you ever noticed about the **response codes** that the web-application offer as **"301" or "302"**, they simply speak out about the **URL redirection!**

Many developers set up their web-applications in order to **request resources over from the web pages** or to **send their visitors to some different location**, that **reside within or outside the web-interface**. To do so, they implement some basic functions such as **header()** in **PHP**, **redirect()** in **Python**, **Response.Redirect()** in **C#** and many others.

But developers often overlook this URL Redirection, as they sometimes do not properly validate or filter the functions they use, or even let users enter their desired input, which thus could lead to one of the most common vulnerabilities i.e. "Open Redirect".

**"Open Redirect" or "Unvalidated Redirection"** is possible when a web application **accepts untrusted input** that could **cause the web application to redirect the request to a URL contained within untrusted input**. By **modifying untrusted URL input to a malicious site**, an **attacker may successfully launch a phishing attack and steal user credentials**.

Source - [OWASP](#)

Didn't understood well, let's check out the following scenario:

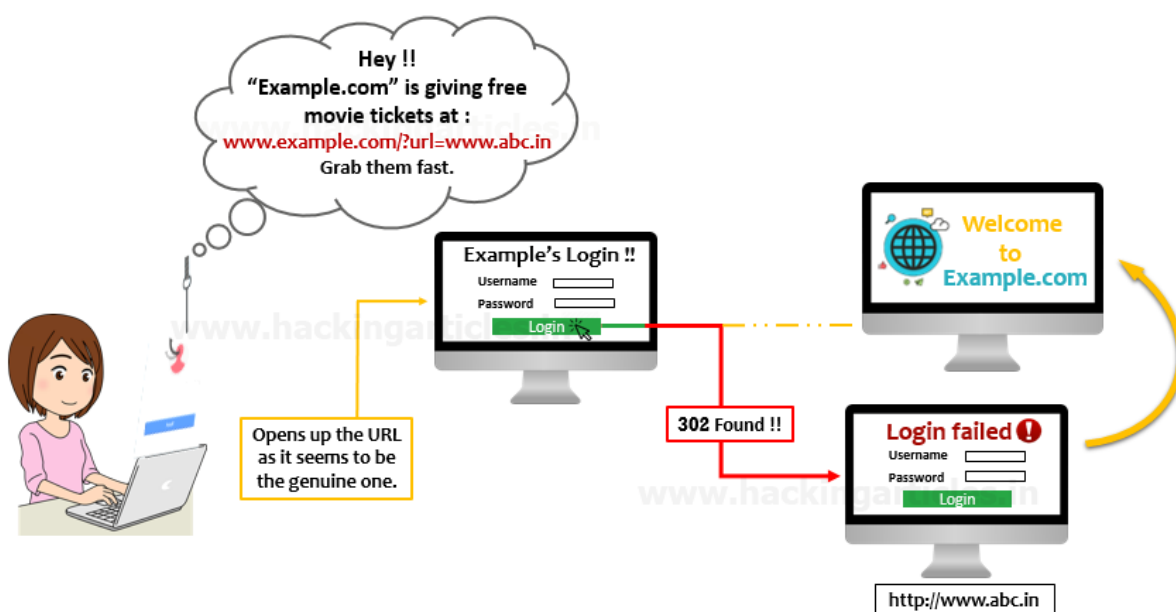


The user gets a phishing email stating that “**Example.com – A movie booking web-app**” is giving its users “**a free movie ticket**” over the URL specified in the email as:

“ **www.example.com/?url=www.abc.in** ”

The URL seems to be the genuine one, as it is having the domain name of “**example.com**”, but the same URL is thus having a redirecting over to “**abc.in**” which is nothing but the attacker’s fake web application.

Now, when the user opens up the URL and enters her credentials over the “**example’s login portal**”, and as she clicks on the **login button**, she thus gets redirected to a different login page rather than example’s home page, as the **Login button** over **example.com** was suffering from “**Open Redirect Vulnerability**”. Therefore, now when she enters up her credentials again, they got compromised and thus then she’ll get redirected to the **home page**.



## Open redirection Impact

Open Redirection is itself a minor vulnerability, but it thus itself can cause major damage to the web-application when integrated with others as with “**RCE**” or “**XSS**”.

Therefore, it thus has been reported with “**Medium Severity**” with a **CVSS score** of “**6.1**” under:

- **CWE-601**: URL Redirection to Untrusted Site ('Open Redirect')

So, in order to exploit this all, we’ve developed some PHP codes as the similar way the developer creates them to enable URL redirections into their applications, further we’ve even used a bWAPP-a vulnerable web-application and the PortSwigger lab.

**Let’s Start!!**

## Open Redirection Exploitation

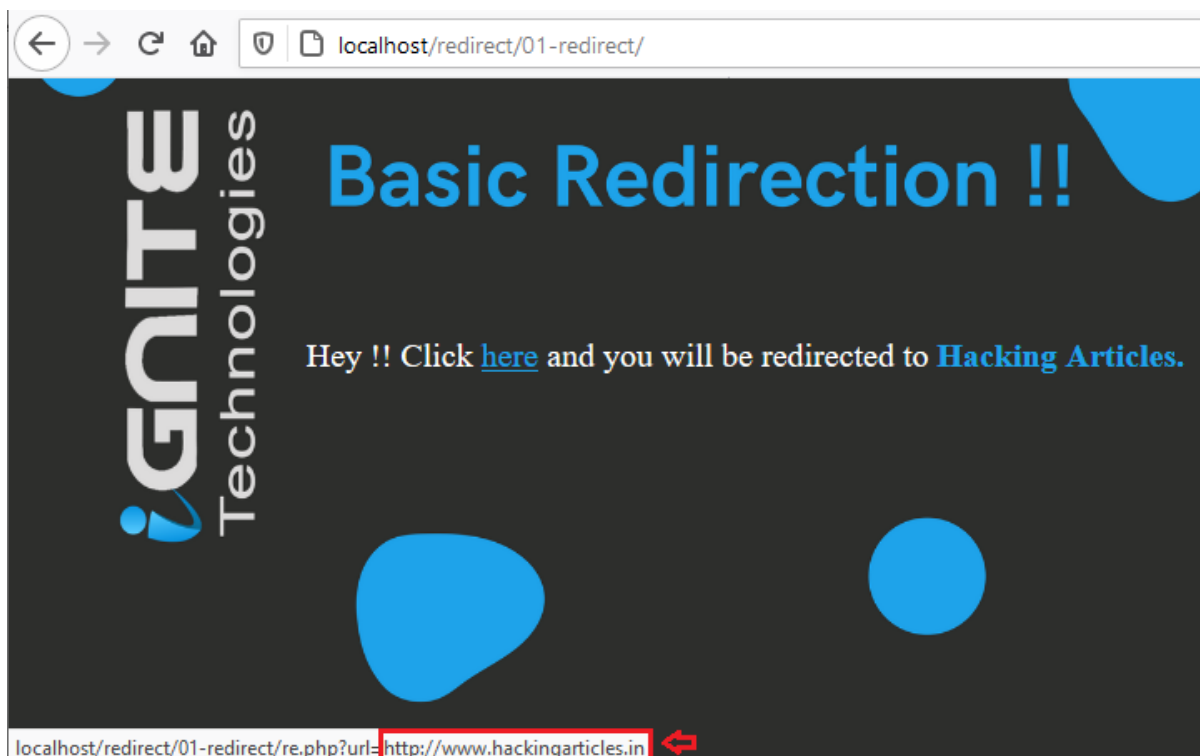
### Basic Redirection

There are chances when the developer **does not care** about the **input validations** or **filtrations** or anything specific and simply **implements** the **redirection functions** as **header()** and let the **redirected URL** be in the **clear texts**.

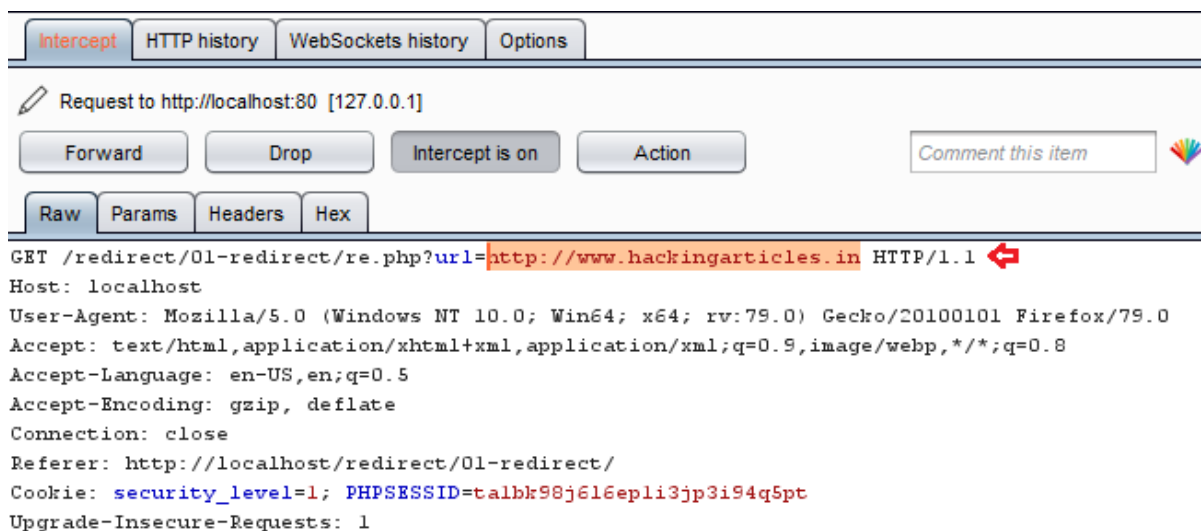
```
<html>
<?php
header('Location:'.($_GET["url"]));
die();

?>
</html>
```

From the below image you can see that the **redirected URL** over at the **“here”** text, is simply reflected as in the cleartext, which means that if we click over on it, we'll be redirected to **“hackingarticles.in”**.



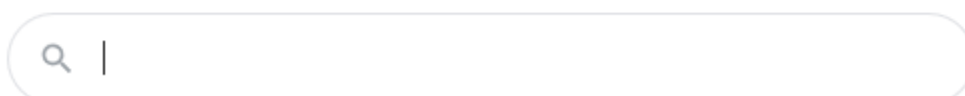
So, let's try to capture this all in our burpsuite and check what we could manipulate over on.



From the above image, you can see that, in the “url=” parameter **http://www.hackingarticles.in** is travelling, let’s try to manipulate this simple clear text with “**http://www.google.com**”.



Simple!! From the below image you can see that, we’ve been redirected to “**google.com**” with a basic manipulation.





## Encoded Redirection

In order to secure up the redirection process, sometimes the developers encode up the “url=” values and let the URL travel over on the internet as in with the redirection. The major encoding methods that a developer can implement are **URL-encoding or the base-64 encoding**.

*But, as we are aware that encoding is a 1-way technique, thus this security can simply be breached if we know about what encoding methodology the developer used into all this.*

### URL Encoded Redirection

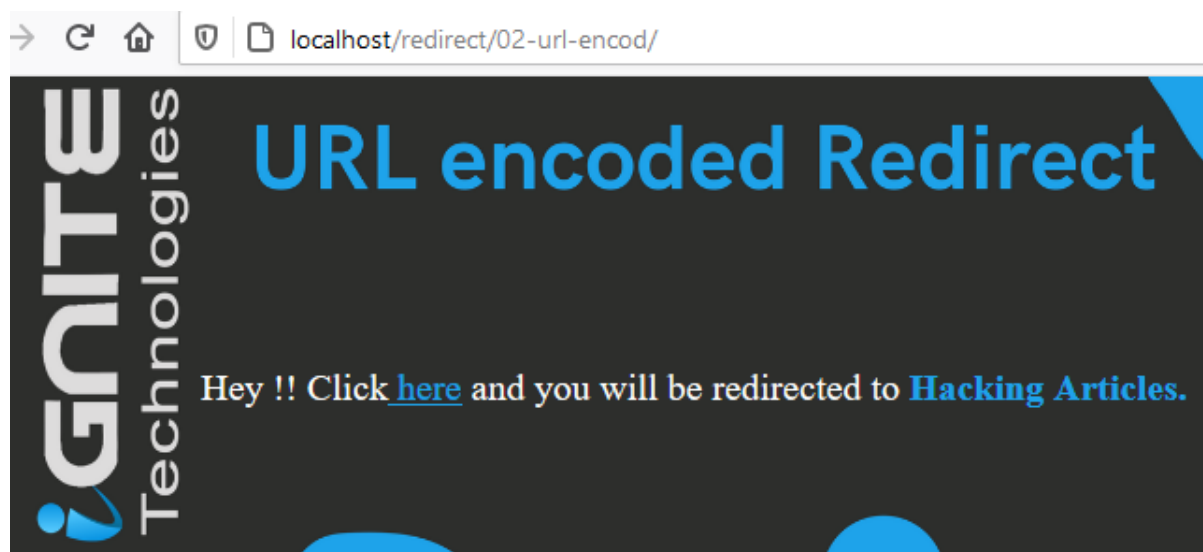
URL Encoding is one of the most common encoding methodologies that the developer use in order to **disallow some vulnerable characters** such as “<” or “>”, to get embedded into the URL.

*The following PHP code snippet shows up a **basic URL decoding redirection**, which first decodes up the encoded input URL and then redirects it over to its desired destination.*

```
<html>
<?php
header('Location:'.urldecode($_GET["url"]));
die();

?>
</html>
```

Let's try to bypass this encoding with some simple tricks.



From the following image, you can see that, as when I **captured up** the **ongoing HTTP Request** over of the “here” text, I was presented with the “url=” parameter containing the **URL encoded redirection link**.



The screenshot shows a web proxy interface with tabs for Intercept, HTTP history, WebSockets history, and Options. The Intercept tab is active, showing a request to http://localhost:80 [127.0.0.1]. Below the request details, there are buttons for Forward, Drop, Intercept is on, and Action. A 'Comment this item' button is also present. The request details are shown in a 'Raw' tab, displaying the following text:

```
GET /redirect/02-url-encod/re.php?url=http%3A%2F%2Fwww.hackingarticles.in HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Gecko/20100101 Firefox/79.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer: http://localhost/redirect/02-url-encod/
```

By analyzing the captured request, it makes us clear that the “url=” parameter is having a simple **basic URL encoded value**. Though, let’s now try to encode our URL i.e.

“https://www.ignitetechnologies.in” over from the **encode tab** in our burpsuite.

The screenshot shows the Burp Suite interface with the 'Encode' tab selected. The input field contains the URL https://www.ignitetechnologies.in. The output field shows the encoded URL: %68%74%74%70%73%3a%2f%2f%77%77%77%77%2e%69%67%6e%69%74%65%74%65%63%68%6e%6f%6e%6f%6c%6f%67%69%65%73%2e%69%6e. The 'Encode as ...' dropdown menu is open, showing options: Plain, URL (selected), HTML, Base64, ASCII hex, and Hex.

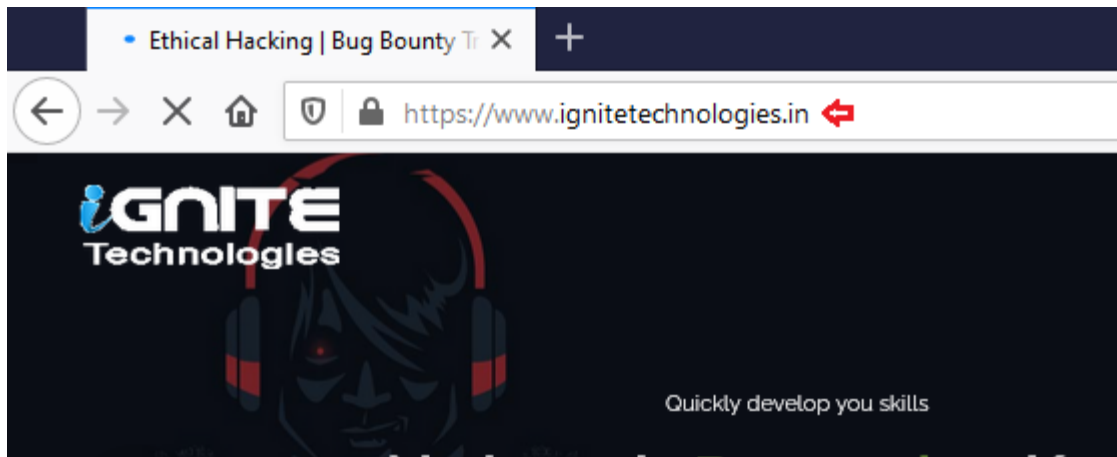
Cool!! Let’s check whether this would work or not. Manipulate the “url=” parameter with the **ignite’s encoded value**.

The screenshot shows the web proxy interface with the Intercept tab active. The request details are shown in a 'Raw' tab, displaying the following text:

```
GET /redirect/02-url-encod/re.php?url=%68%74%74%70%73%3a%2f%2f%77%77%77%77%2e%69%67%6e%69%74%65%74%65%63%68%6e%6f%6e%6f%6c%6f%67%69%65%73%2e%69%6e HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Gecko/20100101 Firefox/79.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer: http://localhost/redirect/02-url-encod/
```

From the below image, you can see that, we’ve successfully defaced the website over with the **“Open Redirection”** vulnerability with some simple clicks.





### Double URL Encoded Redirection

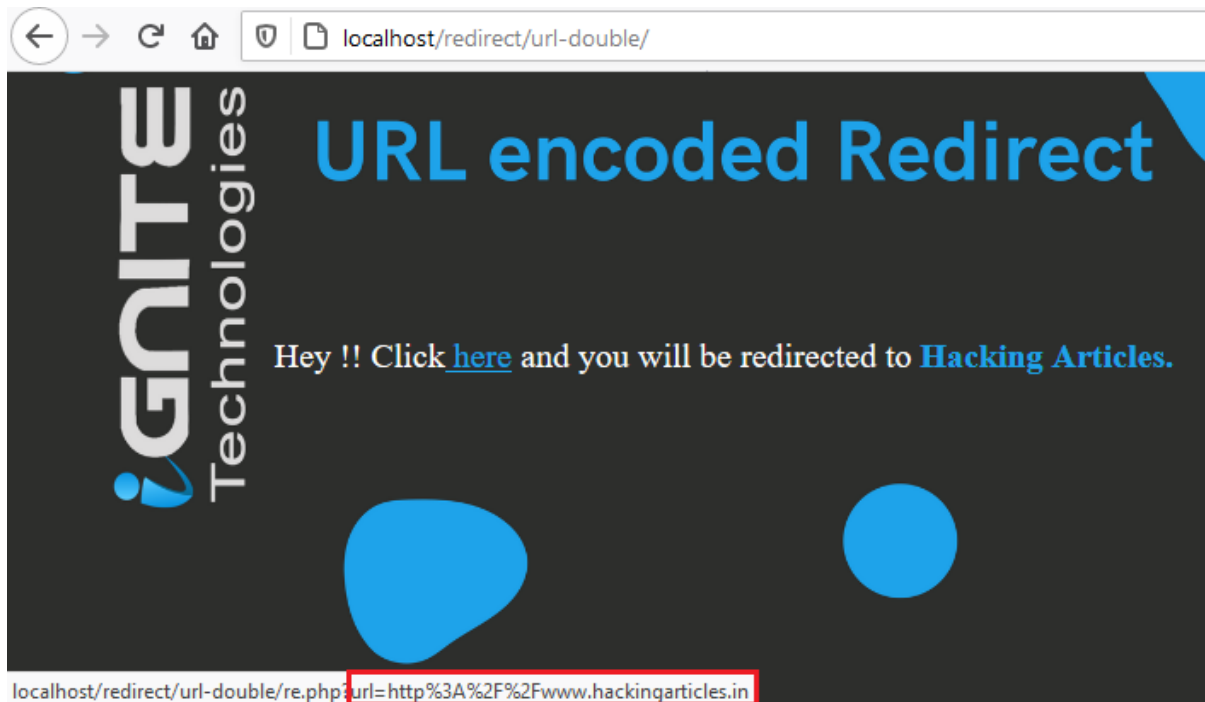
Being a developer, he knows that some simple clicks can easily bypass basic URL encoding; therefore, to make his application more secure in the redirection section, he implements “Double URL Encoding,” using the “urlencode() and urldecode()” functions twice, one after the other, as on the home page and the redirection page respectively.

*The following redirection code snippet explains how to perform the decoding. It first takes the encoded URL and decodes it. Then, using another urldecode function, it decodes the previously decoded URL and passes it for the redirection.*

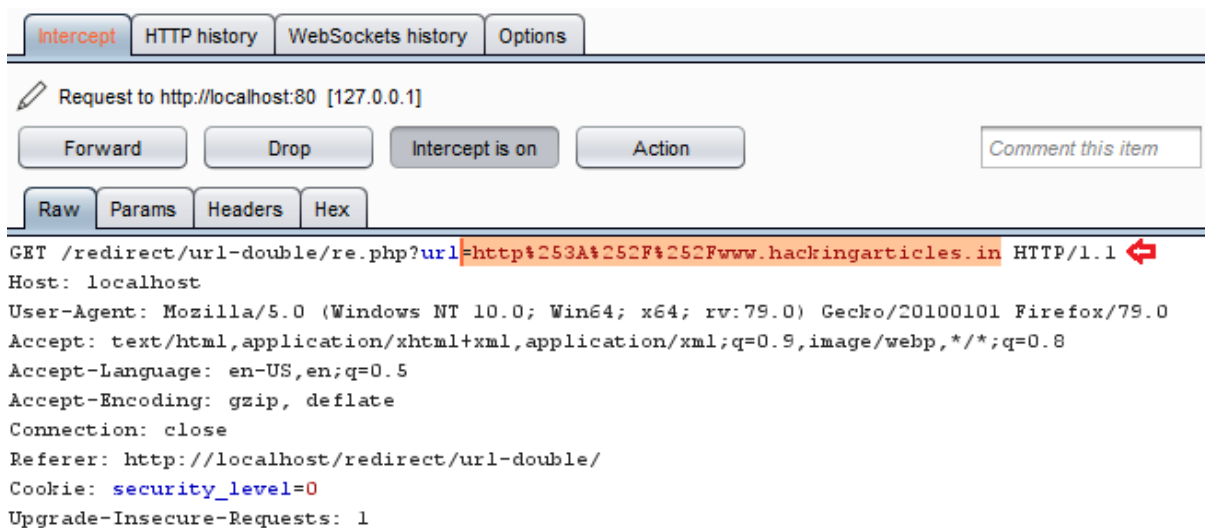
```
<html>
<?php
header('Location:'.urlencode(urldecode($_GET["url"])));
die();

?>
</html>
```

So now, as we hover on the “**here**” text, we’ll find that this time the URL is not in the readable format, thus to be more precise let’s capture this all over in the burpsuite.



From the below image, you can see that the “url=” parameter’s value is different from the one that we see earlier when we hovered on the text, which simply means that, there is some more encoding over it.



Let’s copy it out and check it over in the **decode** tab. From the below image, you can see that we got the decoded URL to be as “**http://hackingarticles.in**” when we opted the “**URL Decode**” option for about two times.

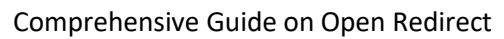


The screenshot shows a web application security tool interface with tabs for Decoder, Comparer, Extender, Project options, and User options. The Decoder tab is active. It contains three input fields and a list of decoding options. The first input field contains the URL `http%253A%252F%252Fwww.hackingarticles.in` with a red double-headed arrow. The second input field contains `http%3A%2F%2Fwww.hackingarticles.in` with a red double-headed arrow. The third input field contains `http://www.hackingarticles.in` with a red double-headed arrow. To the right of the input fields are radio buttons for Text and Hex, and a dropdown menu for Decode as ... The dropdown menu is open, showing options: Plain, URL (highlighted in red), HTML, Base64, ASCII hex, Hex, and Octal. A red arrow points to the URL option in the dropdown menu.

## Manipulating URL

Until now, we are aware that, this application is taking up the URL's that are **double-encoded**. Let's now try to deface this web-application by manipulating up its "**url=**" **parameter value** again with "***https://ignitetechnologies.in***".

But wait, before that, we need to implement the **double encoding** methodology, which will thus make the redirection successful.



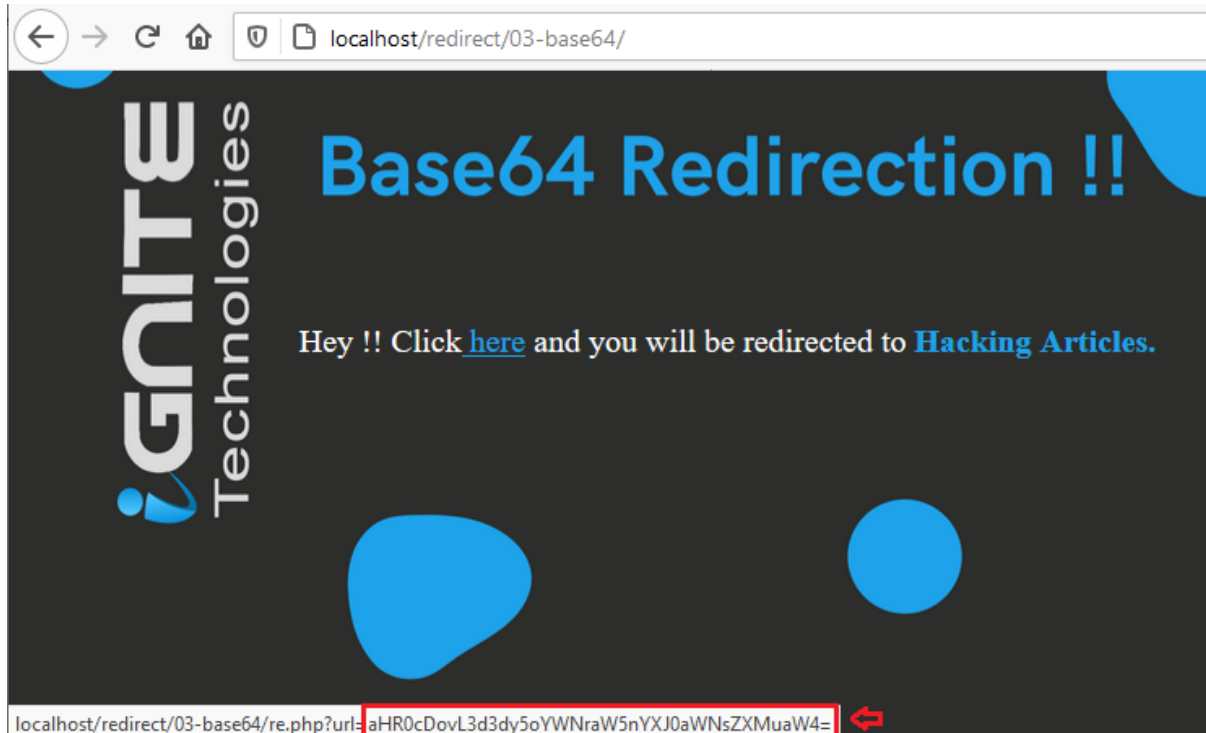
Copy the double encoded value, and paste it over to the **"url="** value.

Great!! We are almost done. Let's click on the **Forward button**, and check out what it displays to us.



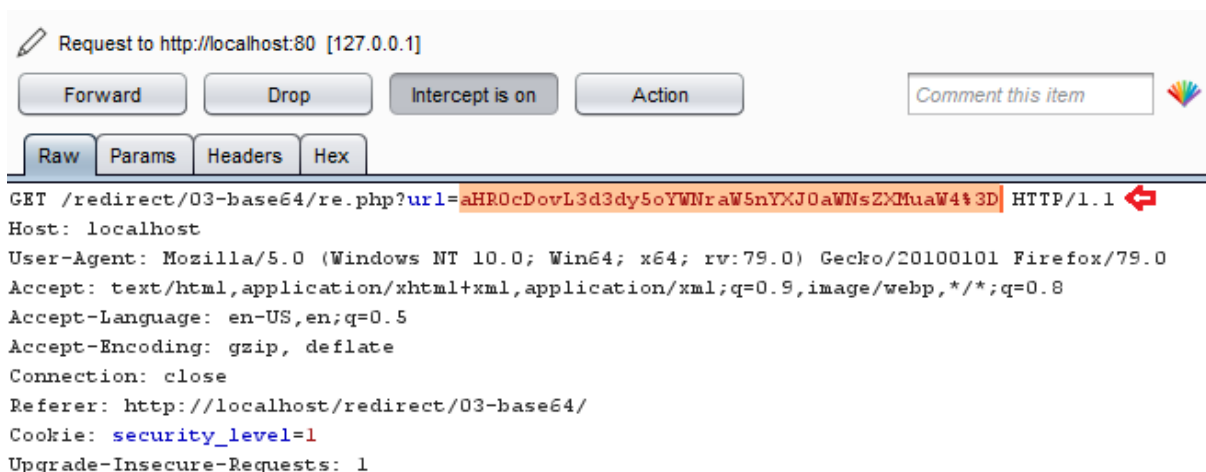
## Base64 Redirection

URL Encoding is not the only encoding methodology that the developer implements. Thus in order to make the **redirection process more secure**, they may use base64, hex, octal, binary, HTML or anything specific.



From the above image, you can see that if we hover over the “here” text. We get the “url=” value in the encoded form. So, let’s try to bypass this encoding:

In my burp suite monitor, I’ve captured up the ongoing request and have copied the “url=” value.



Now, with the **copied encoded value**. I tried to **decode** it with different encoding methodologies, and thus there, I got it for **Base64**.



For encoded binaries (like images, documents, etc.) use the file upload

UTF-8 Source character set.

☐ Decode each line separately (useful for multiple entries).

☐ Live mode OFF Decodes in real-time when you type or paste (s

< DECODE > Decodes your data into the textarea below.

http://www.hackingarticles.in

Cool!! As we are now aware of the encoding method. Let's now try to encode some other URL's and check for their outcomes.

Back in the burpsuite, and in the **decode** tab. I've tried to encode "**google.com**" over with **Base64**. There I've further copied up its encoded value.

Text Hex ?

Decode as ...

Encode as ...

Plain

URL

HTML

Base64

ASCII hex

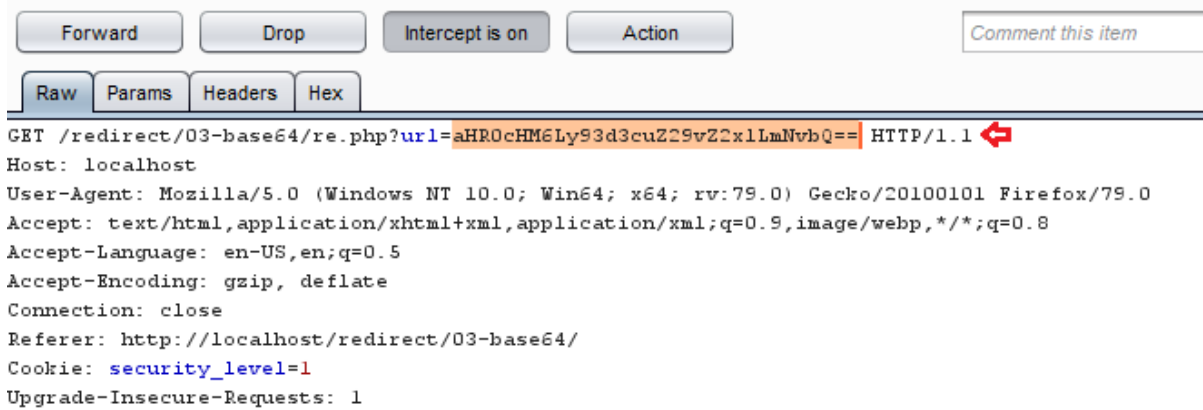
Hex

Octal

aHR0cHM6Ly93d3cuZ29vZ2xILmNvbQ==

Let's now manipulate the "**url=**" **parameter** with the copied value and thus then fire up the **Forward button** and check the response over back in the browser.





Great!! From the image below, you can see that we've again successfully bypassed this security.



Let's check out why this all happened:

From the below code snippet, you can see that the developer is again trusting his visitors and is reliable on the **header()** function which **first decodes** up the **encoded input URL** with **URL decode** and then with **base-64 decoding**. Thus further redirects the user to his desired webpage.

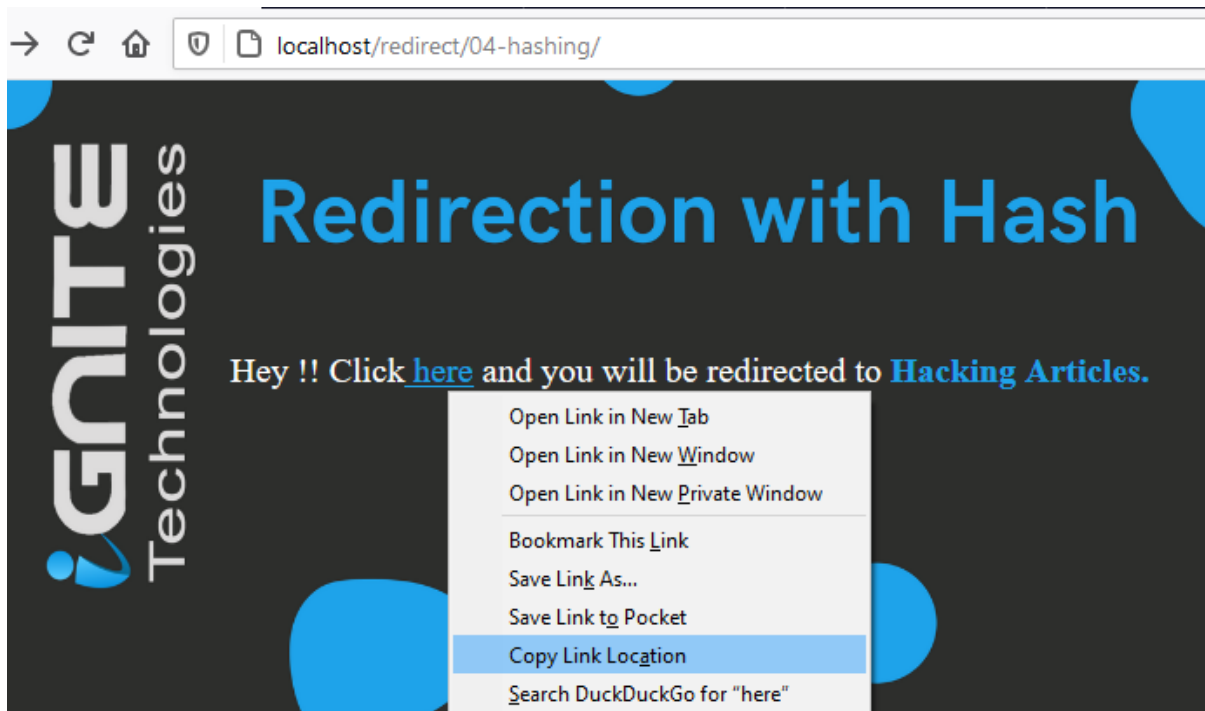
```
<html>
<?php
header('Location:'.base64_decode(urldecode($_GET["url"])));
die();

?>
</html>
```

## URL Redirection with Hash values

Experienced developers use hash values, which immunizes the web applications from the "Open Redirect" vulnerability. They could have used any hashing algorithm, whether it is MD5, SHA512 or SH1 or any other.

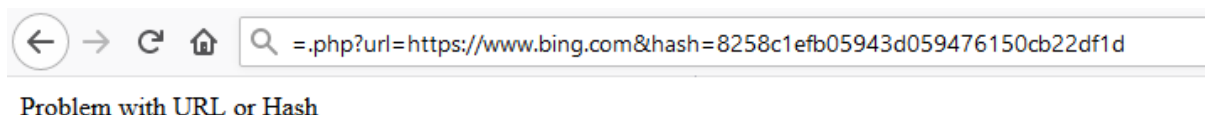
Some developers even implement the **combination** of **URL** and the **hash values** as we've used in this section i.e. when choose the "Copy Link Location" of "here" text we'll get the output



From the below image, you can see that this time rather than the “url=” parameter. We have one more as “hash=”.

```
http://localhost/redirect/04-hashing/re.php?url=http://www.hackingarticles.in;hash=8258c1efb05943d059476150cb22df1d
```

Now, if we manipulate the “url=” with “https://www.bing.com” and leaves up the hash value the same as it was. Thus then we will face the error.



*“As when we clicked on the “here” text, the redirection script will catch the passed URL and generate its hash value. And **compare the generated hash value with the hash value we have sent with the request.** If both the hash values match the redirection would **work** else it will **fail**.”*

*Here in this segment, the developer used the **MD5 hash algorithm**.*

So, let’s now try to exploit this major security again with some manipulations. But this time we need to encrypt our URL over with the **MD5 hash**.

Your Hash: **5b5951e4f10c268c49ff44a705023a13**

Your String: <https://www.bing.com>

Use this generator to create an MD5 hash of a string:

<https://www.bing.com> ↩

With a basic MD5 Hash generator, we've encrypted <https://www.bing.com>. Copy all this and craft it in with the URL.

```
http://localhost/redirect/04-hashing/re.php?url=http://www.hackingarticles.in&hash=8258c1efb05943d059476150cb22df1d
```

```
http://localhost/redirect/04-hashing/re.php?url=https://www.bing.com&hash=5b5951e4f10c268c49ff44a705023a13 ↩
```

Great!! From the below image you can see that, as soon as I execute the above-manipulated URL in the browser, I got redirected to my desired result.

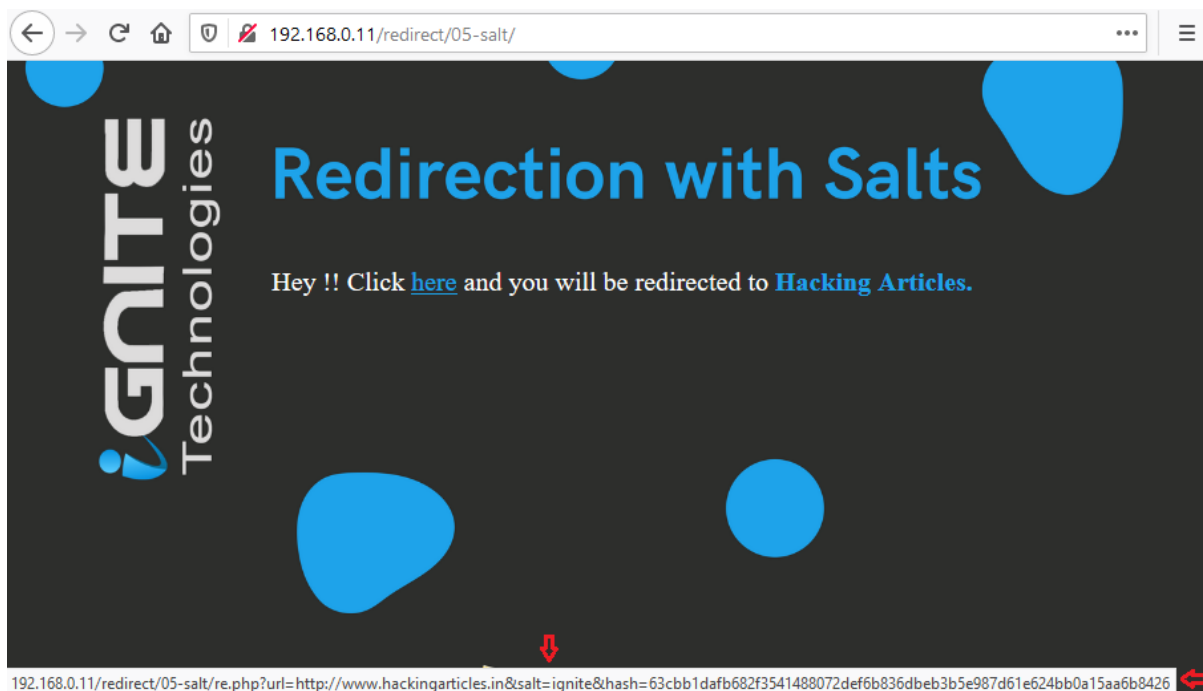


### Redirection with Hash values using Salt

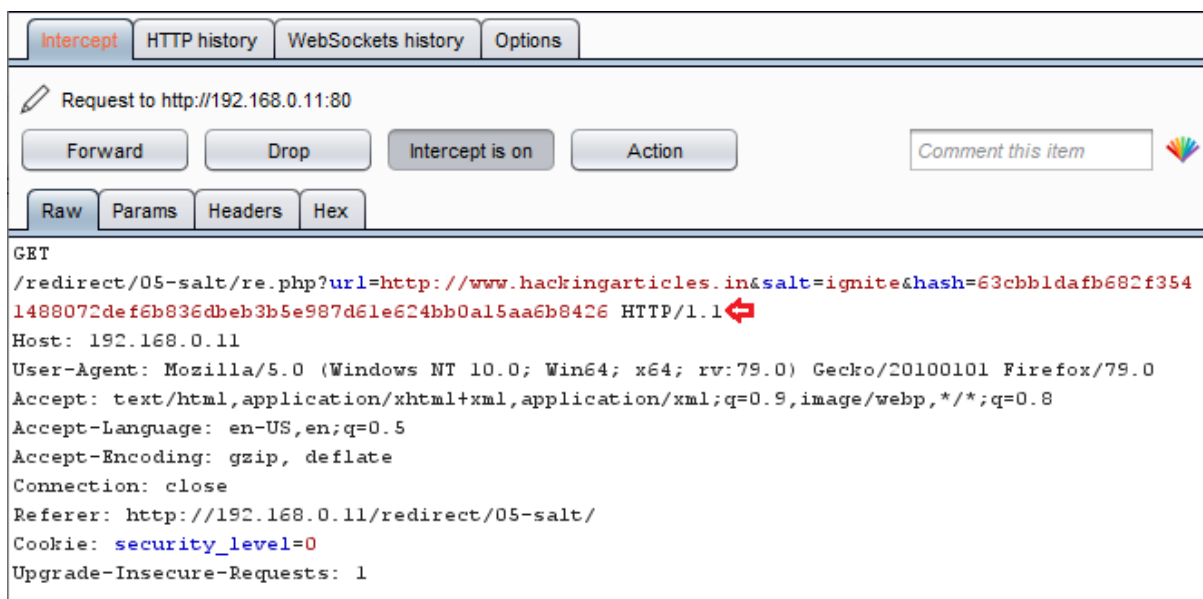
*Encoded URL's can be bypassed, URL parameter with a hash can even be misguided, but what, if the redirected URL is encrypted with a salt value?*

**Salt** could be **anything**, it could be a combination of characters, digits, alphanumeric, special character or anything we want. A salt further increases the security for redirecting the URL or even it makes up impossible to deface a web-application. But there is still a chance to misguide the users, if we could guess up the salt value or even if the developer displays it in the URL itself.

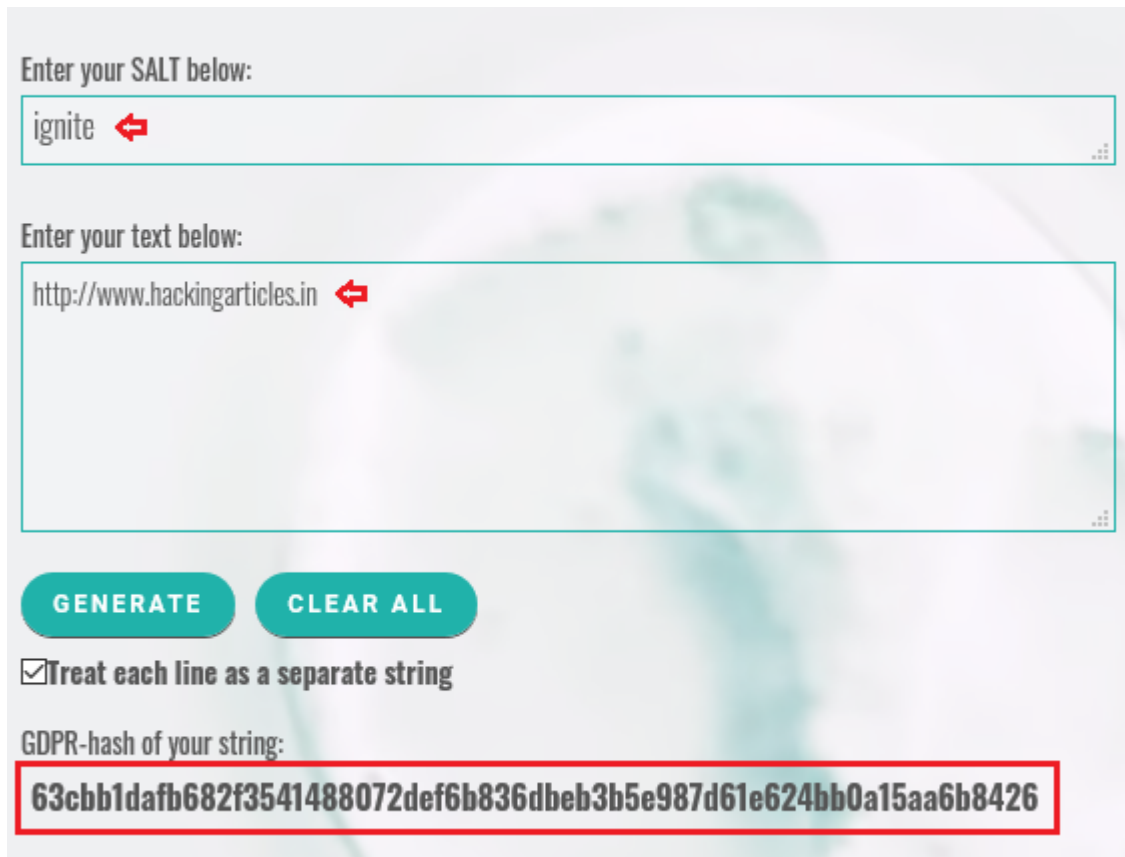
From the below image, you can see that here the developer passed up his salt value in the URL as **"ignite"**. Cool!! This was all we need to deface this web-application, let's try to do so.



Over in **burpsuite**, I've captured the ongoing HTTP Request that was generated with the "here" text.



Great!! So, it's time to manipulate the URL as we are now aware of **the salt value** but wait we don't know the type of hash it is using. Okay !! with some permutations and combinations and some hit & trial methods, I got it as a **SHA256 hash**.



The screenshot shows a web application for generating SHA256 hashes. It has two input fields: 'Enter your SALT below:' with the text 'ignite' and 'Enter your text below:' with the URL 'http://www.hackingarticles.in'. Below these are 'GENERATE' and 'CLEAR ALL' buttons. A checkbox labeled 'Treat each line as a separate string' is checked. The output, labeled 'GDPR-hash of your string:', is a long alphanumeric string: '63cbb1dafb682f3541488072def6b836dbeb3b5e987d61e624bb0a15aa6b8426', which is highlighted with a red border.

So, let's now try to deface this web-application by generating the hash value of our "URL" with the same procedure we decrypted that earlier.

Therefore, you can do so over through this [SHA256 hash generator](#).



Enter your SALT below:

ignite ↩

Enter your text below:

https://www.gmail.com| ↩

**GENERATE** **CLEAR ALL**

☒ Treat each line as a separate string

GDPR-hash of your string:

**2ca38c908f5e1e0c1431721644227869df14c0c8fe0e5ef009acf0525a43dd71**

Copy this all - the **URL** and the **Hash generated value**. And thus, **manipulate** it over in our **burpsuite**.

Request to http://192.168.0.11:80

Forward Drop Intercept is on Action

Comment this item

Raw Params Headers Hex

GET  
/redirect/05-salt/re.php?url=https://www.gmail.com&salt=ignite&hash=2ca38c908f5e1e0c1431721644227869df14c0c8fe0e5ef009acf0525a43dd71 HTTP/1.1 ↩  
Host: 192.168.0.11  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Gecko/20100101 Firefox/79.0  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Connection: close  
Referer: http://192.168.0.11/redirect/05-salt/  
Cookie: security\_level=0  
Upgrade-Insecure-Requests: 1

Woah!! We've successfully bypassed this security level which was expected to the most secure one.





### Redirection over inside a Web-Page

So, up till now, we have seen that the developer redirects his visitors to the resources that were outside the web-application. But there are chances that he might set the **redirect()** or **header()** function to make its users, to travel the web-pages which reside inside the application's interface.

Let's boot into our vulnerable web-application **bWAPP** as "**bee : bug**" and set the "**Choose your bug**" option to "**Unvalidated Redirects & Forwards (2)**".

Thus, there on the page, if we hover the "**here**" text we can see that there is a "**ReturnUrl=**" parameter.



Let's capture the passing HTTP Request over of the "**here**" text and check what we could grab with it.

From the below image. You can see that the "**ReturnURL=**" parameter forwards the user back to the "**portal.php**" page. Which is thus nothing but a URL redirection and therefore can leverage an "Open Redirect" vulnerability.



Request to http://192.168.0.11:80

Forward Drop Intercept is on Action Comment this item

Raw Params Headers Hex

GET /bWAPP/unvalidated\_redir\_fwd\_2.php?ReturnUrl=portal.php HTTP/1.1  
Host: 192.168.0.11  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Gecko/20100101 Firefox/79.0  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Connection: close  
Referer: http://192.168.0.11/bWAPP/unvalidated\_redir\_fwd\_2.php  
Cookie: security\_level=0; PHPSESSID=qkgrttckos8b63lccgcloj5eod4  
Upgrade-Insecure-Requests: 1

*Isn't it great that we, as low privileged users, can grab the content of the "configuration files" or anything specific for which we're not authorized, and this all happens when the developers patch the "LFI" vulnerability?*

So let's try to do so - as we're aware, that the **configuration files** reside inside the web-applications default folder. Thus it would be easy to call that up over from the "here" text, by manipulating the captured request with our desired URL.

[http://192.168.0.11/bWAPP/unvalidated\\_redir\\_fwd\\_2.php?ReturnUrl=config.php](http://192.168.0.11/bWAPP/unvalidated_redir_fwd_2.php?ReturnUrl=config.php)

Intercept HTTP history WebSockets history Options

Request to http://192.168.0.11:80

Forward Drop Intercept is on Action Comment this item

Raw Params Headers Hex

GET /bWAPP/unvalidated\_redir\_fwd\_2.php?ReturnUrl=config.inc HTTP/1.1 ↩  
Host: 192.168.0.11  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Gecko/20100101 Firefox/79.0  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Connection: close  
Referer: http://192.168.0.11/bWAPP/unvalidated\_redir\_fwd\_2.php  
Cookie: security\_level=0; PHPSESSID=g8qslrpj324333b2aadsre2ub7h  
Upgrade-Insecure-Requests: 1

As soon as we hit the **Forward** button. We got redirected to the **config.inc** which thus contains up all the basic configurations.

Great!! We're in the **config.inc** file and though we now having more information about our target's server.



```
← → ↻ 🏠 🔒 192.168.0.11/bWAPP/config.inc

/*
bWAPP, or a buggy web application, is a free and open sour
It helps security enthusiasts, developers and students to
bWAPP covers all major known web vulnerabilities, includin
It is for security-testing and educational purposes only.

Enjoy!

Malik Mesellem
Twitter: @MME_IT

bWAPP is licensed under a Creative Commons Attribution-Non
(http://creativecommons.org/licenses/by-nc-nd/4.0/). Copyr

*/

// Connection settings
$server = "localhost";
$username = "alice";
$password = "loveZombies";
$database = "bWAPP_BAK";

?>
```

## DOM-based Open Redirect

DOM-based open-redirection vulnerabilities arise when a script writes attacker-controllable data into a sink that can trigger cross-domain navigation which thus **facilitates phishing attacks** against users of the websites.

### So, what is the sink?

*An attacker can cause undesirable effects if they pass controlled data to a potentially dangerous JavaScript function or DOM object.*

*The major sinks that could lead to DOM-based Open Redirection vulnerability are “**location**” “**location.hostname**” “**location.href**” “**location.pathname**”.*

So, let's try to implement it, in some real scenarios over at [The Portswigger Academy](#) and exploit this DOM-based Open Redirect vulnerability.

[Web Security Academy](#) » [DOM-based](#) » [Open redirection](#) » [Lab](#)

## Lab: DOM-based open redirection



PRACTITIONER

LAB

Solved



This lab contains a DOM-based open-redirection vulnerability. To solve this lab, exploit this vulnerability and redirect the victim to the exploit server.

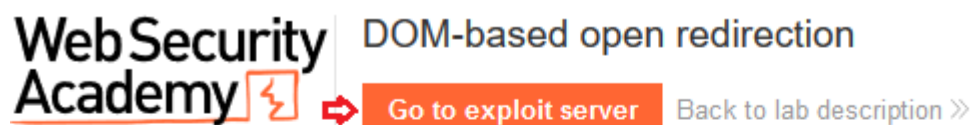
[Access the lab](#)



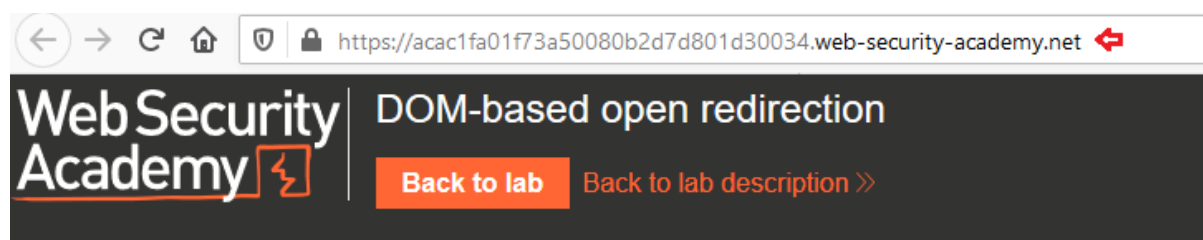


As we hit the “**Access the lab**” button. A blog that is somewhere or the other suffering from “**Open Redirection**” redirects us.

But before that, we got something new as “**Go to exploit server**”, let’s check that out.



That’s nice, we’re having our exploit server here, let’s copy its URL and we’ll try to redirect that blog to our server.



## Craft a response

URL: <https://acac1fa01f73a50080b2d7d801d30034.web-security-academy.net/exploit>

HTTPS

So, let’s open a specific blog-post, say the first one and check **its source code** with a simple right-click over any segment of the page.

https://ac8e1f531f16a5a78017d73500e000c0.web-security-academy.net/post?postId=1

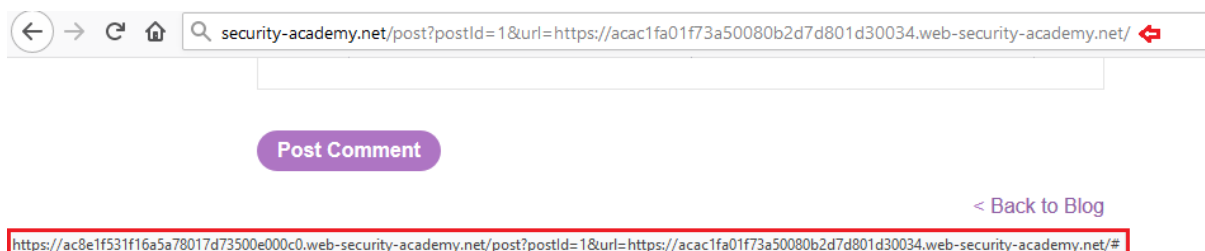


Great!! From the below image, I see that the “Back to Blog” text contains some URL, indicating an “Open Redirect” since the code does not validate or sanitize user input in the “location” property, and it even sets an onclick event as “url=(https”

```
<input pattern="(http|https):.+" type="text" name="website">
<button class="button" type="submit">Post Comment</button>
</form>
</section>
<div class="is-linkback">
<a href="#" onclick='returnUrl = /url=(https?:\\\/.+)/.exec(location); if(returnUrl)location.href = returnUrl[1];else location.href = "/">Back to Blog</a>
</div>
</div>
</div>
```

However, if we manipulate the **web-page URL** by **integrating our exploit server’s URL** with “&url=” parameter as

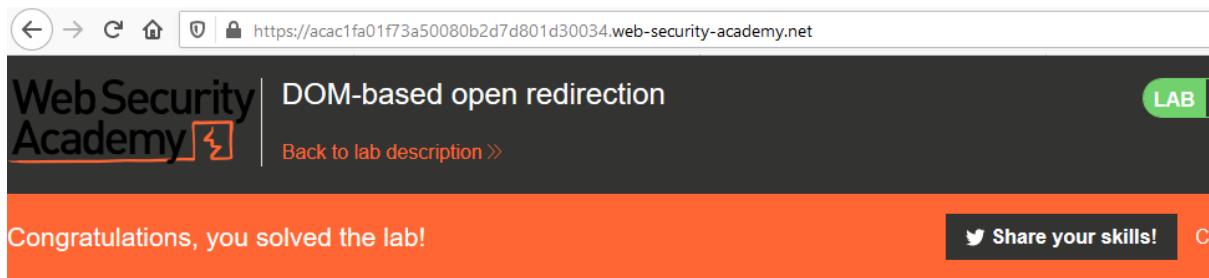
web-security-academy.net/post?postId=1&url=https://acae1fa01f50080b2d7d801d30034.web-security-academy.net



From the above image, you can see that, now if I hover the “Back to Blog” section. I got the redirection link to the one I manipulated earlier.

Let’s check out its output, by clicking over the “Back to Blog” text.

Great!! From the image below, you can see that we successfully solved this lab and thus redirected ourselves to our exploit server.



## Mitigation Steps

- The system should properly sanitize and validate the user input.
- The developer should set up a whitelist of his trustworthy URLs.
- If the developer is setting up any hash or a salt value, they should hide it and not display it in the URL.
- Instead of using the redirect function, the developer should implement specific links over on the text keywords.
- The web-application should pop out a warning when a user tries to redirect to an untrusted domain.

To learn more about Website Hacking. Follow this [Link](#).



# JOIN OUR TRAINING PROGRAMS

