Web Application Hacking Lesson – In Depth XSS

Lesson Objectives

- Complete lab on Cross Site Scripting(below)

Lab for Cross Site Scripting Introduction

Lab setup

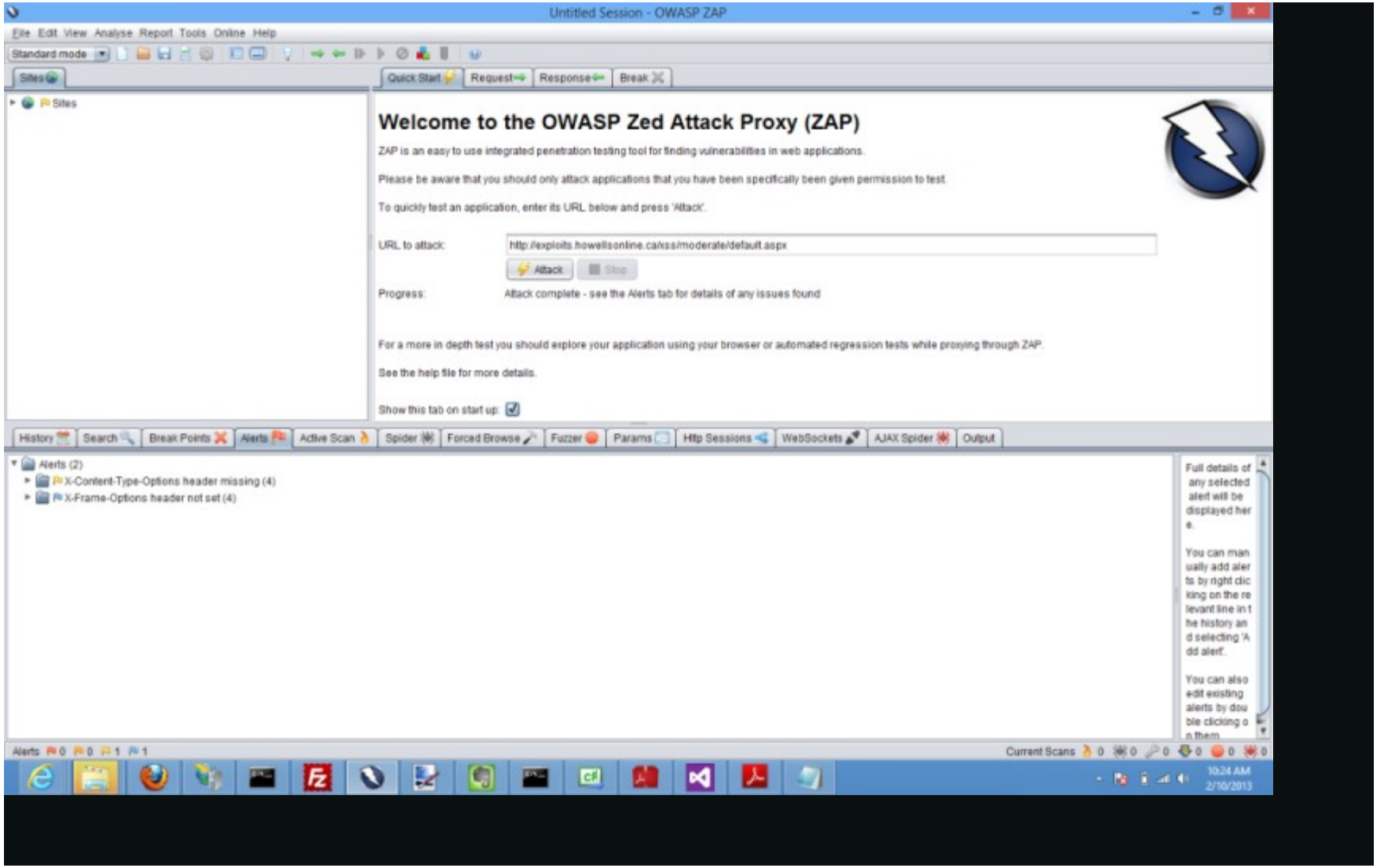
- **Instructions**:
  1. The target site is located here: http://exploits.howellsonline.ca/xss/simple/default.aspx
  2.

Cross Site Scripting

There are essentially 5 main input areas where we can introduce cross site scripting vulnerabilities. They are:

1. Hidden Fields on a Form
2. Session Storage
3. Cookie Storage
4. Injected Files
5. Injected Cookies

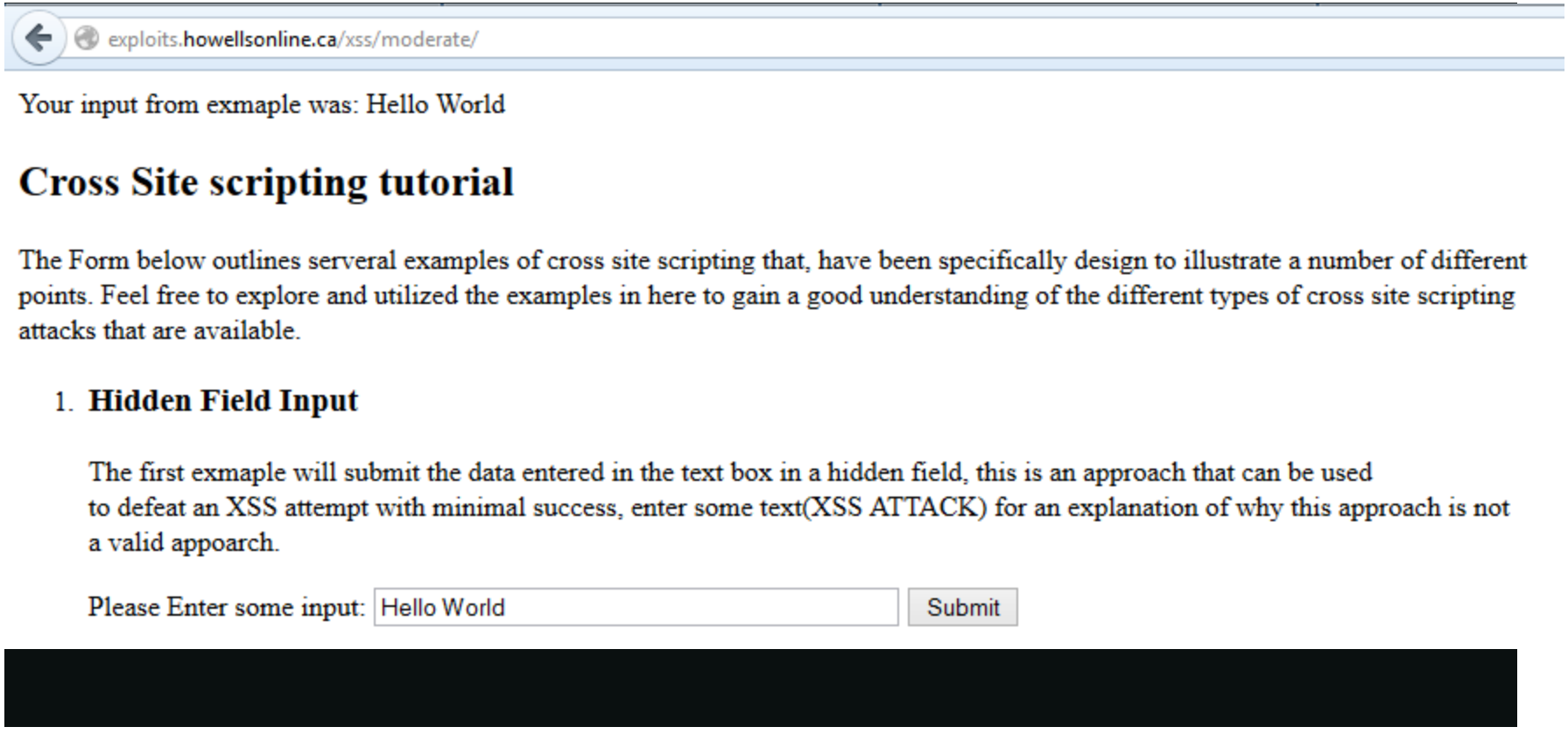
Some of these areas are more severe and damaging for various reasons which I'll explore as I go. An interesting side note, if you notice the screen shot, below, or try it for yourself. When I point the ZAP attack proxy at http://exploits.howellsonline.ca/xss/moderate/default.aspx the ZAP attack proxy does not detect any cross site scripting vulnerabilities, as seen in the image below. **Therefore it's not always enough for QA or Dev to use an attack proxy and expect to find all the vulnerabilities through such a manner, attack proxies are just a tool in the tool box but should not replace actual testing.** These examples for this demo have been tested and work in firefox. Your mileage may vary in other browsers. Note wordpress seems to have some difficulties with the javascript attack strings rendering properly. You can find them in order for this tutorial here


Untitled Session - OWASP ZAP
File Edit View Analyse Report Tools Online Help
Standard mode
Sites
Quick Start
Request
Response
Break
Sites
Welcome to the OWASP Zed Attack Proxy (ZAP)
ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications.
Please be aware that you should only attack applications that you have been specifically been given permission to test.
To quickly test an application, enter its URL below and press 'Attack'.
URL to attack:
http://exploits.howellsonline.ca/xss/moderate/default.aspx
Attack
Stop
Progress:
Attack complete - see the Alerts tab for details of any issues found
For a more in depth test you should explore your application using your browser or automated regression tests while proxying through ZAP.
See the help file for more details.
Show this tab on start up:
History
Search
Break Points
Alerts
Active Scan
Spider
Forced Browse
Fuzzer
Params
Http Sessions
WebSockets
AJAX Spider
Output
Alerts (2)
X-Content-Type-Options header missing (4)
X-Frame-Options header not set (4)
Full details of any selected alert will be displayed here.
You can manually add alerts by right clicking on the relevant line in the history and selecting 'Add alert'.
You can also edit existing alerts by double clicking on them
Alerts
Current Scans
10:24 AM
2/10/2013

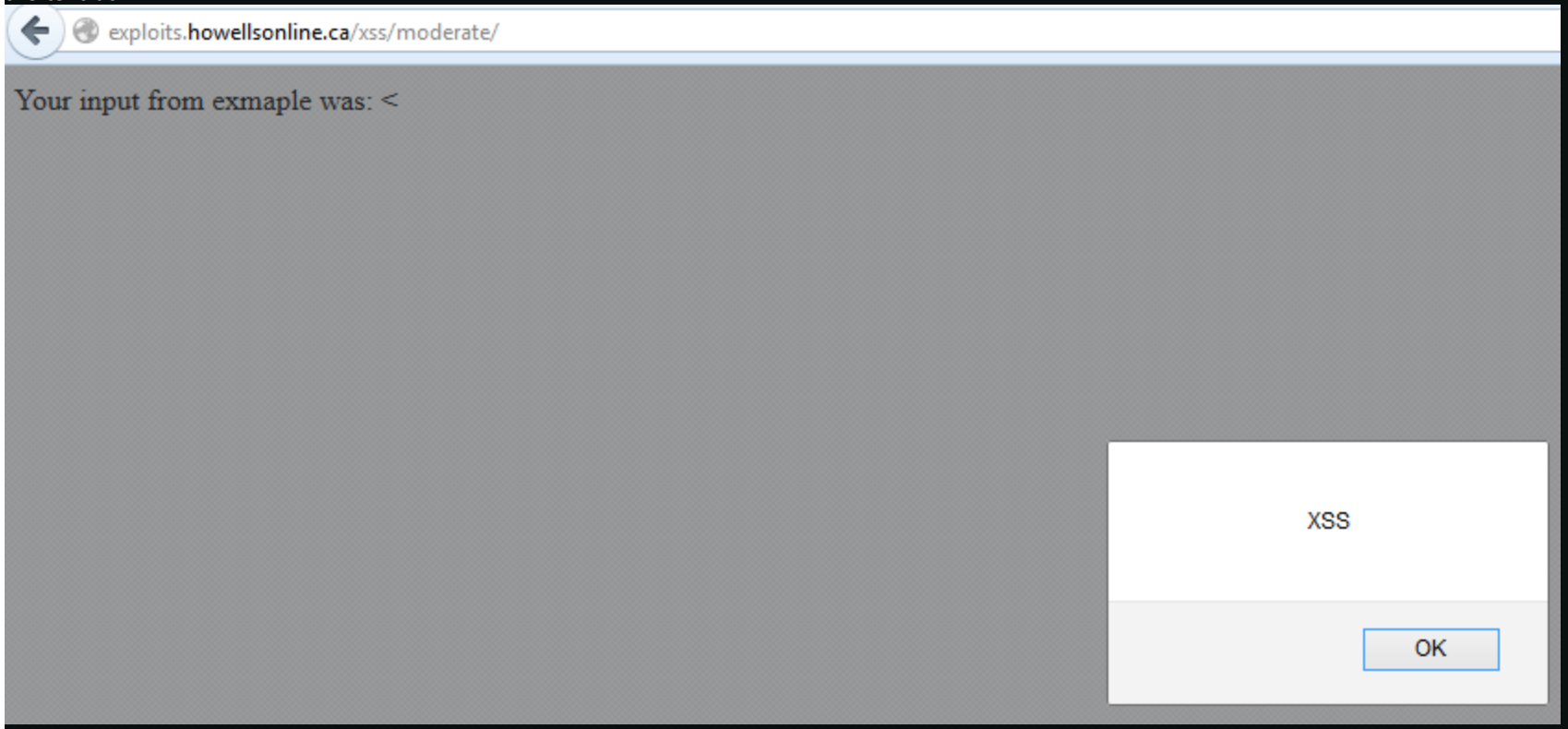
Hidden Input Field

The hidden input field on an html form is used for storing input that's well hidden from the user. Essentially what the first example in the demo does is set a hidden input and post back, on the post back the hidden form value is read and displayed to the user. When used correctly everything works as expected and indicated from the image below.
The attack string we'll use to demonstrate this vulnerability is:

```
<script type="text/javascript">// <![CDATA[
alert("XSS")//<
// ]]></script>
```



However when this input is used for malicious purposes & the cross site scripting vulnerability is discovered look at the outcome from the image below. To try this example below yourself input the following string into
the text box.



Now lets examine the code that causes this vulnerability.
Client Side
Javscript:

```
<script type="text/javascript">// <![CDATA[

function populateData() {            form1.elements["m_hdnInpt"].value = document.getElementById("m_txtHidden-
Field").value;          }
```

```
// ]]></script>
```

Server:

```
if (Page.IsPostBack)
            {
                Response.Output.WriteLine("Your input from exmaple  was: " + m_hdnInpt.Value);
            }
```

It's obvious to see what the problem is here, we are populating a hidden input on client side, with the value of the user input and then blatantly writing the value back into the raw HTML on the server to be redisplayed to the client. Note that this is not detected by ZAP as a cross site scripting vulnerability. The server should not just trust the input coming from the client.

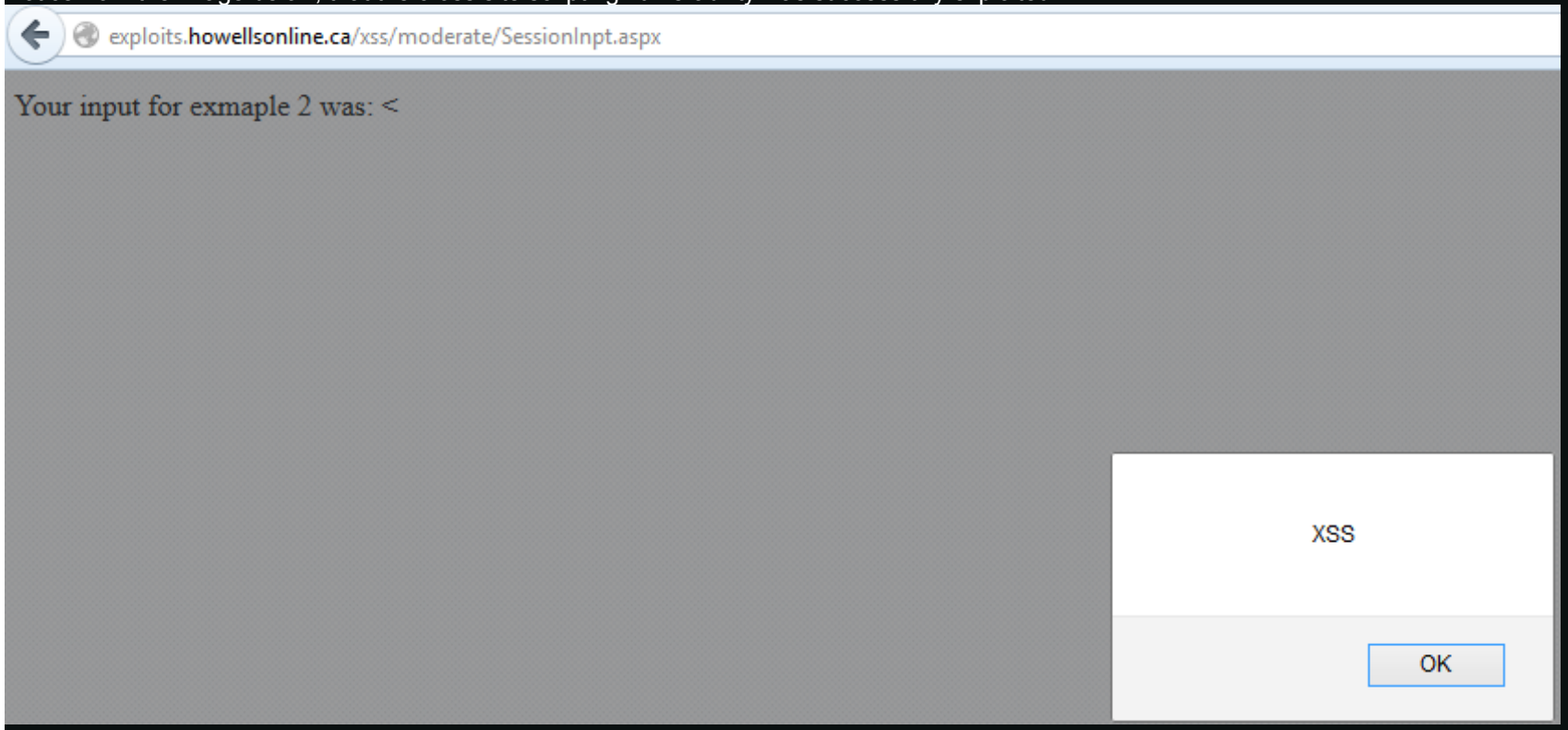
Session Storage

In this example, if you want you can first test on the demo application that normal input works as desired, I know it does so I am going to move straight to the vulnerability. The difference with between the first example and this example is that, in the first example the form posted back to Default.aspx, in this example the form redirects to SessionInpt.aspx, this example and the remaining examples are similar to the first post in this series were data is being passed between two pages.

The attack string I am going to use for this example is:

```
&lt;<script type="text/javascript">// <![CDATA[
alert("XSS");//<
// ]]></script>
```

Notice from the image below, that the cross site scripting vulnerability was successfully exploited.



Now lets examine the code:

Server Side

```
Session["SessionInpt"] = m_sessionInpt.Text;
            Response.Redirect("/xss/moderate/SessionInpt.aspx");
```

Redirect Server

```
Response.Output.WriteLine("Your input for exmaple 2 was: " + Session["SessionInpt"]);
```

As you can see Default.aspx blindly throws the data from the user input into a session variable, which SessionInpt.aspx gladly accepts, trusting that everything in the session is safe. If I were in a leadership position reviewing this code, I would be upset that Default.aspx did not do any input validation, I would be even more upset that SessionInpt.aspx did not do any input validation and just trusted the input. This type of a situation can result if you have two devs working on these pages separately each trusts the other to handle security. This is also very good justification for performing a security review. Having validation on both pages is best and builds on a concept of defense in depth.

Cookie Storage

This example is one of my favourites, and the reason being is because of the many ways that it can actually impact the user. Up until this point the previous two examples and the previous post were essentially a 1 time cross site scripting attack. In this example we're actually going to store the attack in a cookie.
The attack string we'll use for this attack is:

```
&lt;<script type="text/javascript">// <![CDATA[

alert("XSS Attack")//<

// ]]></script>
```

As you can see from the image below the exploit is successful.

The more disturbing trend here is because this exploit is saved in a cookie everytime the user visits the page they will re see the exploit. If you're following along in the demo application I would encourage you to enter the attack string on Default.aspx. When you're redirected, copy the url that you're redirected too, in your URL bar navigate to a different url say google. Once Google loads, past the redirected url that you copied back into your browser. Notice how the exploit immediately reappears and will continue to reappear until you clean your cookie. If you would like to clean you cookie and try again, selected the delete cookie button.

Lets examine the code:

Server Side

```
Response.Cookies["UserInpt"]["Data"] = m_txtCookieInpt.Text;
          Response.Redirect("/xss/moderate/CookieInpt.aspx");
```

Redirected Page

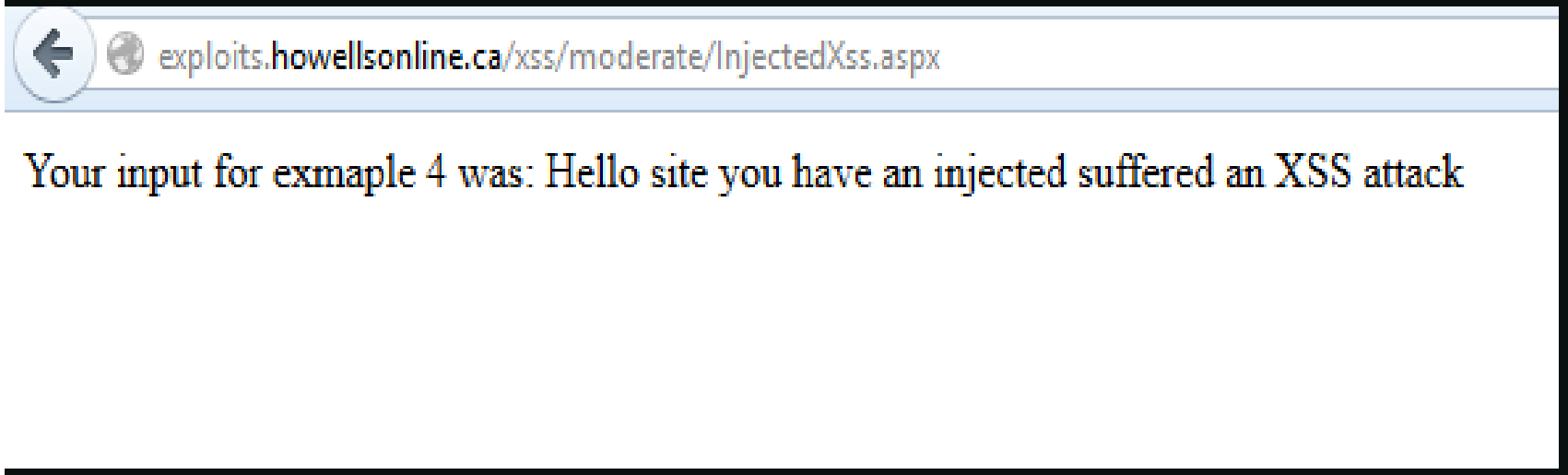
```
String txt = Request.Cookies["UserInpt"]["Data"];
            Response.Output.WriteLine("Your input for exmaple 3 was: " + txt);
```

As you can see, Default.aspx, drops a cookie when the button is clicked, that CookieInput.aspx is going to pick up and constantly read from the cookie for exploits.howellsonline.ca until the cookie is cleared, therefore the user will constantly see this exploit, and it may drive them away from your site.

Injected Files

This is one of the more advanced forms of cross site scripting, this is where the attack attempts to execute some malicious code on your website, they do this by injecting their own javascript file.
For this example our attack string will be:

Look at the output shown below:



That wasn't at all what you entered in the text box was it? How was the attacker able to do this? Lets look at the code: **Server Side**

```
Session["Injected"] = m_txtBoxInjected.Text;
            Response.Redirect("/xss/moderate/InjectedXss.aspx");
```

Redirected Page

```
            String txt = (String)Session["Injected"];
            Response.Output.WriteLine("Your input for exmaple 4 was: " + txt);
```

There is one more important file that we need to look at and that, was the script we injected which can be found here

Injected Script

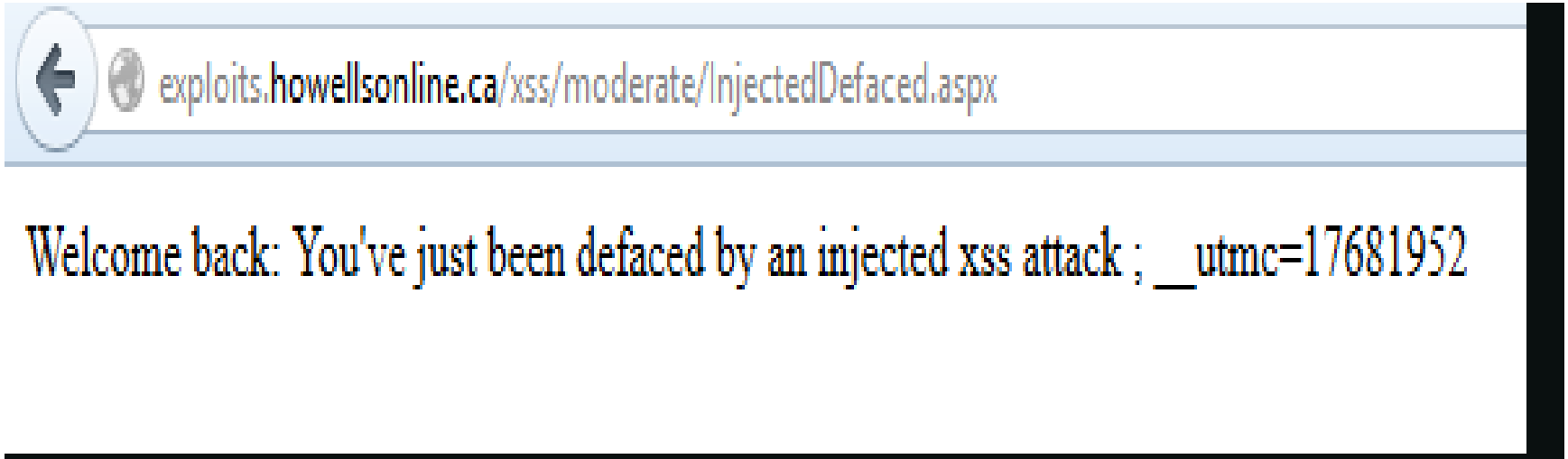
```
document.writeln("Hello site you have an injected suffered an XSS attack");
```

This attack was pretty simple essentially I was able to get my injected script to run. The reason that this was possible was because the Redirected page blindly accepted input from Session, and immediately wrote that input to the HTML without verification. When it did this and the page rendered, the page went off and made an HTTP GET request for the injected url, as soon as the requested script was received the script executed and defaced my web page. Interestingly enough up until this point, all of the other attacks that we've seen have displayed a message box, however this script does not display a message box and if it were not for the output the user wouldn't know that this script had executed at all.

Injected Cookies

This next example demonstrates the combination of a couple of attack vector's we've already seen, in this example we are going to inject a script into a cookie, every time the page loads the page will download the script as in the previous example.
The attack string we'll use for this attack is:

As you see the attack is successful.



Up until this point you may have noticed a pattern all of the previous examples use the pattern

```
Response.Output.WriteLine(Some OUTPUT)
```

It would stand to reason then that as long as you don't use this pattern that you're reasonably okay. Up until this example you would be correct. Lets' look at the code:

Server Side

```
 Response.Cookies["UserInpt"]["Deface"] = TextBox1.Text;
            Response.Redirect("/xss/moderate/InjectedDefaced.aspx");
```

InjectedDefaced

```
<script type="text/javascript">// <![CDATA[
function handleCookies() {          var cookie = document.cookie;          var cookieValue = cookie.substr(cookie.lastIn-
dexOf("Deface=") + 7, cookie.length - 7);          document.write("Welcome back: " + cookieValue);      }
// ]]></script>
```

Notice the difference here? The difference is that the cookie is read from the client when the HTML body loads. So instead of the vulnerability being made in the back end server side code, it's actually all happening on the client, therefore the request does not have to back to the server and come back to the client.

In Closing

The truest danger with example #5 is, once an attacker has their url for javascript stored in somewhere like a cookie, they can modify the javascript to do whatever they want, against the unsuspecting user. So perhaps for maybe a month my javascript file does nothing at all, Then I have it start stealing the data the user enters from on the form of your website. Because the url for my attack vector is saved in your cookie I almost have complete control to do as I wish.

I hope the moral of the story is do not trust unvalidated user input. I've demonstrated 5 sources that user input can come from, all of it as you see should be properly validated before being displayed and shown to the user. I would even recommend validating the input both on input and before you should it to the user, however as a minimum it should be validated before being written to the html.

-

Proof of Lab

1. Proof of Lab
    - **Proof of Lab Instructions:**
        1. Do a <PrtScn> of all injection screens
        2. Paste into a word document
        3. Email me
    -