

Deliverable #3: Report

Group 3

Team Leader: Tai-Juan Rennie

Webmaster: Vivian Liang

Report: Matthew Trinh

Report: Shahid Nawaz

Course Code: BTN710NBB

Introduction

In this report we will discuss how cross-site scripting can be used to inject malicious code into vulnerable web applications. We will go into detail about what cross-site scripting is, when it can be used, and how it works. We have built and deployed a web application that will be used to demonstrate how cross-site scripting can be used in a real-world environment. We will also discuss how cross-site scripting can be detected and what can be done to ensure that your web application is protected against it.

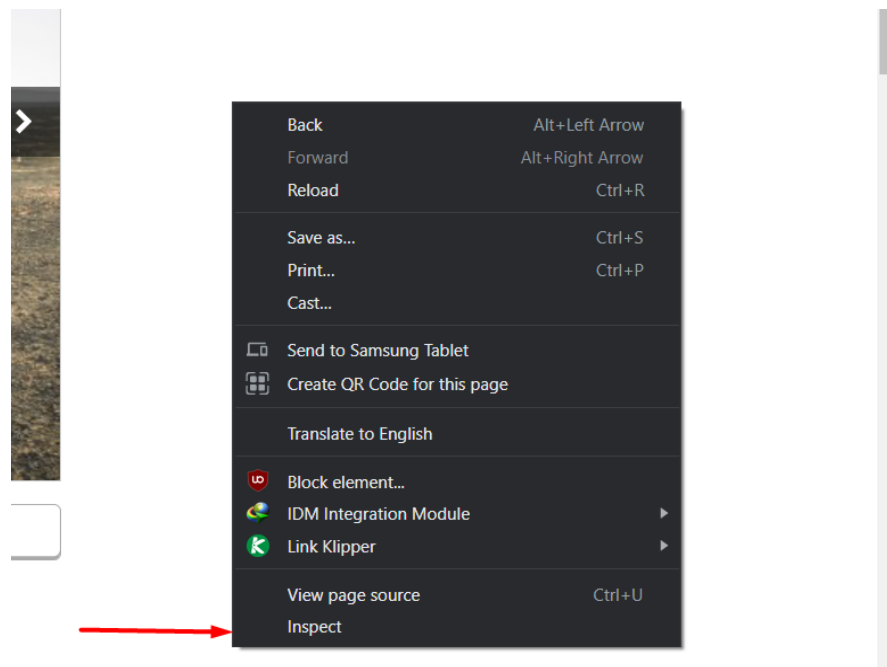
Section 1: Vulnerability

The vulnerability our group chose to explore is cross-site scripting. Cross-site scripting attacks are injections of malicious scripts into otherwise trusted websites. The malicious scripts are sent to an unsuspecting user of the trusted website, where the user's browser will execute the script upon loading the webpage. Because the browser has no way of knowing which parts of the script are malicious, it will run everything as if it were trusted. This means the attacker has access to the unsuspecting user's cookies, session tokens, and any other personal or sensitive information that would otherwise only be available to the trusted source itself.

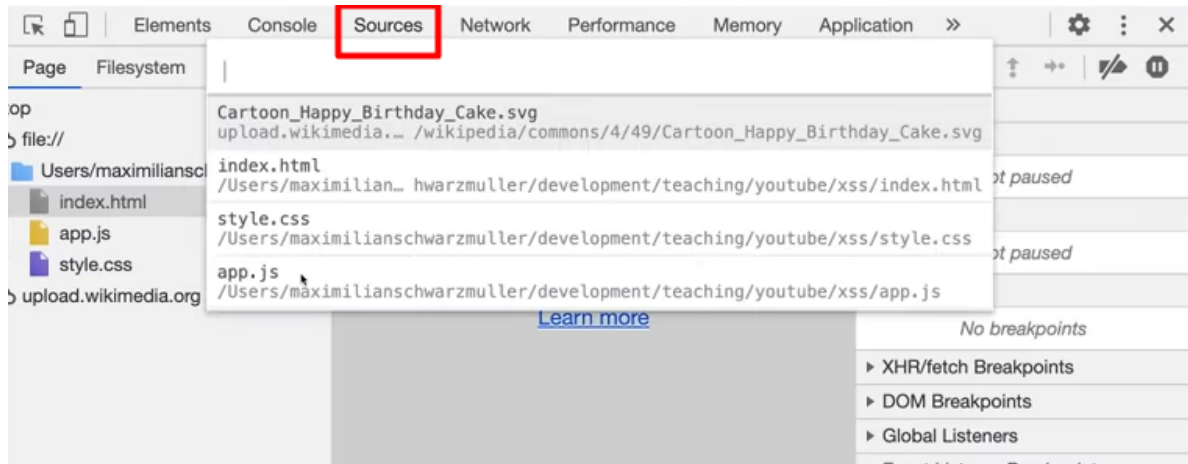
Once the attacker has access to the unsuspecting user's cookies and session tokens, the attacker can then impersonate the user and perform actions as the user to gain increasingly sensitive data belonging to the user. Because JavaScript in modern browsers have access to HTML5 APIs, the attacker with some creativity in social engineering may gain access to the user's webcam, microphone, geolocation, or even private files within the user's file system on their operating system.

First thing we need to find out on which site we can execute our malicious code. For that we need to inspect the website in a browser.

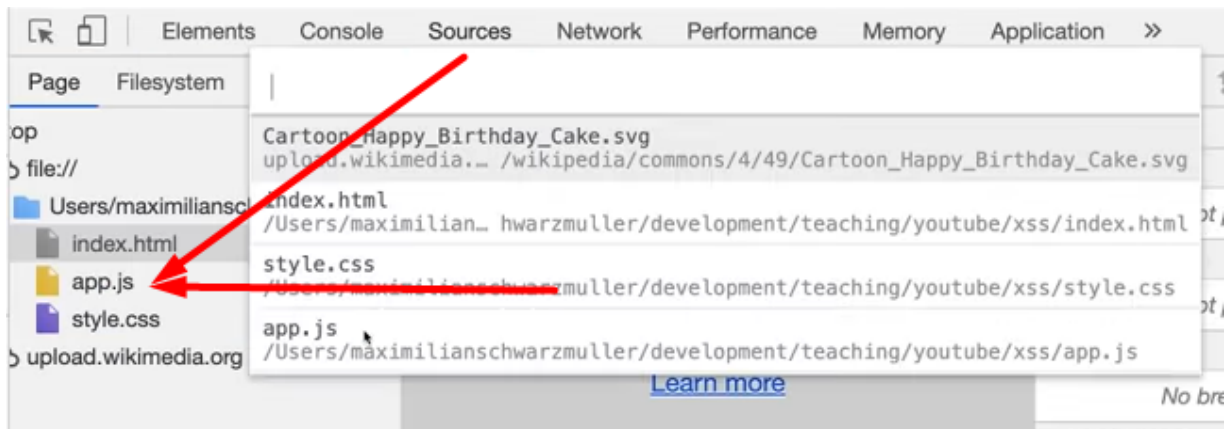
We can either press F12 key or right click and then select "inspect" on any web page.



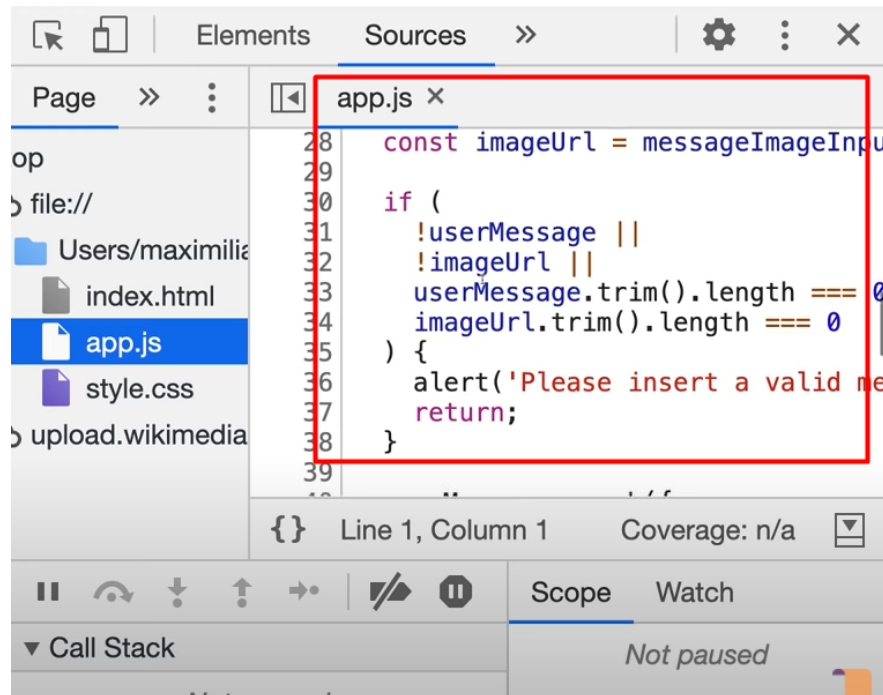
It opens another screen on the side or bottom depending on settings



In that screen we can go to the sources tab and can locate app.js on the side menu



Then we can simply open app.js or any .js file to take a look at javascript code of that site



Section 2: System Setup

The system that we are exploiting is a web application hosted on Heroku. The back end is running on a Node.js server served through Express that uses Postgres as the database. Handlebars is being used as the frontend templating framework.

Cross-site scripting is versatile in that a web browser is all that is needed. This means that this attack is not restricted to specific hardware or operating systems. In our case, we used Google Chrome running on a Windows Desktop computer.

This attack does not work by exploiting vulnerabilities in networks or protocols, therefore no discussion of network, protocol, or service is necessary to replicate this attack.

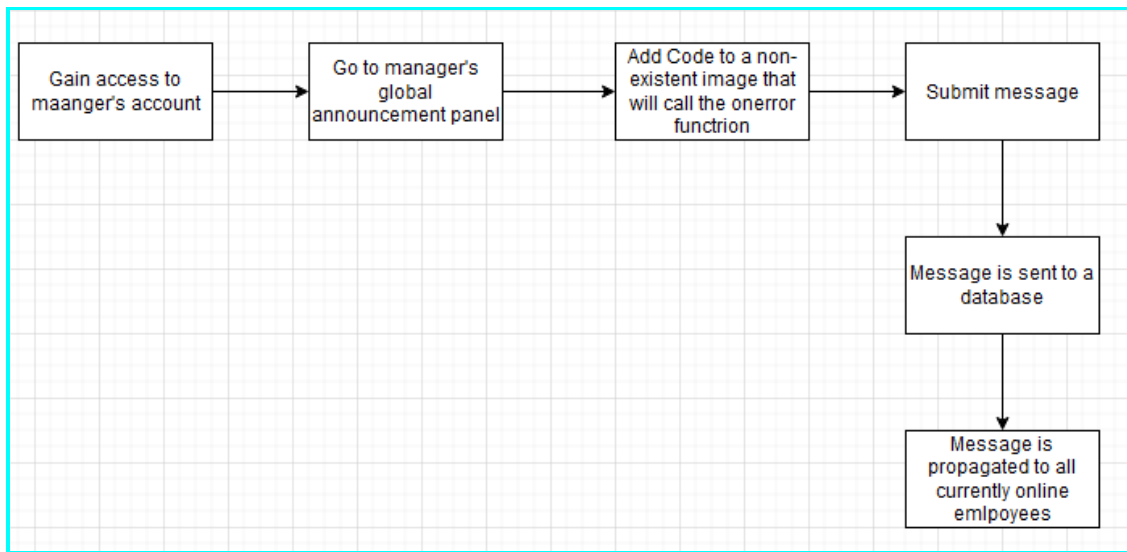
Section 3: The Exploit

Cross-site scripting works by taking advantage of vulnerabilities in source code. Mainly, when the developer neglects to clean user input before using it or storing the data into the database. When the necessary steps to sanitize user input are not taken, attackers can inject SQL code or, in our case, inject malicious scripts in place of images that will execute when any user requests a page that contains that “image”.

Our attack exploits an image uploading function in the target website. We submit our script inside of the input for the image URL and the website stores it inside of its database. Then, when users request the page that contains the image, the script will automatically run in the user's browser.

```
23 function formSubmitHandler(event) {  
24     event.preventDefault();  
25     const userMessageInput = event.target.querySelector('textarea');  
26     const messageImageInput = event.target.querySelector('input');  
27     const userMessage = userMessageInput.value;  
28     const imageUrl = messageImageInput.value;  
29  
30     if (  
31         !userMessage ||  
32         !imageUrl ||  
33         userMessage.trim().length === 0 ||  
34         imageUrl.trim().length === 0  
35     ) {  
36         alert('Please insert a valid message and image.');37         return;  
38     }  
39  
40     userMessages.push({  
41         text: userMessage,  
42         image: imageUrl,  
43     });  
44  
45     userMessageInput.value = '';  
46     messageImageInput.value = '';  
47  
48     renderMessages();  
49 }
```

This screenshot shows the functionality that handles image submission. On line 30 we can see that the program is simply checking that **any** value has been entered. If the input passes this check, the input is pushed directly into the database. This makes the website vulnerable to a cross-site script attack.

Description and diagram of the attack

While a majority of the site uses the handlebars framework which has in-built XSS protection there is a section of the site that does not seem to utilize handlebars and thus does not have similar protections. The attack works by first gaining access to a manager's account, be it through a phishing email or any other method and then accessing the global announcement feature.

The image shows a dark-themed web interface for sending a global announcement. It features a text input field labeled "Your Message", an image input field labeled "Message Image", and a yellow "Send Message" button.

Manager's Global Announcement Page

Due to protections put into place by most modern browsers you will not be able to execute a script through the message box. However because the announcement allows you to attach an image link you can abuse the src attribute by closing it pre-maturely and then adding whatever attribute you want after it.

Your Message

`<script>alert("HACKED")</script>`

Browser protections would stop this script from running once posted ;)

Message Image

Send Message

```

```

Your Message

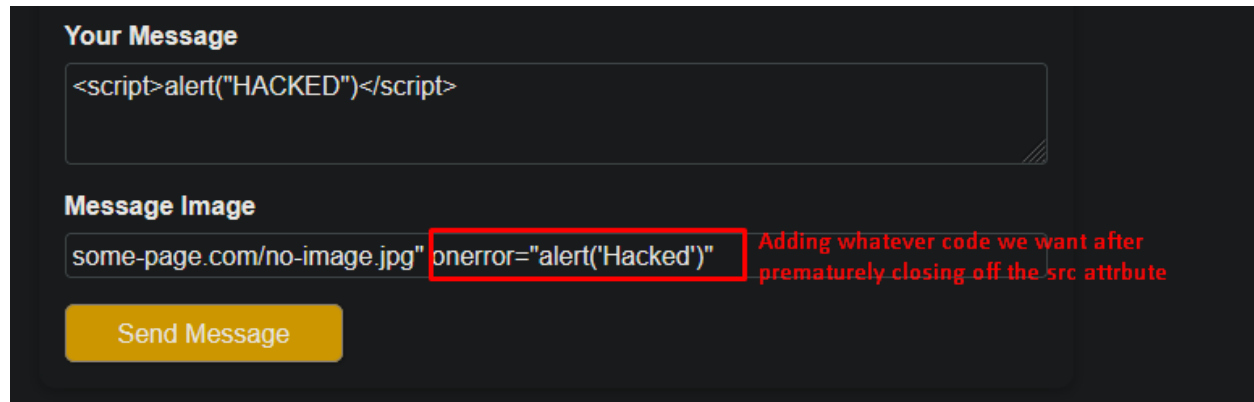
`<script>alert("HACKED")</script>`

Message Image

some-page.com/no-image.jpg" This double-quote closes off the src attribute and allows us to inject whatever code we want after it

Send Message

Once we close off the src attribute we can then add any other attribute we want. Since the onerror attribute allows us to run javascript it would be the best method to use.



Your Message

`<script>alert("HACKED")</script>`

Message Image

`some-page.com/no-image.jpg" onerror="alert("Hacked")"`

Adding whatever code we want after prematurely closing off the src attribute

Send Message



This exploit pattern is not caught by the browser and is sent off to a database to be propagated to all online employees.

Signature of the attack

Cross-site script attacks are a type of injection attack in which malicious code is injected into websites wherein the browser loads and executes the malicious script. This attack is made possible through a vulnerable codebase that does not have proper input validation. It is difficult to detect most instances of cross-site scripts because the malware itself is not stored on the webserver and thus there is nothing for malware scanners to analyze. It is, however, possible to run a malware detection system that relies on scanning files for malware attack signatures. So although the cross-site script attack itself does not have a signature that can be used to detect it, it is possible to use signature-based malware scanners that are capable of evaluating websites and detecting known attack signatures, a characteristic byte pattern that is used in malicious code. However, most malware scanners will not detect new malware or scripts that have no attack signatures. Therefore, although cross-site scripts may contain characteristic attack signatures, the scripts themselves do not contain signatures that can be used to detect the attack.

In order to effectively detect cross-site script attacks, a comprehensive security solution is needed. This includes evaluating the webpage for any objects the browser may open, for example, any documents that contain HTML links within the page itself. There must also be a frequent and comprehensive analysis of all objects that are located on the web server to ensure there are no malicious links or known attack signatures that are embedded. A full manual behavioral analysis may be onerous, but it may be the most surefire way to detect any harmful or malicious scripts that may be contained within the webpage. This is done by executing each object on the website to test for malicious actions, as well as inputting basic cross-site scripts to test if user inputs have been exhaustively sanitized. Lastly, it is important to monitor the network for malware activity and to check the logs frequently.

Example of a log of a cross-site script attack inside the HTTP client request processed by the server:

```
10.0.0.1 - - [15/Mar/2018:14:47:24 +0000] "GET  
/vulnerabilities/xss_r/?name=%3Cscript%3Ealert%28%27xss%27%29%3B%3C%2Fscript%3E  
HTTP/1.1" 200 1715 "http://10.0.0.1/vulnerabilities/xss_r/" "Mozilla/5.0 (X11; Linux x86_64;  
rv:59.0) Gecko/20100101 Firefox/59.0"
```

Section 4: Security Policy and Controls

According to WhiteHat's security statistics report in which the prevalence of cross-site scripts was tracked from 2012 to 2015, it was found that nearly 50% of vulnerabilities detected on websites are cross-site scripting vulnerabilities. This statistic establishes cross-site scripting vulnerabilities' as a consistent contender for top three vulnerabilities detected across the Internet. The prevalence of these vulnerabilities should painstakingly remind us that protection against it is invaluable, and that most of us are not properly safeguarding ourselves across such attacks. There are steps we must take in order to prevent our systems from being compromised. In the case of cross-site scripting attacks, we know that the vulnerability exists within the injection of scripts and data the attacker uses in order to manipulate the logic of the web application. To prevent such injections, we must check all data for standardization of types, syntax, and semantics in order to restrict output from being interpreted as Javascript by users' browsers.

Steps taken for prevention:

1. Enforce data types for all input fields

User inputs come in the form of strings that we must transform into usable objects. To ensure we are receiving proper input, we can force the input to be specific data types, and for the specific data types to be required parameters within our framework. When a type mismatch occurs, the system should throw an error before any form of control is transferred from the framework over to the application.

2. Validate data received from inputs

Analyze all input data received for appropriate grammar and semantics. For example, the year of birth should have four numeric values (grammar), and should be greater than 1900 (semantics). To ensure standardization of inputs, a whitelist could be created of validation that contains rules with specific accepted patterns.

3. Sanitize data being converted to outputs

Analyze outputs to ensure data cannot be passed to the HTML document so that browsers are properly treating output as data and not code. Specifically, attributes of XML and HTML should be converted into safe variants, such as & to & and < to <. Doing so also protects the system against SQL injections and other vulnerabilities that

handle URLs and path directories.

4. Header level protection

Within headers, we are able to specify the Content-Type, and thus we can ensure the content type is not of type "text/html". We can also prevent the browser from automatically detecting the data type by passing "X-Content-Type-Options:nosniff" to the header.

5. Frequently use appropriate testing tools

There are open-source and enterprise level testing tools that are available. These testing tools are generally capable of detecting vulnerabilities within the application source code itself, and depending on the efficacy of the tool, some could point to the exact line of code that introduces the vulnerability.

6. Manually test for vulnerabilities on the front-end

This may be tedious but the most surefire way to ensure all previous steps are implemented and working properly. To do this, we could insert test strings with HTML and Javascript into our application, anywhere an input exists, whether it may be forms, URL parameters, or even hidden inputs.

7. Write unit tests for data outputs

Although it may not be feasible to write unit tests for every function that handles user input, it may be worthwhile to write some in order to check if the appropriate whitelisted patterns are properly sanitizing data output.

In terms of cross-site scripting attacks, prevention and protection is also the solution. If a vendor were to be attacked using a cross-site script, the best action would be to apply the steps of prevention so that the attack cannot be executed by the browser anymore. However in the case of a persistent or stored cross-site script attack, where the executed script persists on the browser or resides in the web server or database, then the malicious script must be found and eliminated.

In order to find the malicious script inserted into the web server or database, the first step would be to look through any logs such as access logs, activity logs, and error logs. To ensure

this is possible, it is important to enable detailed logging of web server activities at all times. It would be impractical to look through every single line inside the logs, so we must filter out the logs for POST requests and access to admin pages. POST requests typically come from form submissions, which is a key entry point for cross-site script attacks. After the results are filtered, the only thing that can be done is looking through the logs for unknown IP addresses that successfully gained access to any admin page. Once an unknown IP address is found, we can then look at logs specifically pertaining to the unknown IP address in order to see a timeline of their actions and to discover the paths that have been accessed. Once all this information is known, it is possible to look into accessed paths to remove records that contain the maliciously injected commands.

Conclusion

In summary, this report has gone into the details of the nature of cross-site scripting attacks and what can be done to protect against it. In order to determine if a site is vulnerable to a cross-site script attack, you can inspect the page on your browser and see if the developers have left the javascript files exposed and unminified. If so, then it is easy to locate the functions that handle user input and determine if the code sanitizes user input before storing it or using it elsewhere. If no sanitization is detected, then the website is vulnerable to a cross-site script attack.

Since cross-site scripting targets deficiencies in source code, no specific hardware is needed to execute the attack. The only requirement is that the attacker has access to a web browser, the operating system does not matter.

To demonstrate this attack, our team built a website that runs on Node.js, Express, and Postgres. The functionality that handles user form submission intentionally does not sanitize the data being passed in order to show how a cross site script can take advantage of deficient code. In our case, an attacker could inject part of an HTML img tag that contained a malicious script in place of an actual image. This image would then be stored in the database and served to any user that navigates to the page where the image is used.

Detection of cross-site scripting is difficult because there is technically no signature of a cross-site script attack. Proper detection would require malware detection that can search through databases and web pages to search for known byte patterns typically used in malicious code. Protection against cross-site scripting requires a comprehensive approach that covers multiple areas. Mainly, user inputs must be thoroughly tested to ensure that proper sanitization has been implemented. Developers can use data typing, data validation, sanitization, testing, and header level protection to protect against cross-site script attacks. Additionally, regular scanning and testing of web elements can be done to ensure that no malicious code has already been injected.

Bibliography

- <https://www.acunetix.com/blog/articles/using-logs-to-investigate-a-web-application-attack/>
- <https://www.ptsecurity.com/ww-en/analytics/knowledge-base/what-is-a-cross-site-scripting-xss-attack/#5>
- <https://www.google.com/about/appsecurity/learning/xss/>
- <https://snyk.io/blog/xss-attacks-the-next-wave/>
- <https://www.lastline.com/blog/cross-site-scripting-attack/>
- <https://www.netscout.com/what-is/attack-signature>
- <https://owasp.org/www-community/attacks/xss/>