

Sri Lanka Institute of Information Technology



## **Bug Bounty - Report 03**

### **DOM-Based XSS Vulnerability / PII Disclosure**

**`myessaywriter.ai/grammar-checker`**

Student Name – Wanasinghe N.K

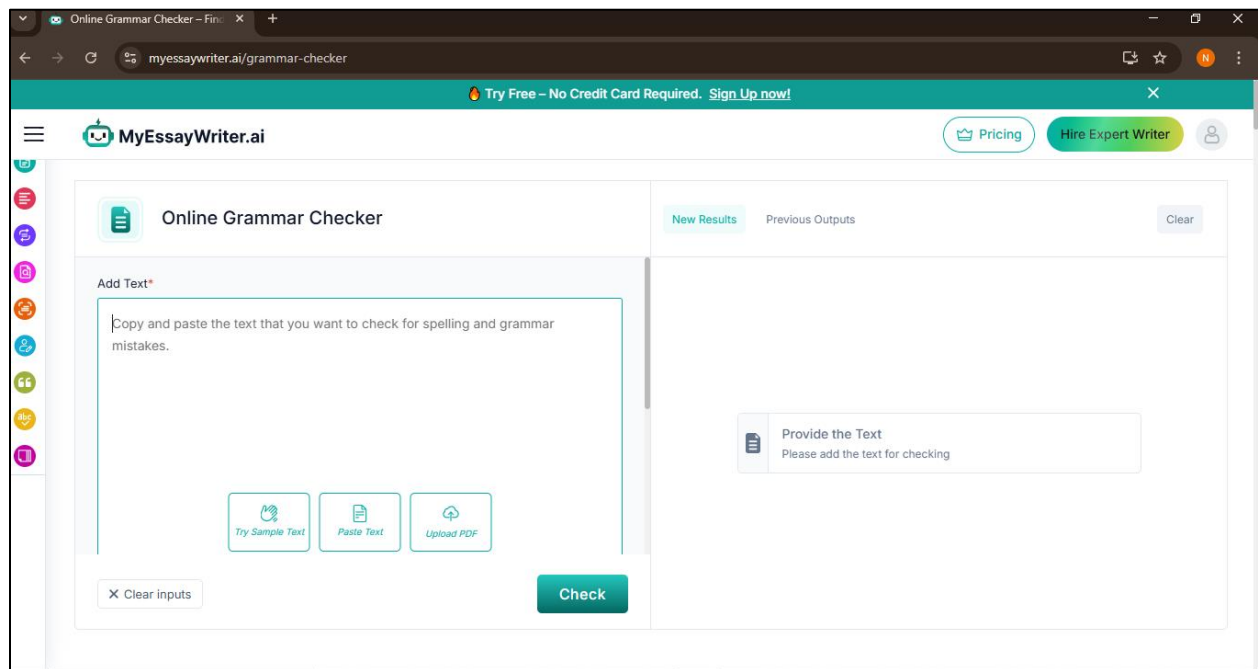
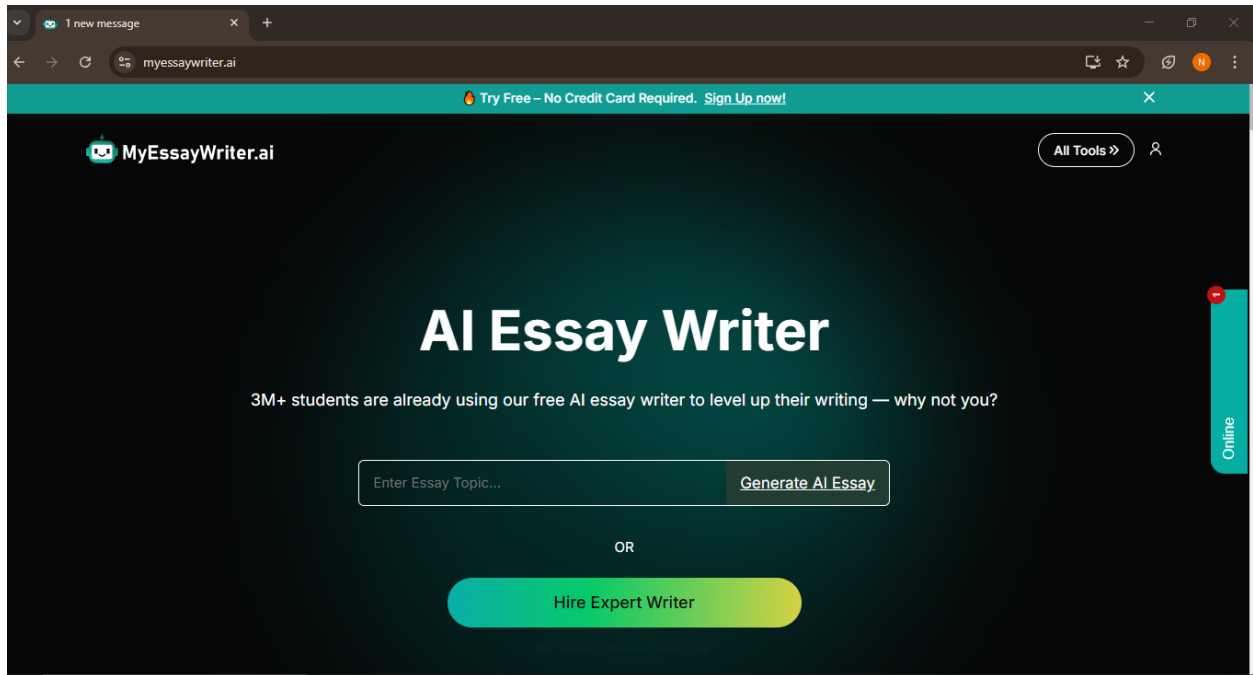
Student ID – IT23221000

**IE2062 - Web Security**

B.Sc. (Hons) in information Technology Specializing in Cyber Security

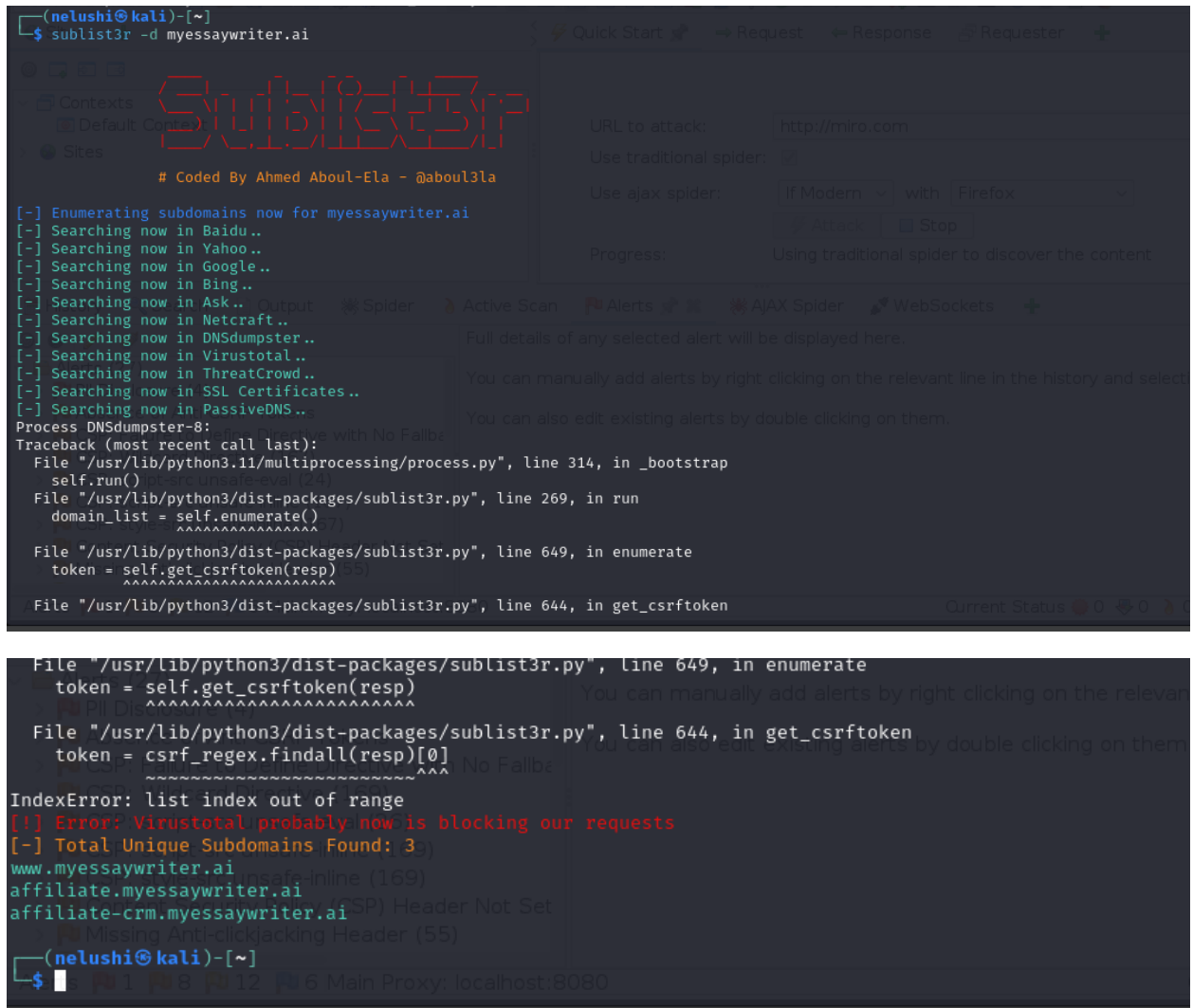
# Report 03 – myessaywriter.ai/grammar-checker

Main domain – <https://www.myessaywriter.ai/grammar-checker>



## Reconnaissance: Gather information about the target.

I used **sublist3r** tool to find the subdomains of humanizeai.io.



```
(nelushi@kali)-[~]
$ sublist3r -d myessaywriter.ai

# Coded By Ahmed Aboul-Ela - @aboul3la

[-] Enumerating subdomains now for myessaywriter.ai
[-] Searching now in Baidu..
[-] Searching now in Yahoo..
[-] Searching now in Google..
[-] Searching now in Bing..
[-] Searching now in Ask..
[-] Searching now in Netcraft..
[-] Searching now in DNSdumpster..
[-] Searching now in Virustotal..
[-] Searching now in ThreatCrowd..
[-] Searching now in SSL Certificates..
[-] Searching now in PassiveDNS..
Process DNSdumpster-8:
Traceback (most recent call last):
  File "/usr/lib/python3.11/multiprocessing/process.py", line 314, in _bootstrap
    self.run()
  File "/usr/lib/python3/dist-packages/sublist3r.py", line 269, in run
    domain_list = self.enumerate()
  File "/usr/lib/python3/dist-packages/sublist3r.py", line 649, in enumerate
    token = self.get_csrf_token(resp)
  File "/usr/lib/python3/dist-packages/sublist3r.py", line 644, in get_csrf_token
    token = csrf_regex.findall(resp)[0]
IndexError: list index out of range
[!] Error: Virustotal probably now is blocking our requests
[-] Total Unique Subdomains Found: 39
www.myessaywriter.ai
affiliate.myessaywriter.ai
affiliate-crm.myessaywriter.ai
Missing Anti-clickjacking Header (55)
```

## Nmap – Network scanning and enumeration

I found all the open ports and detected the running services on the target server using Nmap.

```
(nelushi@kali)-[~]
$ nmap myessaywriter.ai -vv

Starting Nmap 7.95 ( https://nmap.org ) at 2025-05-01 14:53 CDT
Initiating Ping Scan at 14:53
Scanning myessaywriter.ai (54.80.29.27) [4 ports]
Completed Ping Scan at 14:53, 0.01s elapsed (1 total hosts)
Initiating Parallel DNS resolution of 1 host. at 14:53
Completed Parallel DNS resolution of 1 host. at 14:53, 0.08s elapsed
Initiating SYN Stealth Scan at 14:53
Scanning myessaywriter.ai (54.80.29.27) [1000 ports]
Discovered open port 443/tcp on 54.80.29.27
Discovered open port 21/tcp on 54.80.29.27
Discovered open port 80/tcp on 54.80.29.27
Discovered open port 22/tcp on 54.80.29.27
Completed SYN Stealth Scan at 14:53, 13.28s elapsed (1000 total ports)
Nmap scan report for myessaywriter.ai (54.80.29.27)
Host is up, received reset ttl 255 (0.038s latency).
rDNS record for 54.80.29.27: ec2-54-80-29-27.compute-1.amazonaws.com
Scanned at 2025-05-01 14:53:26 CDT for 14s
Not shown: 996 filtered tcp ports (no-response)
PORT      STATE SERVICE REASON
21/tcp    open  ftp     syn-ack ttl 64
22/tcp    open  ssh     syn-ack ttl 64
80/tcp    open  http    syn-ack ttl 64
443/tcp   open  https   syn-ack ttl 64

Read data files from: /usr/share/nmap
Nmap done: 1 IP address (1 host up) scanned in 13.59 seconds
Raw packets sent: 2008 (88.296KB) | Rcvd: 16 (648B)
```

## Amass – Subdomain and DNS mapping

I found all the subdomains related to the target domain using Amass.

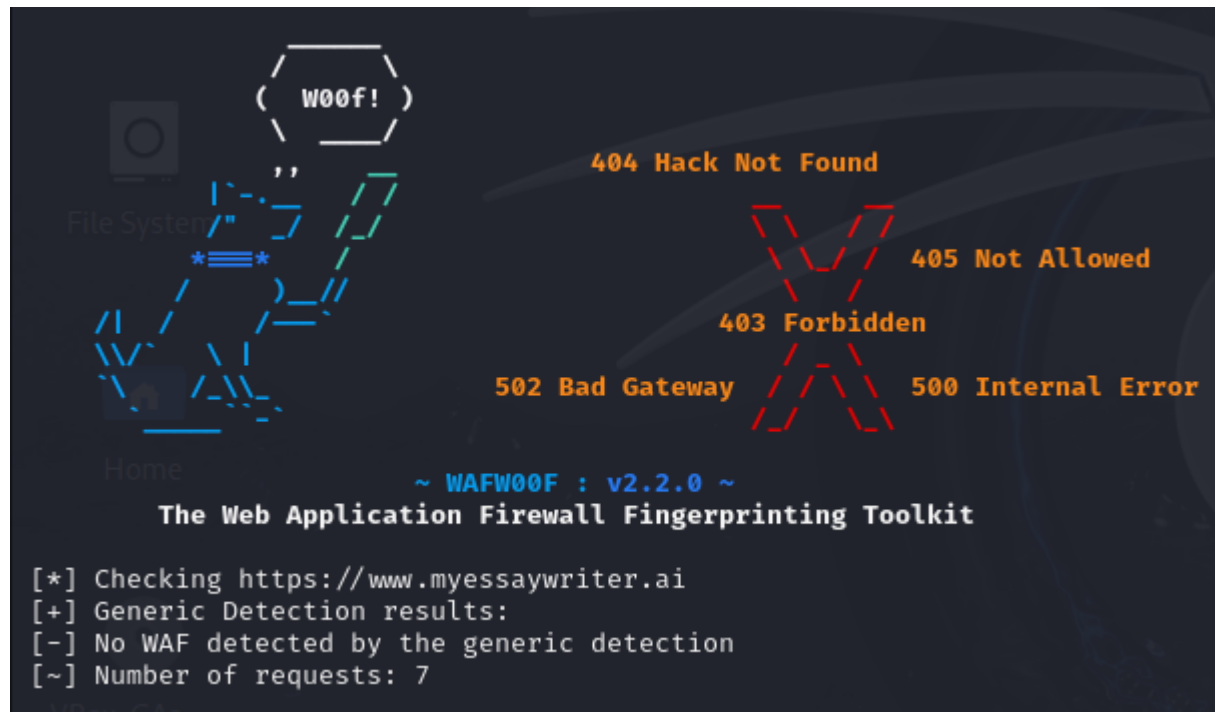
```
(nelushi@kali)-[~]
$ amass enum -d myessaywriter.ai -rfsgl-13

myessaywriter.ai (FQDN) → ns_record → ns-468.awsdns-58.com (FQDN)
myessaywriter.ai (FQDN) → ns_record → ns-1485.awsdns-57.org (FQDN)
myessaywriter.ai (FQDN) → ns_record → ns-656.awsdns-18.net (FQDN)
myessaywriter.ai (FQDN) → ns_record → ns-1546.awsdns-01.co.uk (FQDN)
ns-1485.awsdns-57.org (FQDN) → a_record → 205.251.197.205 (IPAddress)
ns-1485.awsdns-57.org (FQDN) → aaaa_record → 2600:9000:5305:cd00::1 (IPAddress)
205.251.192.0/21 (Netblock) → contains → 205.251.197.205 (IPAddress)
2600:9000:5300::/45 (Netblock) → contains → 2600:9000:5305:cd00::1 (IPAddress)
16509 (ASN) → managed_by → AMAZON-02 - Amazon.com, Inc. (RIROrganization)
16509 (ASN) → announces → 205.251.192.0/21 (Netblock)
16509 (ASN) → announces → 2600:9000:5300::/45 (Netblock)
ns-1546.awsdns-01.co.uk (FQDN) → a_record → 205.251.198.10 (IPAddress)
ns-1546.awsdns-01.co.uk (FQDN) → aaaa_record → 2600:9000:5306:a00::1 (IPAddress)
205.251.192.0/21 (Netblock) → contains → 205.251.198.10 (IPAddress)
2600:9000:5300::/45 (Netblock) → contains → 2600:9000:5306:a00::1 (IPAddress)

The enumeration has finished. (ctrl key) before using this command
```

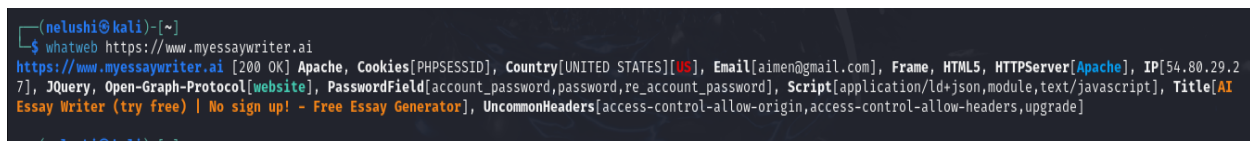
## Wafw00f – Firewall Detection

Command used – wafw00f <https://www.myessaywriter.ai/>



**Whatweb** – to identify the technologies used by the site.

Commans used – whatweb <https://www.myessaywriter.ai/>



# Vulnerability 01

## Domain

<https://www.myessaywriter.ai/grammar-checker>

## Vulnerability title

DOM-Based XSS Vulnerability

## Vulnerability description

DOM-Based Cross-Site Scripting Vulnerability

Cross-Site Scripting (XSS) describes a security vulnerability where an attacker injects malicious scripts that can be executed inside a user's browser.

DOM-based XSS is a client-side exploit that occurs whenever the browser executes untrusted data as code due to poor JavaScript sanitization.

This means that the malicious input is never sent to the server, instead being completely handled and executed in the user's browser by means of DOM manipulation via JavaScript.

This means that when the client-side script reads data from sources such as the URL and writes it back to the page without any sanitization or proper encoding, it opens the application to this vulnerability.

This type of XSS execution occurs only on the client's browser when the victim visits a malicious URL or interacts with some elements created using unsanitized DOM data.

It also often requires social engineering to trick that victim into clicking a specially crafted link or opening a manipulated page.

### How DOM Based XSS works

An attacker creates a malicious URL containing a script payload in one of the parameters.

For example,

[https://example.com/profile#<script>alert\('XSS'\)</script>](https://example.com/profile#<script>alert('XSS')</script>)

Client-side JavaScript reads the fragment (location.hash) and injects it directly into the page using innerHTML like,

```
document.getElementById("info").innerHTML = location.hash;
```

Which leads to the following HTML,

```
<div id="info"><script>alert('XSS')</script></div>
```

As a result, the script is executed in the context of the victim's session, giving the attacker the ability to,

- Steal cookies or local storage data
- Redirect the victim to a malicious site
- Manipulate the user interface
- Run unauthorized commands.

## **Affected components**

The **grammar checker input field** passes unescaped user input into the **<div id="generatedEssay-2">** via innerHTML, enabling DOM-based XSS.

## **Impact assessment**

### ***Severity – High***

- Arbitrary JavaScript Execution

Attackers can inject scripts into the victim browser, which can then lead to different exploits.

- Session Hijacking

Attackers steal session cookies to impersonate a user using document.cookie

- Credential Theft

Fake login forms or keylogging scripts can collect username and passwords of users.

- User Redirection to Malicious Sites

Users can be tricked into visiting phishing or malware sites.

- Browser Exploration

Malicious scripts may exploit browser-specific vulnerabilities, giving a way to more serious attacks.

- Denial of Service

Through infinite loops, alerts, or DOM manipulations, scripts can crash the browser.

- Damage User Trust

Suspicious activities like popups, sliding windows, and redirects reduce the credibility of the platform.

- Bypassing Client-Side Validations

Attackers will manipulate client-side checks that can lead to security or logic being bypassed.

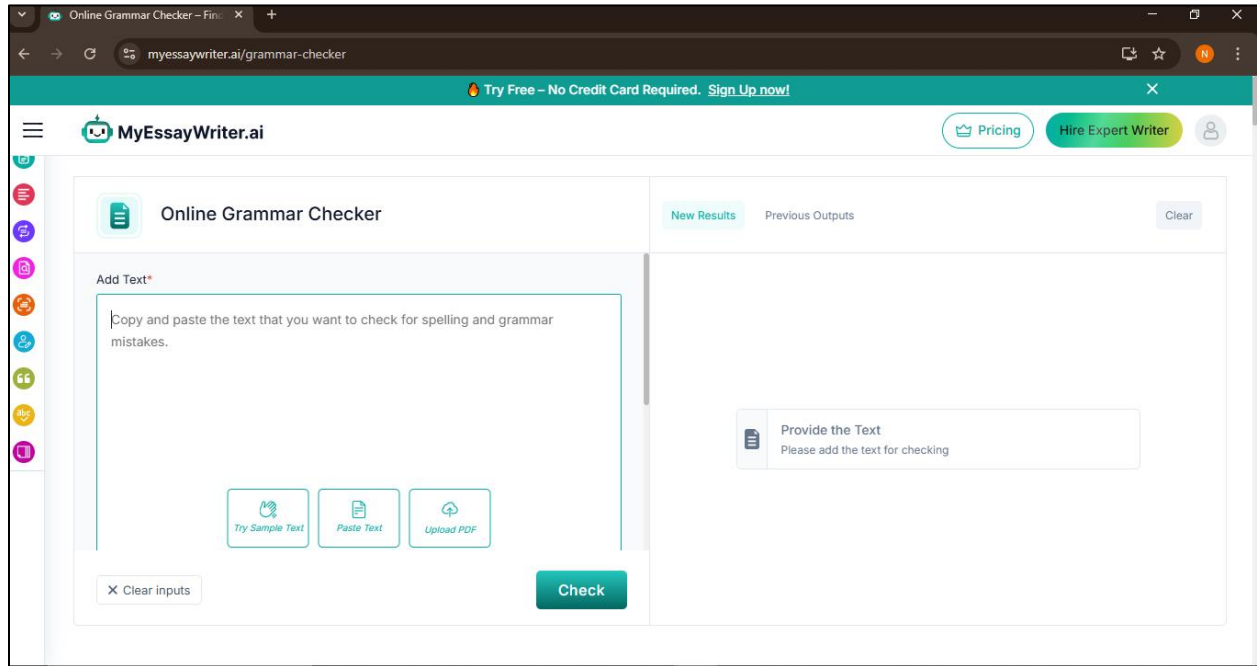
- Targeted Attacks (Spear Phishing)

Spear-phishing is when a personalized payload is engineered for a specific target to increase success rates.



## Steps to reproduce with Proof of Concept (poc)

1. First, I navigated the vulnerable site (myessaywriter.ai) and found the text input field in its grammar checker platform.

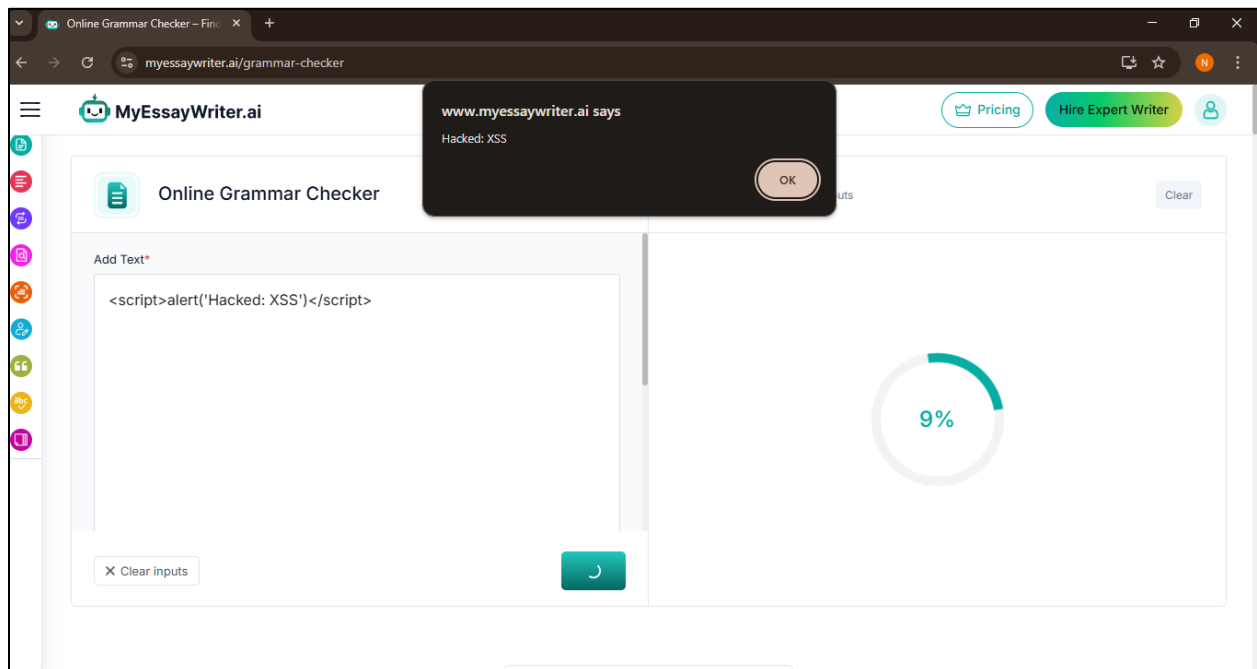


## Reflection

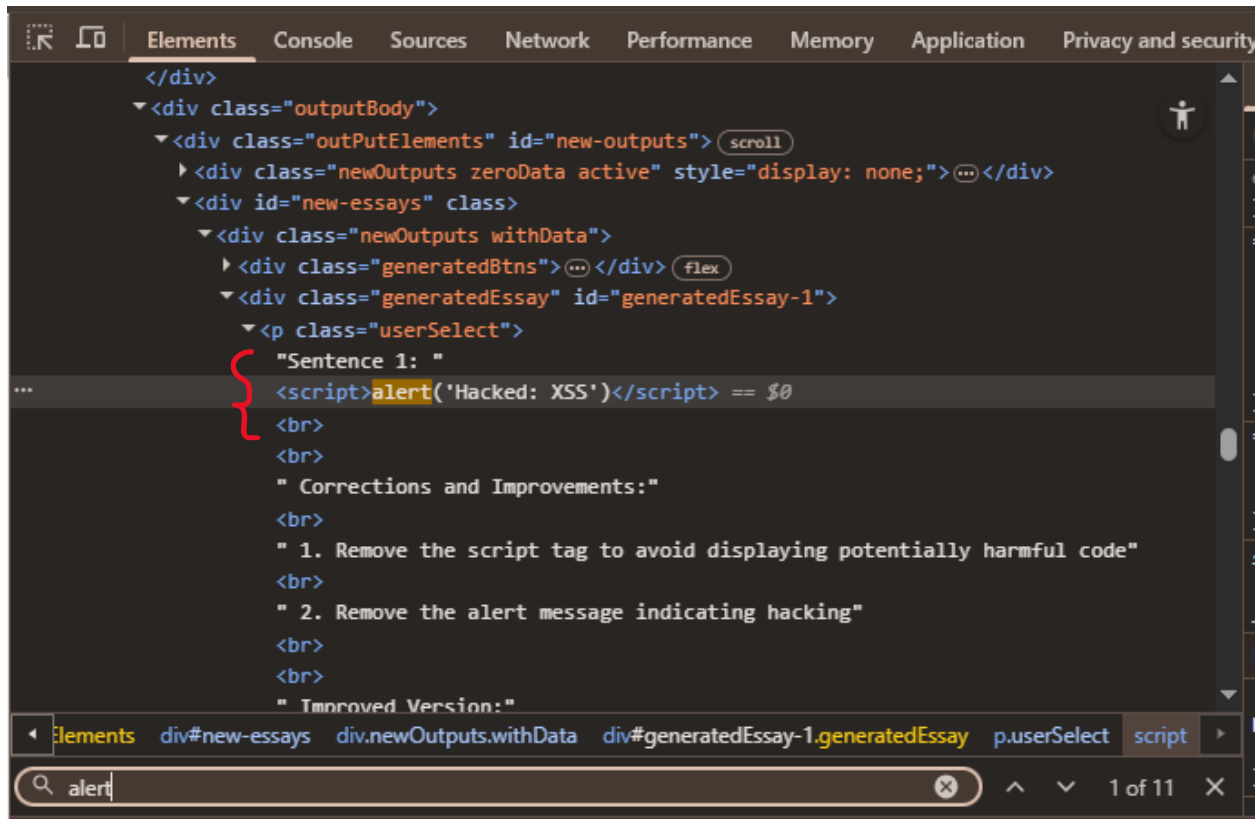
The site loads normally, and it has a large input text field and a button “Check” to process the input.

2. I tried to inject a basic XSS payload like this and clicked the check button.

`<script>alert('Hacked: XSS')</script>`



I checked the source code as well to see if it's being reflected.



### Observation

- An alert pops up with the message 'Hacked: XSS'.
- So the `<script>` tag was **not sanitized or filtered**.

### Reflection

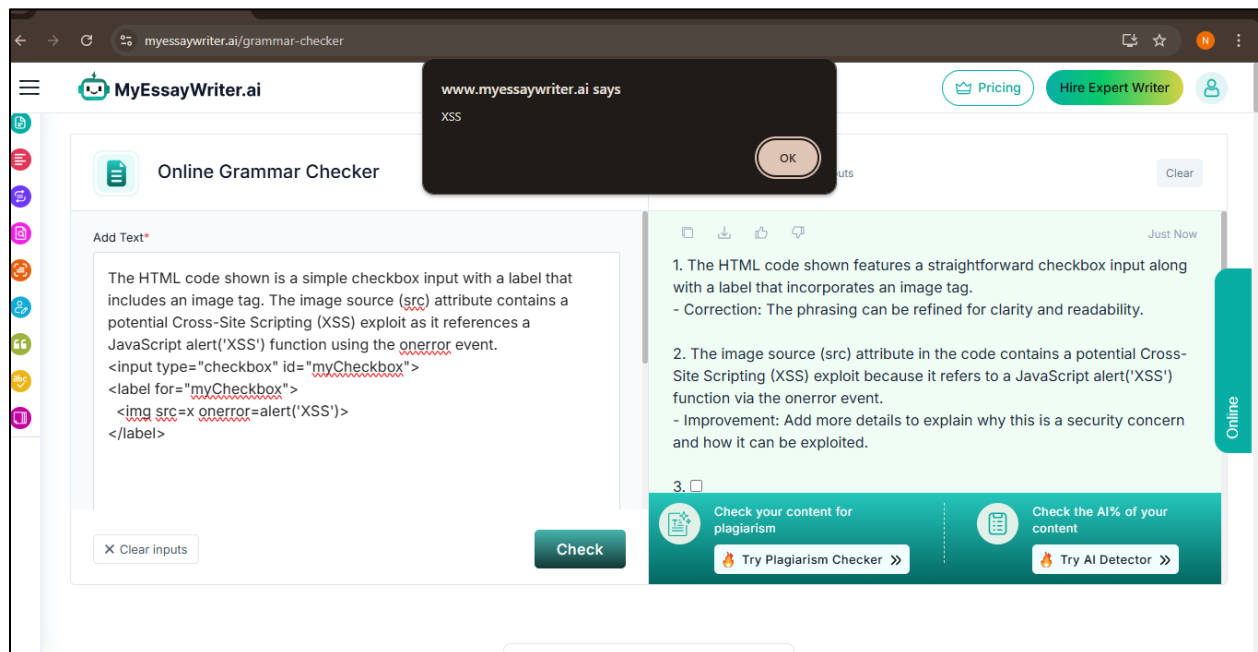
- This is a Reflected XSS.
- `<script>` injections are the **most common and well known** and modern apps should block or sanitize them.
- Since this payload executed, the input is being inserted directly into the HTML without any **escaping, encoding or filtering**.

### Security insight

This is a big security hole, you can inject and execute arbitrary JavaScript code and hijack sessions, steal cookies or redirect.

3. Next, I tried an alternative image-based payload with JavaScript in an event handler that bypasses filtering.

```
<input type="checkbox" id="myCheckbox">
<label for="myCheckbox">
  <img src=x onerror=alert('XSS')>
</label>
```



## Reflection

I got a pop-up alert saying “XSS”.

This alert proves that the **input is reflected unsanitized**, and javascript was successfully executed.

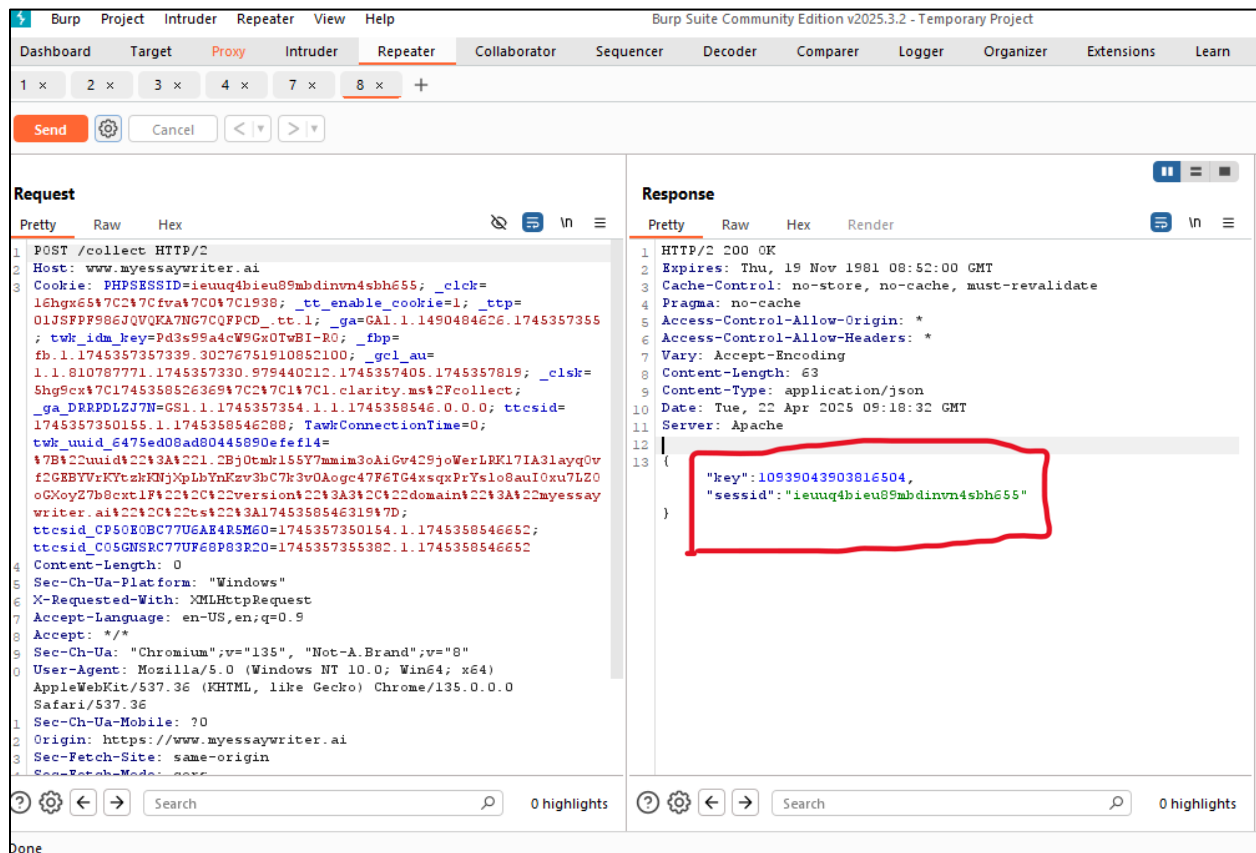
This is a clever way to exploit XSS, **using onerror** event of the `<img>` tag, executes JavaScript `alert('XSS')` when the image fails to load because `src=x` is invalid.

The HTML I inserted would have been parsed directly into the DOM with no sanitization for tags like `<img/>` or for attributes like `onerror`.

## Security Insight

This is more serious because **multiple injection vectors are working** — both classic `<script>` tags and more creative HTML-based payloads. This means the site is missing **output encoding**, **content security policy (CSP)** and **input validation**.

4. I used **Burp Suite** checked the HTTP response of the request by sending the request to the repeater.



## Reflection

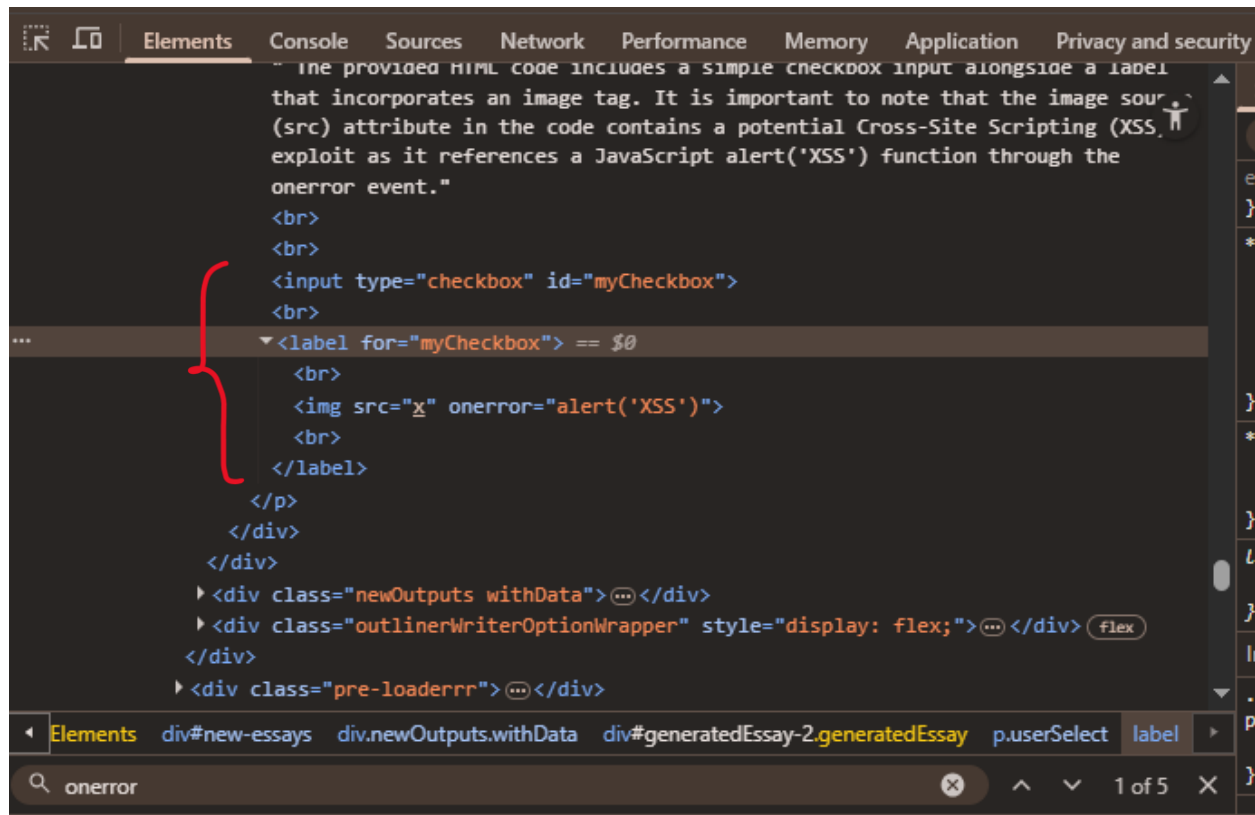
The final payload does not appear in the HTTP response. Instead, I got an HTTP/2 200 OK response, and headers like Cache-Control: no-store, indicating no caching of the response.

The body of the response contains a JSON object with a key and session ID. If those sensitive data are exposed, that would be vulnerable.

**This is a PII Disclosure.**

If these are not sufficiently secured (encrypting sessions, secure cookies, session rotation), identifiers such as session IDs, or session keys can be sniffed by an attacker and used to hijack an active session or impersonate a user.

5. I checked the site's source code to see whether the payload is stored.



### Reflection

It is stored inside an **id** called **genetaedEssay-2**.

I observed that whenever I inject a payload, it will be shown in the source code but when I refresh the page it will be removed.

### How I decided that this is a DOM-Based XSS

- ✓ The payload executes directly in the browser without any interaction with the server - it only makes use of client-side JavaScript.
- ✓ This payload is not reflected in the HTTP response - the server does not reflect input but manipulates the DOM using the untrusted input through client-side JavaScript.
- ✓ The script runs in the browser - XSS is triggered by client-side code in the DOM.
- ✓ The script persists only during the current session - it does not require storing on the server and disappears while refreshing the page.

IS NOT:

Stored XSS→ It does not last after reloading the page.

Reflected XSS→ in this case it is client-side DOM that triggers execution and not the server reflection direct.

DOM-based XSS is classified under **OWASP 2021 A03: Injection** and **WSTG-v42-INPV-01**. It occurs when untrusted data (usually from the URL or other sources) is processed by client-side JavaScript and inserted into the DOM without sanitization. The payload gets executed on user interaction with the page, typically affected by malicious links or crafted URLs. Attackers can, therefore, use this vulnerability to steal information, manipulate the DOM, or perform unauthorized actions on the victim's browser.

## **Proposed Mitigation or Fix (DOM-based XSS)**

First, validate and sanitize your inputs on the client and server sides, inspect those inputs coming from the URL, document.location, or document.referrer names since they can never be trusted and must be sanitized before using them.

Avoid using dangerous JavaScript methods like eval(), innerHTML, document.write(), and setTimeout(string), include textContent and setAttribute.

Encode output (HTML, JavaScript, URL) into the DOM and do not enter raw input into it.

Tools like DOMPurify can be trusted to clean and do safe injection of user-generated content into the DOM.

Implementing a strong content security policy could help greatly in restricting script execution and blocking inline and unauthorized scripts.

Cookies should be created with HttpOnly, Secure, and SameSite flags to reduce their accessibility for JavaScript and lower the risk of session theft.

Adopt secure frameworks for the front end such as React or Angular, which automatically escape output and reduce the chances of introducing DOM-based XSS vulnerabilities.

## Ethical Note

As a student who learns the importance of the security of web applications, I value ethical hacking and responsible disclosure. The test did no damage, defacement, or unauthorized access.

I have strictly followed the principle, "Do no harm," respectfully and ensured that the vulnerabilities found were documented and responsibly disclosed this finding to the site via email, including a detailed proof of concept (PoC) and a respectful explanation. No malicious actions were taken.

This corresponds to global standards of respectability in education and ethics with respect to cybersecurity.

