# XSS Vulnerability Test Enhancement for Progressive Web Applications

Josep Pegueroles Valles(✉), Sebastien Kanj Bongard, and Arnau Estebanell Castellví

Universitat Politècnica de Catalunya, Jordi Girona 1-3, 08034 Barcelona, Spain
{josep.pegueroles,sebastien.kanj}@upc.edu

**Abstract.** Progressive Web Applications produce false negative results when scanned with security vulnerability scanners. In this paper the authors investigate the causes behind vulnerability scanners missing simple vulnerabilities when being used on Progressive Web Applications (PWAs).

Moreover, an analysis of the caveats of only having fully automated vulnerability scans and manual pentests, without a semi-automatic tool covering the gap between the two, will be performed. An explanation of how such tool has been built will be delivered at the end of the paper.

**Keywords:** Progressive Web Application · security · vulnerability scanners · XSS · sqli · false negatives

## 1 Introduction

During a penetration test against an ecommerce website, the authors of this paper noticed that Qualys [1], a commercial web application scanning tool which ranks in the first positions of the Gartner ranking [2], had missed the most basic Cross-Site Scripting (XSS) vulnerability. In order to exploit this vulnerability you just needed to use a non-complex XSS payload, inserting '<img src = x onerror = javascript:alert(1) >' in the input field of the search functionality of the main page.

The first idea of the researchers was to rerun the scan. Surprisingly, the scan missed it again. The situation did not change regardless of how many times we repeated the experiment. That was an unauthenticated scan, so there was no reason for Qualys to miss such a simple vulnerability. We tried getting some logs from Qualys in order to figure out what was going on. Unsurprisingly, the logs returned by Qualys were very limited and did not provide enough information to discern what was happening between the web application scanning tool and the Progressive Web Application. Moreover, as this was a scan performed to an external company we did not have access to the machine hosting the application to retrieve the logs directly from the apache2 or nginx log directory.

Lastly, we realized that when thinking about the penetration test as a whole, at that moment he had no semi-automatic way of testing for XSS vulnerabilities. If Qualys was missing one XSS in the main page it was probably missing even more XSS vulnerabilities


S. Goel and P. R. Nunes de Souza (Eds.): ICDF2C 2023, LNICST 571, pp. 152–163, 2024.
https://doi.org/10.1007/978-3-031-56583-0_10

in the rest of the webpage. How could we test all the input fields without spending too much time when Qualys was not working properly?

The need for a semi-automatic tool that could allow the pentester to hunt for XSS vulnerabilities while allowing him to tweak parameters or define payload dictionaries in order to make sure all vulnerabilities were found became a reality.

## 2 About Progressive Web Applications

Before Progressive Web Apps (PWAs) were a reality, business looking to build a webpage with fast loading times, high performance and deep functionality looked at native apps. PWAs offer the same rich capabilities with additional benefits. Code for PWAs can be written once and deployed across web, Android and iOS platforms, saving teams time and expenditure.

PWAs are web applications that have been designed so they are capable, reliable and installable. These three pillars transform them into an experience that feels like a platform-specific application. Understanding the Service Worker is crucial in understanding how PWAs work. The introduction of the Service Worker by Google in late 2014 is the heart of modern web application architecture that enables a PWA to deliver content to users faster, even offline.

"The JavaScript Service worker runs in the background separate from the web page and operates as the intermediary between the user and content. It hosts the platform's logic and intercepts requests from the user. A PWA's Service Worker functions as "the dispatcher" of the app, deciding whether to respond to the user with cached content or reach out to the internet for current content." [3].

## 3 The Target

The authors of this paper needed to discover why Qualys had not worked in the first place and then build an alternative tool to use in future pentests where this situation happened again.

The easiest choice was to recreate the scenario that had caused this research to happen. The authors built the same Progressive Web Application found in the pentest (from fingerprinting he knew it was an Open Source solution) and tried to recreate the vulnerable endpoint. The ecommerce website was using the Vue Storefront application [4], which as explained before used a Progressive Web Application architecture.

Their architecture looks like this (Fig. 1):

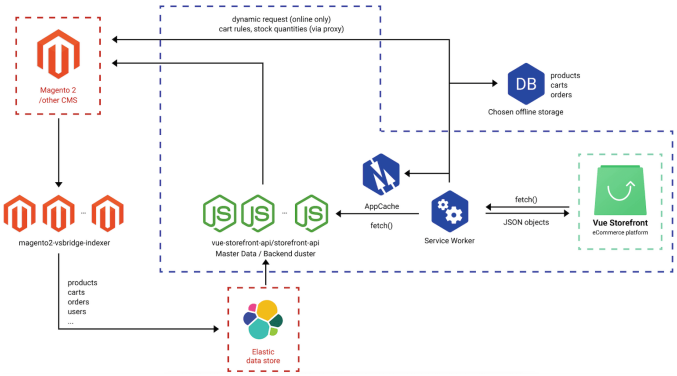


**Fig. 1.** Architecture of the Vue Storefront application.

As can be seen in the picture, Vue Storefront can use Magento 2 or a variety of CMSs as the back-end, and then uses its own engine as the frontend. Interestingly, the application also uses an Elasticsearch data storage and indexer in order to index content and retrieve it in a faster way, and uses workers to communicate the front-end with the back-end.

## 4 The Setup

In order to test the application, the authors recreated the same environment in a server under their control. They took a free open-source template for the Vue storefront application and recreated the vulnerable application found during the pentest. The sever was using a Vue Store application which had a search input where the parameter q was vulnerable to a XSS vulnerability (Fig. 2).

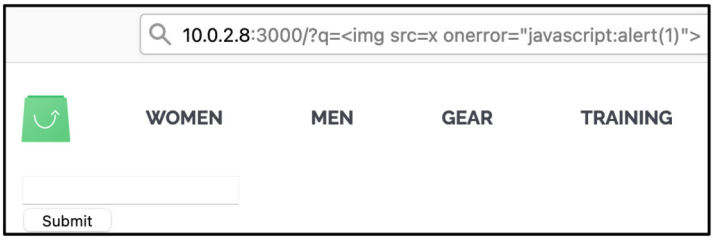


**Fig. 2.** Search box vulnerable to a XSS injection.

When injected, the browser would produce a reflected XSS vulnerability, as the input was provided as a get parameter and was injected directly to the browser after being processed and returned by the server. The request as can be seen in the server is the following (Fig. 3):

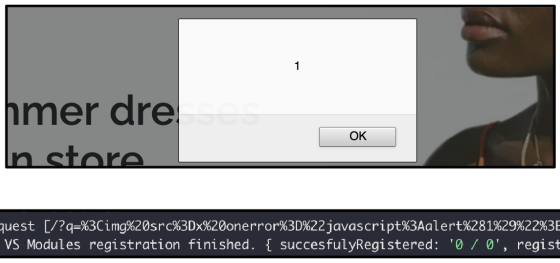


**Fig. 3.** Execution of the XSS vulnerability and Request as seen in the server.

As can be seen, the q parameter is injected with the beforementioned payload, which inserts an alert(1) popup using javascript code. The interesting part is analysing what Qualys and other web vulnerability scanners do when they find a similar scenario.

Once having the application recreated in a testing environment we prepared a Qualys scan and doublecheck that it was still missing the XSS vulnerability. Although Qualys did rank the webpage with a High Severity and when checking the found vulnerabilities one can see lots of reported XSS vulnerabilities (89) (Fig. 4):

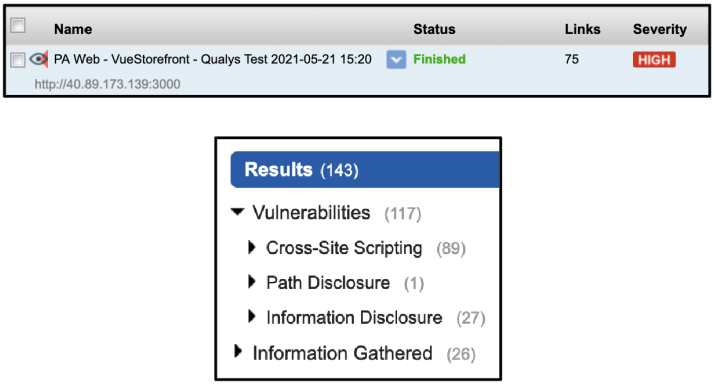


**Fig. 4.** Qualys scan results and List of vulnerabilities found by Qualys.

The truth is that all of them are false positives, as Qualys mistakes them as an Unencoded Characters vulnerability, which is a vulnerability which means that the input you have inputted to the application does appear in the response but does not produce a valid XSS injection. Surprisingly, Qualys still places them in the Cross-Site Scripting category. Once it had been proved that Qualys was missing the XSS vulnerability we gathered the logs from the vulnerable server and analyzed them in order to see what Qualys was doing (Fig. 5).

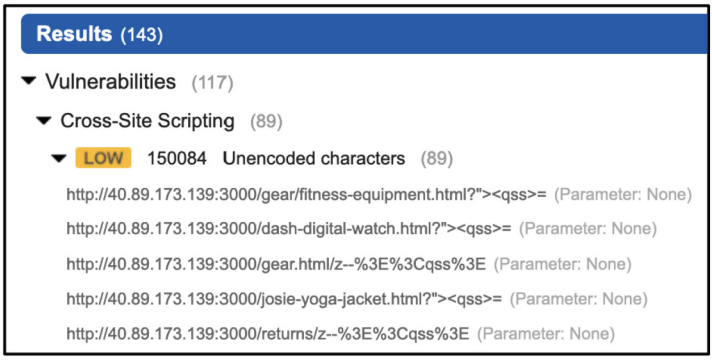


**Fig. 5.** Detail of the vulnerabilities found by Qualys.

## 5 Analysing the Logs

When first reviewing the logs it can be seen that Qualys starts analysing the main page at slash ('/'). And then it starts checking for files like crossdomain.xml, random.html files, and after parsing the pages he stars testing the parameters of the same page. In Fig. 6 you can see the first time Qualys detects the q parameter and sends a 1 to it.

```
whole request [/]: 529ms
```

```
whole request [/?q=1]: 455ms
```

**Fig. 6.** Start of the scan and First time Qualys interacts with the vulnerable parameter.

The next step Qualys does is to enumerate the files within the website, in order to do so it mixes queries which point to endpoints that do not exist and queries to endpoints which do exist but adding or requesting different types of files to enumerate the underlying page in the parameters of the request.

Although not common, during the scan of the application Qualys does find the vulnerable endpoint and tries to inject it in different ways (Fig. 7).

```
1  /?q=
2  /?q=%00%3Cscript%3E_q%3Drandom(X146984300Y1_1Z)%3C%2Fscript%3E
3  /?q=%22%20onEvent%3DX146984300Y1_1Z%20
4  /?q=%22'%3E%3Cqss%20a%3DX146984300Y1_1Z%3E
5  /?q='%20onEvent%3DX146984300Y1_1Z%20
6  /?q=1
```

**Fig. 7.** Payloads sent to the vulnerable endpoint.

The most interesting thing that can be found is that Qualys does consume the '?q=' endpoint 1873 times in a 7 h scan, but still does not send a valid injection payload and hence it does not find the vulnerability. With this information we started analysing the page's source code to see why Qualys was not finding the vulnerability.

There are two reasons why Qualys might have missed the XSS vulnerability. Probably it is not one of the two but a combination of both. The first reason is that from the analysis of the logs Qualys produced we can confirm that Qualys did not try any payload which was indeed valid for the injectable parameter. It is hard to believe that it just "missed it" or that it does not have any valid payload in its dictionary, so probably the reality is that with its custom intelligence, Qualys was not able to detect the reflection of the payload, and hence stopped trying to inject in that specific parameter.

The second reason is that Vue storefront uses a window backbone dispatcher which is basically in charge of receiving the information from the server and rendering it in the front-end. The window dispatcher looks as follow (Fig. 8):

```
<script>
window.__INITIAL_STATE__={version:"",__DEMO_MODE__:!1,config:{server:{host:"0.
0.0.0",port:3e3,protocol:"http",api:"api-search-query",devServiceWorker:!1,
useHtmlMinifier:!0,htmlMinifierOptions:{minifyJS:!0,minifyCSS:!0},
useOutputCacheTagging:!1,useOutputCache:!1,outputCacheDefaultTtl:86400,
availableCacheTags:["attribute","C","category","checkout","compare","error",
"home","my-account","P","page-not-found","product","taxrule"],
invalidateCacheKey:"aeSu7aip",invalidateCacheForwarding:!1,
```

**Fig. 8.** Web dispatcher.

The window dispatcher requests a snapshot of the webpage at the initial request, and based on the user's behaviour and using Javascript, it changes the view of the user only using the client and without the need to request anything from the server.

In our opinion, Qualys just sees the initial response from the server, and as the server is using a dispatcher to render the webpage, and does not initially inject received payloads directly into the HTML code, Qualys does not see the injection happening. Moreover, Qualys probably does not let Javascript from applications run in its testing sandbox, so as the dispatcher is never allowed to render the front page the injection does not actually happen for Qualys.

Moreover, this is totally aligned with the 89 Unencoded characters vulnerability that Qualys has reported. Each time it has tried to inject into the website, the website has returned a response which includes a window dispatcher which in turn includes the injected payload. This is the reason why Qualys sees its input is being reflected in the response, without it confirming a XSS.

It is important to note that this specific behaviour has not only been seen in the firstly analysed webpage, but has also happened in the past in other webpages which were also using dispatchers to render applications.

## 6 Additional Web Vulnerability Scanners

With the conclusions from the previous section, the question that comes to mind is if this is something which is common across vulnerability scanners or if perhaps is a behaviour that one can only observe in Qualys.

Thus, the test that follows is to run the scan with BurpSuite's Active Scan++ [5], a commercial solution from PortSwigger, and with OWASP's ZAP [6], which is advertised as the "most widely used web app scanner".

We started with BurpSuite, and launched a crawl, a passive scan and an active scan to the webpage. When launching the scan, even if it has not finished yet, we observe a similar behaviour than the one found in Qualys. In the Issue Activity from BurpSuite this can be seen as follows (Figs. 9, 10 and 11):

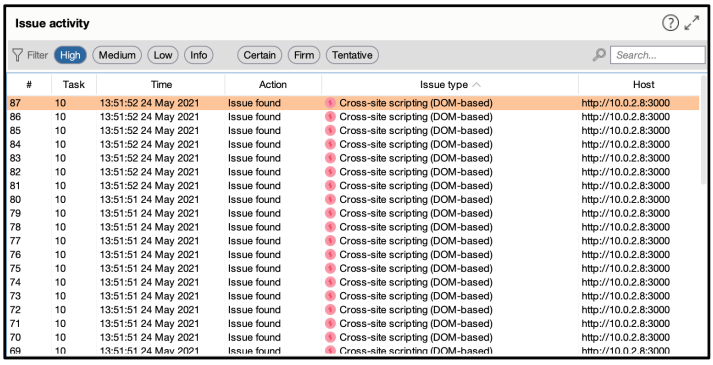


**Fig. 9.** False positive XSS vulnerabilities found by BurpSuite.

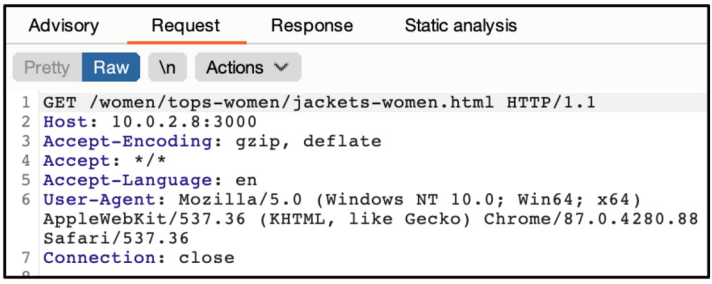


**Fig. 10.** Request of a false positive XSS vulnerability.

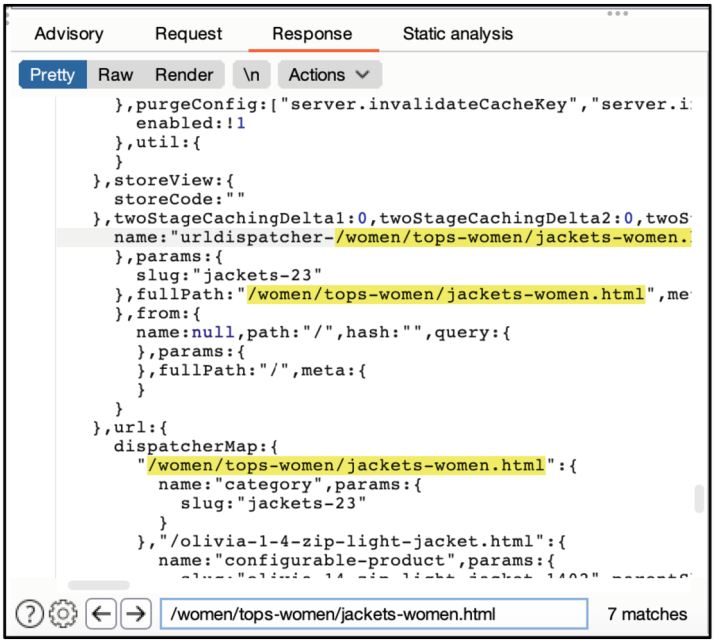


**Fig. 11.** Response that leads BurpSuite to think it was actually a confirmed vulnerability.

When reviewing the outcome of those vulnerabilities one can easily see they are false positives. As it was happening with Qualys, BurpSuite is mistakenly taking the reflection of the URL path in the window dispatcher as a Cross-site Scripting vulnerability.

As BurpSuite sees this behaviour it understands there are many ways to modify the DOM by reflecting whatever is given as the path and reports it as a XSS vulnerability. Nevertheless, what is actually happening is that the application is properly sanitizing whatever is injected in the path or parameters of the request and hence they all are false positives. Once the scan is finished, it is confirmed that BurpSuite does not find the parameter q to be vulnerable.

The last experiment is to rerun the scan using OWASP's ZAP (Fig. 12).

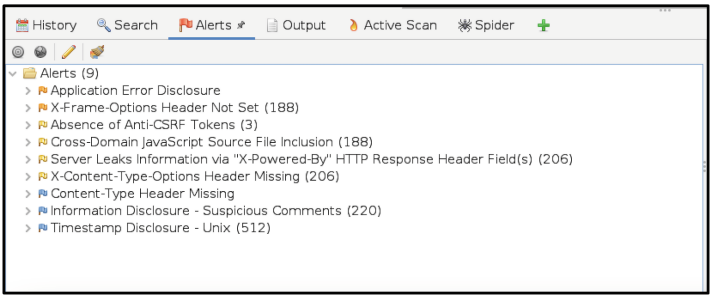


**Fig. 12.** OWASP's ZAP outcome.

Sadly, the opensource solution is the one which performs worse. Not only it has missed the XSS vulnerability, but it has also failed to detect the reflection that is happening from the Path of the URL to the window dispatcher.

## 7 The Intermediate Scan Layer

As explained in Sect. 1, once we realised that we could not trust Qualys to find the majority of XSS vulnerabilities, the need to have an intermediate tool which would cover the gap between the manual pentest and the automated vulnerability scan appeared. The approach we decided to follow was to leverage of different opensource tools and integrate them in a single solution to develop our own intermediate tool.

The list of opensource tools that has used to create the semi-automatic pentest suite can be seen below. We assume we will need adding other interesting opensource tools to the suite in order to continuously improve it.

1. Wfuzz – The Web Fuzzer [7]

   Wfuzz has been created to facilitate the task in web applications assessments and it is based on a simple concept: it replaces any reference to the FUZZ keyword by the value of a given payload. In the custom tool Wfuzz is used to double check that the findings of other vulnerabilities are correct.

2. Arjun – HTTP Parameter Discovery Suite [8]

   Arjun is a HTTP discovery tool which parses the response of an initial webpage and crawls it up to the depth that you define while extracting all the parameters that it finds.

3. XSStrike – Advanced XSS Detection Suite [9]

   XSStrike uses Arjun as a parameter finder and then injects different payloads (which you can define) in the parameters that it has found.

4. SQLMap - Automatic SQLInjection and database takeover [10]

   SQLMap is an opensource penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers.

Besides using the previous tools, it was also agreed to use the GoLang language when coding the suite that would glue together the different opensource tools. The reason for that particular choice is that it is way more efficient when doing lots of requests per second as can be seen in the following graph (Fig. 13):

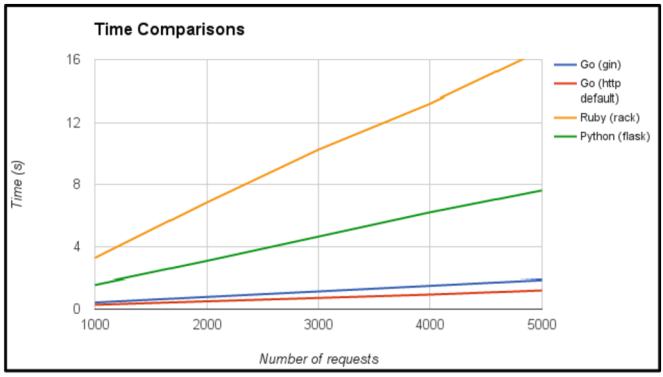


**Fig. 13.** Time Comparisons of requests between different languages.

As a high-level description, the tool starts using Arjun to crawl and extract parameters from a website, which are later on saved on a txt file. This file is available for the pentester to use in case he needs to test something in all the endpoints using a custom script and needs a dictionary of valid endpoints and parameters.

After the page has been crawled and the parameters have been extracted the next step is to launch first XSStrike to find parameters vulnerable to XSS injections and then to launch SQLMap in the same endpoints and parameters in order to try to find endpoints and parameters vulnerable to SQL injections.

Finally, based on the results from XSStrike and SQLMap, Wfuzz is used to double check that the findings are not false positives, if there is still some doubt the payload is left for the pentester to review. In Fig. 14 you can see a picture of how a scan to a vulnerable endpoint works and how it does find the vulnerable q parameter in the Vue Storefront application.

```
root@KaliLinux:/home/oalshaeb/pentestAutomation/tools# ./paWeb -u http://10.0.2.8:3000 -f pwa.txt

----------FUZZING-----------

http://10.0.2.8:3000/test
http://10.0.2.8:3000/test1
http://10.0.2.8:3000/test.php
http://10.0.2.8:3000/test1.php

----------FILES FOUND-----------

http://10.0.2.8:3000/test.php
http://10.0.2.8:3000/test1.php

----------ENDPOINTS FOUND-----------

http://10.0.2.8:3000/test
http://10.0.2.8:3000/test1
http://10.0.2.8:3000

----------STOPPED ENDPOINTS (check manually)-----------

http://10.0.2.8:3000/test/STOPPED
http://10.0.2.8:3000/test1/STOPPED
http://10.0.2.8:3000/STOPPED

----------PARAMETERS FOUND-----------

http://10.0.2.8:3000/?q=

----------SEARCHING FOR XSS VULNERABILITIES-----------


----------VULNERABLE to XSS ENDPOINTS FOUND-----------

http://10.0.2.8:3000/?q=

----------SEARCHING FOR SQLi VULNERABILITIES-----------


----------VULNERABLE to SQLi ENDPOINTS FOUND-----------


root@KaliLinux:/home/oalshaeb/pentestAutomation/tools#
```

**Fig. 14.** Example of execution of the developed tool.

## 8 Conclusions

Penetration testing is rapidly changing. Not only there are new languages and technologies that force pentesters to continuously learn and improve, but also new platforms which need auditing, like IoTs and APIs, which until some years ago were not so widely used as today.

In this paper a problem that was found during a pentest has been depicted, dissected, and analysed in order to find the causes of its behaviour and how to solve it in the future.

After the analysis, it is fair to say that Qualys and other similar web scanners have difficulties when scanning applications that do not use the classic approach of Click -> Request -> Response, but rather when a single request can retrieve enough information for the webpage to continue hours or days without the need of additional requests.

In order to solve this situation, the approach which had more sense and that has been carried out in this research was to build a tool that would allow a pentester to define exactly what he wants to test, keeping a copy of the responses received during the testing in order to clearly differentiate between a false positive and a real vulnerability and also

keeping a copy of all the endpoints and parameters tested in case he or she wants to repeat any test manually.

This approach has been proved to be useful when finding this type of vulnerabilities, as the tool that has been developed does detect the q parameter as a vulnerable one. As future work, we propose to enhance and expand the tool and build a process around it to use it in every vulnerability scan or penetration test done.

## References

1. Qualys website. https://www.qualys.com/
2. Vulnerability Assessment Gartner Ranking. https://www.gartner.com/reviews/market/vulnerability-assessment
3. Harnessing Modern Web Architecture with Progressive Web Apps. https://mentormate.com/blog/modern-web-application-architecture/
4. Vue Storefront Headless Ecommerce. https://www.vuestorefront.io/
5. PortSwigger Active Scan++ website. https://portswigger.net/bappstore/3123d5b5f25c4128894d97ea1acc4976
6. Zed Attack Proxy (ZAP) website. https://www.zaproxy.org/
7. Wfuzz: The Web fuzzer Documentation. https://wfuzz.readthedocs.io/en/latest/
8. Arjunt: HTTP Parameter Discovery Suite Github Project. https://github.com/s0md3v/Arjun
9. XSStrike: Advanced XSS Detection Suite Github Project. https://github.com/s0md3v/XSStrike
10. SQLmap website. https://sqlmap.org/