UNIVERSITY OF TARTU
Institute of Computer Science
Cyber Security Curriculum

Wael AbuSeada

Alternative Approach to Automate Detection of DOM-XSS Vulnerabilities

Master's Thesis (30 ECTS)

Supervisors:
Olaf Manuel Maennel
Raimundas Matulevičius

Tartu 2017

Alternative Approach to Automate Detection of DOM-XSS Vulnerabilities

Abstract:

This thesis proposes an alternative methodology to detect DOM-XSS by building-up on the existing approach used by web scanners in detecting general XSS. Web scanners general approach is to inject payload in the web page inputs and check the received HTML response for possible cross-site scripting vulnerabilities. The thesis proposes to add an extra scan layer which is an actual browser that would be responsible for sending any request and render the received HTML response from webserver. Rendering the response causes any script in the page to be executed, hence any code that alters the page dynamic content causing DOM-XSS will reflect on the rendered response. Then the rendered response is checked for XSS vulnerabilities. The thesis methodology allows detecting both DOM-XSS and other types of XSS. To provide a proof of concept for this methodology, the thesis author created a web-based tool on that premises. The tool can open and control a browser which allows automated loading of web pages and scanning the rendered response for vulnerabilities. Finally, the tool provides detailed scan report that points out possible inputs that might cause XSS in order to assist penetration testers who prefer manual scans.

Keywords:

Cross-site scripting, DOM-XSS, web security, web scanner

CERCS: P170, Computer science, numerical analysis, systems, control

Pealkiri eesti keeles: Alternatiivne lähenemine automatiseeritud DOM-XSS haavatavuste tuvastamisele

Lühikokkuvõte:

Käesolevas lõputöös pakutakse välja alternatiivne meetod DOM-XSS tuvastamiseks, toetudes juba olemasolevatele lähenemistele, mida kasutavad erinevad XSS tuvastamise veebiskännerid. Veebiskännerite üldine lähenemine on selline, et kõikidesse skännitavatesse veebisisenditesse sisestatakse kood ning kontrollitakse HTML vastust, et tuvastada potentsiaalne XSS haavatavus. Antud lõputöös tehakse ettepanek tuua sisse lisaskännimise kiht, mis kujutab endast eraldi veebilehitsejat. See veebilehitseja vastutaks veebiserverisse kõikide päringute saatmise eest ja HTML vastuste kuvamise eest. Vastuse kuvamine käivitaks kõik lehel olevad programmikoodid. Iga kood, mis muudab veebi sisu dünaamiliselt põhjustades DOM-XSS, kajastuks renderdatud vastuses. Kuvatud vastuses kontrollitakse XSS haavatavuse olemasolu. Käesoleva lõputöö meetod võimaldab tuvastada nii DOM-XSS kui ka teisi XSS liike. Selleks, et seda meetodit tõestada, on lõputöö autor loonud veebipõhise tööriista XSS tuvastamiseks. Antud tööriist suudab avada ja kontrollida veebilehitsejat, mis võimaldab automaatselt kuvatud veebilehe haavatavust kontrollida. Lõpuks annab tööriist väljundiks skännimisest tekkinud raporti, mis näitab potentsiaalseid XSS vastu haavatavaid sisendeid. See tööriist aitab penetratsiooni testijaid, kes eelistavad manuaalset testimist.

Võtmesõnad:

Cross-site scripting, DOM-XSS, veebi turvalisus, veebiskänner

CERCS: P170, arvutiteadus, numbriline analüüs, süsteemid, kontroll

Contents

1 Introduction

Cross-site scripting is one of the most common vulnerabilities in web applications. Latest Web security statistics reports show that cross-site scripting vulnerability has high likelihood to be discovered in a web application. The vulnerability is of an injection type, which means it allows hacker to inject malicious code into the website. The vulnerability is caused due to lack of proper encoding on user input data by the website. This allows hackers to inject malicious code that will be stored either on the website or in the URL that the victim is tricked to click. The vulnerability does not harm the website server itself, but will harm the website visitors.

Although cross-site scripting was discovered long time ago, it still could be found in many websites including the ones that belong to big companies like Google, Facebook and others. These big companies usually have separate security team that are responsible of running manual and automated scans to discover any vulnerabilities before launching any website or pushing new updates to existing one. Despite that fact, security researchers are frequently reporting cross-site scripting vulnerabilities through their bug bounty programs. This means that the tools used to automate scan for cross-site scripting are still not capable of fully detecting different types of cross-site scripting or following specific paths in websites on its own.

This thesis concentrate on the part of enhancing the methodology of scanner to detect DOM-XSS. Most general web scanner detect cross-site scripting in general by analysing the received HTML response of webpage and looking for XSS vulnerabilities based on the injected payload. This approach is not capable of detecting DOM-XSS because the payload in that case would only reflect in the page after it has been rendered in the browser; hence scripts in the page are executed. Security researchers have proposed different methodology that could allow them to detect DOM-XSS using taint tracking or dynamic taint tracking. Both techniques propose to monitor if script in the page access page resources that include user input string and follow their execution until it reaches single or various sinks. There has been either standalone tool or integrated tool developed to proof their approach.

This thesis proposes different approach to detect DOM-XSS that could be directly integrated into the general approach used by the web scanners which is to add an extra layer between the steps of receiving the HTML response from the web server and checking that response for possible cross-site scripting vulnerabilities. The layer is an actual browser which renders the HTML response causing any script in the page to get executed. This leaves the burden of analysing big or complicated JS code to the browser to render it. Then analysing that rendered response the same way used by web scanners would allow to find any DOM-XSS. Web scanners can manage to detect all types of cross-site scripting by analysing both responses. Since the methodology proposes communicating with an actual browser to render the HTML response, the browser could also be used to validate if injected payload gets executed which eliminates any false positive results. The tool targets reflected XSS vulnerability which could be caused by URL parameters as a proof of concept of the methodology.

The thesis author developed a web-based tool as a proof of concept of the proposed methodology. The tool uses Selenium to open and control browser as well as inject data into webpages. It follows the same process used by penetration testers in detecting inputs in the webpage then injecting them with unique payload to identify which of these inputs data reflect in both received and rendered response of the webpage. Then it choses list of possible payloads that can bypass user input filters or sanitization and get executed by the browser. The injected payload is tested in the browser to check if it will pop-up an alert box to eliminate any false positive results.

2 Background

In the first section, the thesis author explains cross-site scripting (XSS) and its different types. He also explains the difference between HTML in the response received from the server and the one rendered by the browser. He also explains how scanner automates the process of detecting different types of cross-site scripting vulnerabilities. Finally, he explains briefly the previous related work on automating detection of DOM-XSS.

2.1 What is XSS?

Cross-site scripting is an injection vulnerability that can be found in web applications. It allows hackers to inject malicious code that will be executed on the victim's browser. This makes the vulnerability of an attack user type which means the vulnerability could be used to harm users of that vulnerable web application.

Based on the vulnerability type, attacker can either store malicious code into the website or craft a link that has that malicious code and trick his victim to visit that webpage, which will then make the victim's browser executes that code. The crafted code could allow a hacker to launch multiple different types of attack such as session hijacking, key logging, phishing and much more attacks [1].

Hackers can hijack a session through XSS vulnerability by injecting a malicious code that steals the session login cookie. The malicious injected code could also allow hackers to add keyboard listeners allowing them to log keyboard strokes by the victim. It can also manipulate the DOM allowing the hacker to create fake forms, change the website layout or even redirect the victim to other websites.

2.2 Types of XSS [2]

Stored (persistent)

Stored cross-site scripting vulnerability occurs when injected malicious code is saved into the website's database without proper encoding. Then the next time any user request a page where this data should be displayed and the server does not do proper encoding before adding it to the page dynamic content. For example: if a user adding a comment in web blog, it will be saved into database. Next time someone visits the blog the comment will be queried from the database and loaded into the page. If server does not do proper encoding in both scenarios to filter malicious code, it will store the malicious payload in its database. See Figure (1).

Reflected

Reflected cross-site scripting vulnerability occurs when malicious code is part of the request sent to the web server. The server sends back as part of the response. If the malicious code did not get the proper encoding, it will get executed by the browser causing reflected XSS. For example you can find the vulnerability in search bars that exists in websites that allows users to search for keywords. The vulnerability happens when the user input is not properly checked against malicious code and characters. See Figure (2).

DOM-XSS

This kind of vulnerability occurs when the script code found in the page inject the reflected/stored malicious code into the page dynamic content. This means that the malicious code will only appear after the victim's browser renders the page, hence executing scripts.

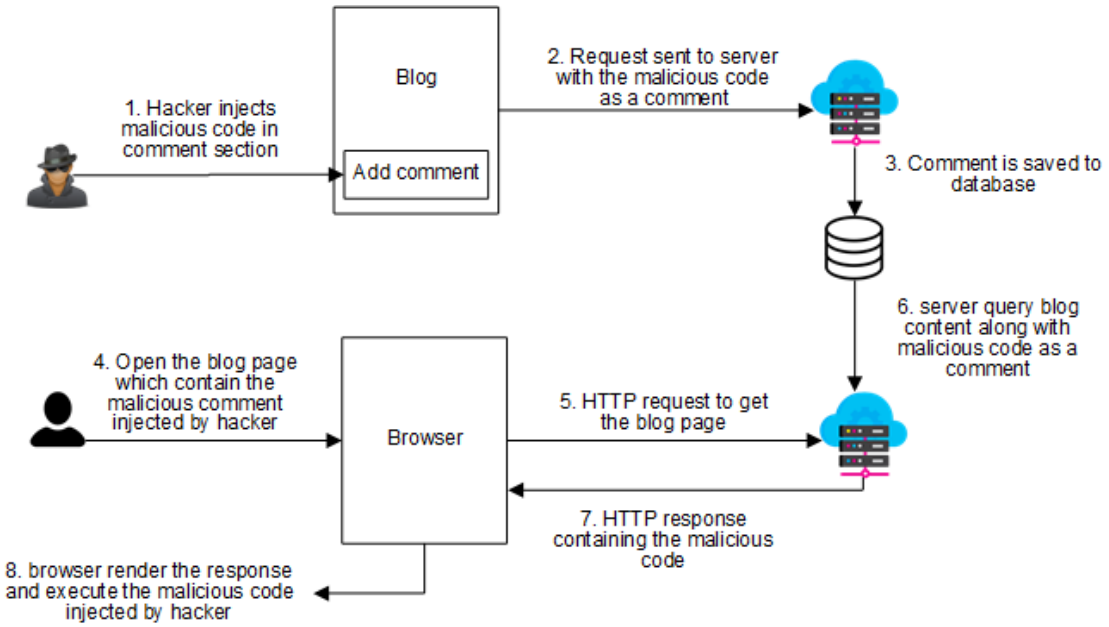

Figure 1. Stored XSS scenario.

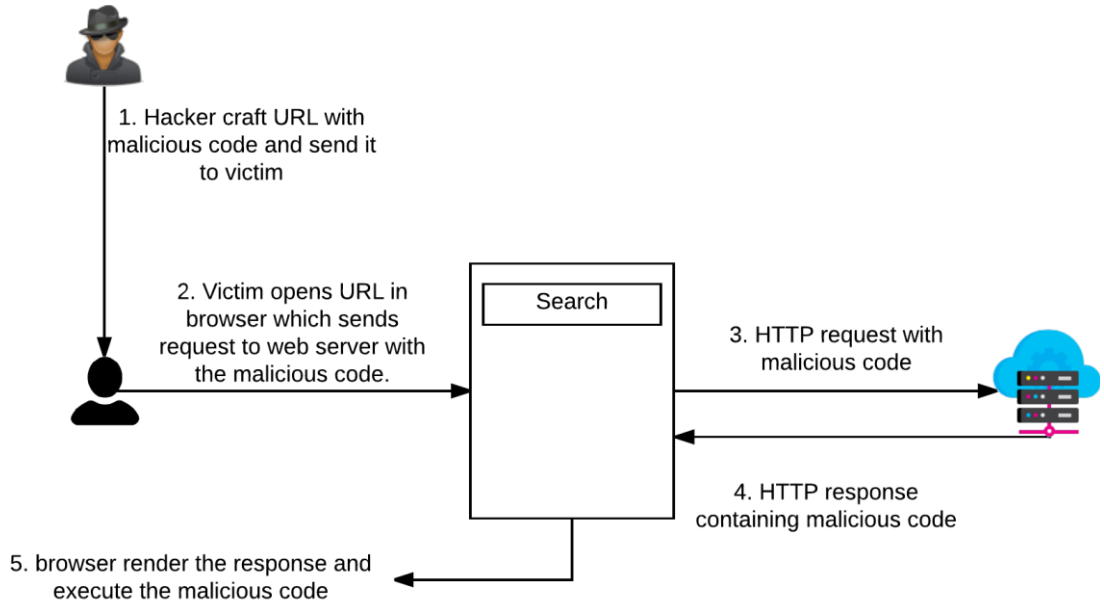

Figure 2. Reflected XSS scenario.

On 2005 Amit Klein identified another kind of XSS vulnerability called DOM based cross-site scripting or XSS of the third kind [3]. The vulnerability type was first identified as the malicious code that does not need to be sent back to server. For example, data after fragment identifiers [4] (commonly known as URL hash, #) in URLs do not get sent to the server and

are processed at client-side. This means server will not receive any malicious code and hence user data encoding at the server side will not be enough to defend against this type of attack.

But later with the massive use of javascript in websites which allowed manipulation of the page dynamic content, the DOM-XSS vulnerability was found to be a mix of reflected and stored XSS. This led researchers to re-identify cross-site scripting types to two general categories: server-XSS and client-XSS which both have two sub-categories stored and reflected.

Reflected server XSS occurs when the malicious code is injected in the URL and that code reflects in the received HTTP response while reflected client XSS also has the malicious code in the URL but it will only reflect in the page content after it is rendered by the browser; hence scripts in the page are executed. Stored server XSS is when the malicious payload is being stored in the website's database and will appear directly in the received HTTP response. Stored client XSS also has the code stored in website's database but will only appear when the webpage is rendered. See Figure (3).

Figure (4) shows an example of reflected client-XSS scenario. After the hacker has discovered that the website has a DOM-XSS vulnerability, he crafted a link that contains malicious

XSS	Server	Client
Stored	Stored Server XSS	Stored Client XSS
Reflected	Reflected Server XSS	Reflected Client XSS

Figure 3. Types of XSS [5].

code and managed to get his victim to open it. When the victim's browser loads the URL, it will send a request that contains the malicious code to website's server. The HTML found in the server response will not contain the malicious code in it, which will bypass any XSS auditors. But when the browser renders that received HTML, the script in the page will get executed which will add the malicious code into the web page dynamic content allowing it to get executed. The example scenario could be found at:

http://alert1.me/vuln/profile.html?accName=qwe&accID=%22%3E%3Cscrpt%3Ealert(1)%3C/script%3E

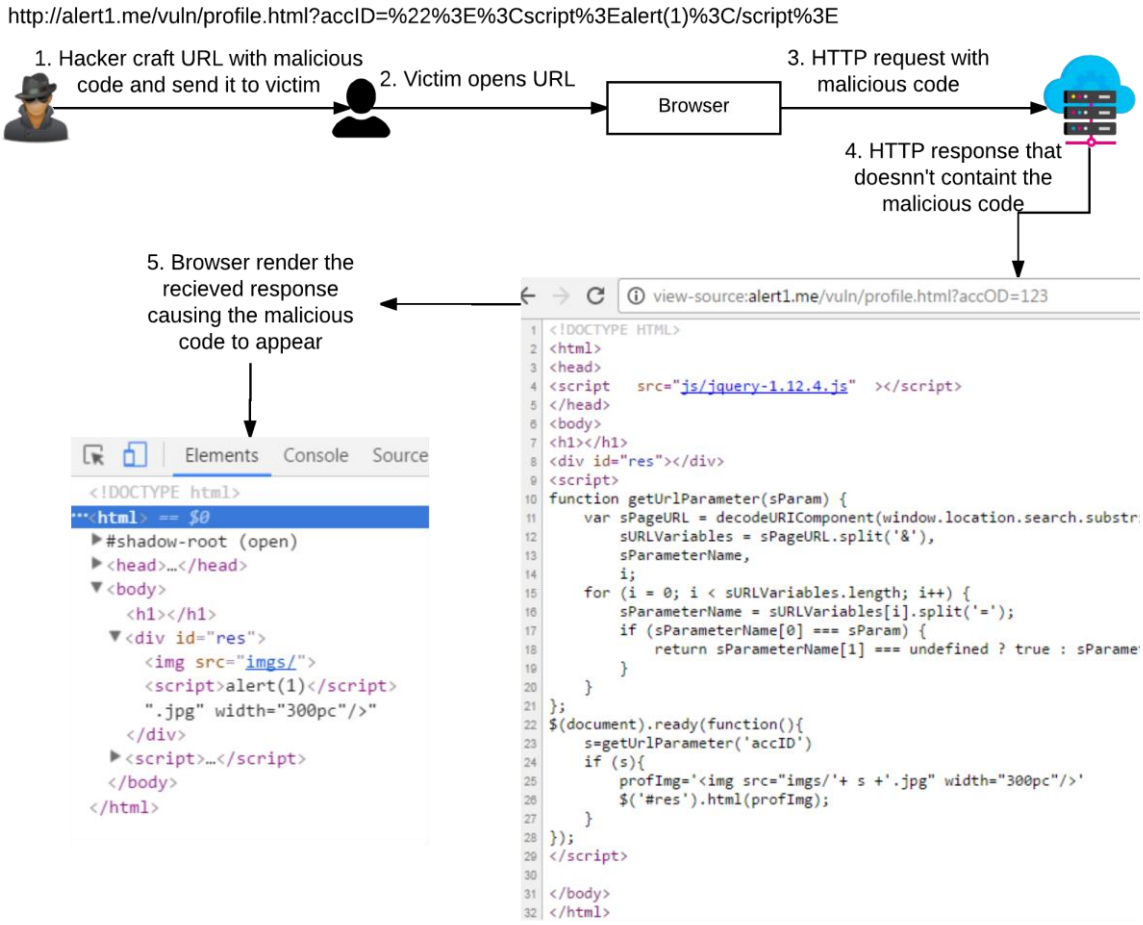

Figure 4. DOM-XSS scenario.

2.3 Why is it important to tackle XSS?

The web application hacker's handbook [6] describes XSS attacks as follows:

> "The Godfather of attacks against other users. It is by some measure the most prevalent web application vulnerability found in the wild."

Web security statistics reports like Edgescan [7] or Whitehat security [8] show that cross site scripting has a high likelihood of being discovered in web applications as shown in Figures (5) and (6) respectively. This is partially due to the fact that some web developers do not have the appropriate training for secure coding. Although it is not always necessarily the reason as there are a lot of discovered XSS bugs in websites owned by big companies which have dedicated security teams that run manual and automatic penetration testing but still missed detecting these vulnerabilities. That proves that XSS vulnerabilities can appear in the most unexpected places in a website, especially in complicated websites that gets updated frequently in order to add more features which might use unsafe user data. That is why a lot of companies like: Microsoft, Facebook, Google and so forth offer bug bounty programs to encourage security researchers to report found bugs by offering rewards for their bug reporting [9].

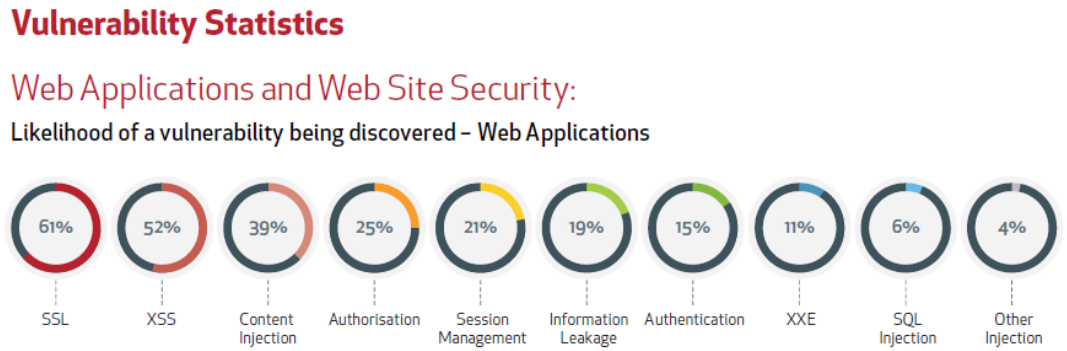

Figure 5. Web vulnerability likelihood in Edgescan statistics report.

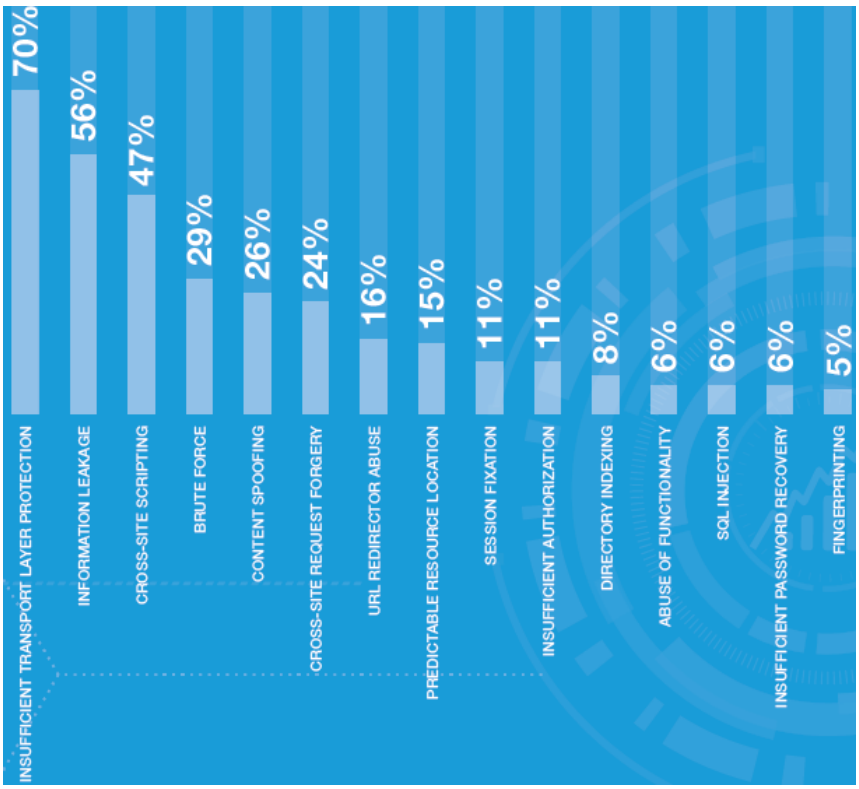

Figure 6. Web vulnerability likelihood in WhiteHat statistics report.

2.4 Server response versus rendered response

In this section, thesis author will explain the difference between HTML code found in the received response and the rendered response after browser loads the web page.

When user tries to open a web page in browser, it sends HTTP request to website's server. The server handle the request and then replies with a response that contains HTML of that web page. Then browser parses and corrects that HTML response and builds the document object model (DOM)[10] which is a map of where objects displayed in the page, cascaded style sheet object model (CSSOM) and render tree [11]. The render process adds objects found in the HTML to the webpage and also executes any scripts found in the server response. Such scripts can alter and add new objects to DOM which makes the server response different than the rendered one by the browser.

For an example of the difference between HTML response and the rendered response, the thesis author created a simple web page that has a script which adds a paragraph div into the page body. Figure (7) shows the HTML of the received response which does not have any data in the page body. But when the browser loads the web page and executes the script, it will inject a paragraph inside the page body as shown in Figure (8).

```
1
2  <!DOCTYPE HTML>
3  <html>
4    <head>
5    </head>
6  <body>
7    <script>
8      document.body.innerHTML='<p>Hello</p>';
9    </script>
10 </body>
11 </html>
12
```

Figure 7. Example of HTML from server response.

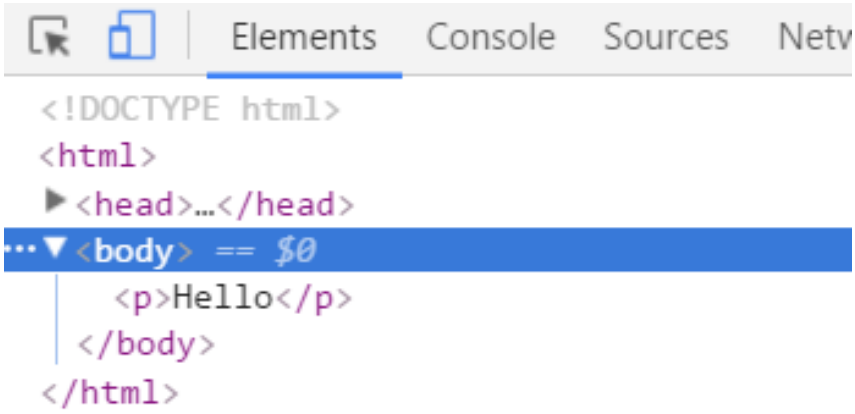

Figure 8. Example of rendered response.

2.5 Scanners approach to detect general XSS

This section explains the approach taken by scanners to detect cross-site scripting vulnerabilities in general. General web vulnerability scanners use a black-box testing approach in which website implementation is not known to scanner. The scanner starts by sending HTTP request to the website's server and then analyses the received response for possible vulnerable inputs. Then it injects payload from a predefined payload list into the possible vulnerable input and send request with that malicious payload. Once again, the response is analysed to confirm that injected payload could cause XSS vulnerability [12]. See Figure (9).

This approach is not capable of detecting DOM-XSS on its own as the vulnerable input's data reflects only after scripts in the page get executed.

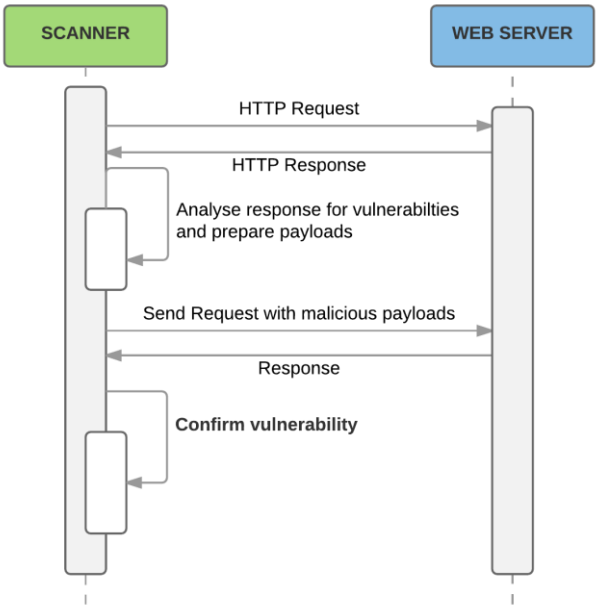

Figure 9. Sequence diagram of general scanners approach to detect XSS.

2.6 Related Work

There are different set of academic tools (DexterJS, DOMinator, Flax, Xenotix and the tool developed by Sebastian Lekies, Ben Stock and Martin Johns) that address the issue of automating detection of DOM-XSS vulnerabilities. Their approaches vary between creating a standalone tool or browser-integrated one that does either taint tracking or dynamic taint-tracking in order to identify vulnerable JS code that causes DOM-XSS. The following is a brief overview of these tools.

DexterJS

DexterJS uses taint tracking to detect DOM-XSS. It acts as a trusted MiM proxy to intercept any request from browser and identify scripts in the page. Then it rewrites these scripts to perform character-precise taint tracking and pass the result to exploit generator which create an attack vector that could cause DOM-XSS [13]. See Figure (10).

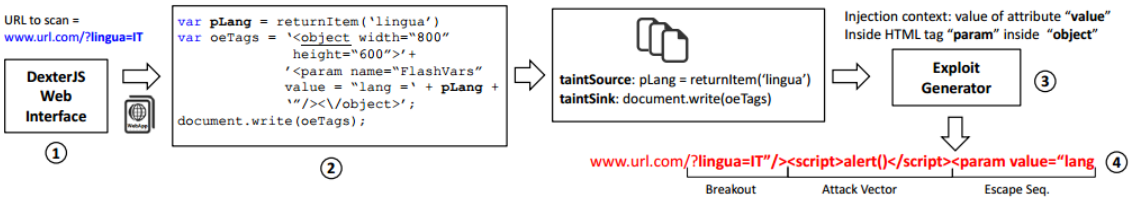

Figure 10. DexterJS approach to detect DOM-XSS [13].

DOMinator

DOMinator uses dynamic taint-tracking to detect DOM-XSS by instrumenting Firefox's SpiderMonkey javascript engine. It has a function that tracks the execution history of tainted input until it reaches its sink and stores this tracking data so that exploit module can alter these data and follow the same path to validate if it was vulnerable [14].

Flax

Flax uses more dynamic approach called taint enhanced black-box fuzzing which combines dynamic taint analysis with automated random fuzzing. The dynamic taint analysis part of the tool identifies all uses of untrusted data in critical sink then automated random fuzzing is responsible to find an input that can cause DOM-XSS [15]. See Figure (11).

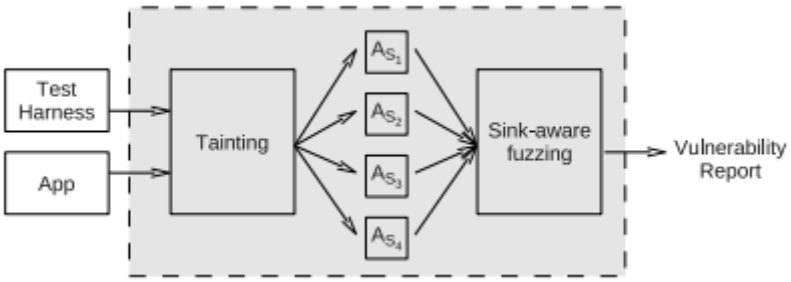

Figure 11. Flax approach to detect DOM-XSS [15].

Xenotix

OWASP Xenotix XSS Exploit Framework is both a scanner and exploit tool. The tool also has a module to specifically detect DOM-XSS that uses taint tracking to analyse scripts in the page and identify used sources and sinks that might cause DOM-XSS [16].

The last academic tool is developed by Sebastian Lekies, Ben Stock and Martin Johns. The tool is browser-integrated one that consists of two separate components which are modified browsing engine and automated vulnerability validation mechanism. The modified browsing engine allows the tool to do dynamic byte-level taint-tracking of suspicious flows. The automated vulnerability validation mechanism processes the information provided by the browsing engine and create attack payload that could exploit existing DOM-XSS vulnerability [17].

On the other hand, Acunetix which is one of the most popular commercial web scanners defines its approach to detect DOM-XSS by using "DeepScan technology" [18] which allows it to trace script execution cycle and monitor page source that the script uses until it reaches single or various sinks. That approach is more like dynamic taint-tracking that was used by previous academic tool.

3 Methodology

In this section, the author will explain this thesis alternative approach to detect DOM-XSS and the benefit of using it over other academic or commercial approaches mentioned in the related work section.

There are couple of challenges that face the automation process to fully detect cross-site scripting vulnerabilities. As mentioned in the background section, web scanners detect cross-site scripting vulnerabilities by detecting all inputs in the webpage then identify which of these inputs data is reflected in the received response from web server by injecting unique malicious payload in inputs and analyse the response for cross-site scripting vulnerabilities. This approach can effectively detect server reflected cross-site scripting vulnerabilities. But it will fail to detect some cases of stored cross-site scripting vulnerabilities as well as DOM-XSS in general. For an example of missed cases of stored XSS: when the form submission doesn't direct to the page where it contains the injected payload but it directs to an intermediate page that contains status of form submission or page with another form, the scanner will fail to detect the vulnerability. It will also fail to detect any DOM-XSS vulnerabilities due to the fact that injected data appears only in the rendered response. These issues are one of the most challenges facing automation of detecting XSS. The thesis methodology concentrate in automating detection of DOM-XSS.

As mentioned in related work section, the previous work on detecting DOM-XSS concentrates on analysing the scripts behaviour in the page to detect if any unsafe user input was injected into page dynamic content by either using taint-tracking or dynamic taint-tracking. This is done by (static or dynamic) monitoring of variables and functions that access page resources which contains user strings and trace their execution until they reach a single or multiple sinks. This approach leads to have a separate tool or module that specifically made for DOM-XSS analysis which was only adopted commercially by Acunetix. The approach also might fail sometimes to detect vulnerabilities with big complicated javascript libraries and even use different ways of accessing page resources and output these resources to page dynamic content.

This thesis proposes an alternative approach in detecting cross-site scripting vulnerabilities which is to add a layer in the scanner that can render all received responses before analysing them for injected payload. To add such a layer the scanner needs to have either a built-in browser or to open and control a browser in order to be able to render the received HTML response. Adding such layer will execute any scripts in the page and also could be used to confirm if the injection successfully exploited XSS vulnerability or not, hence decreasing false positive to zero. The approach also facilitate integrating the tool with any general web scanner giving it the ability to detect DOM-XSS and validate any vulnerability through an actual browser.

To provide a proof of concept that this methodology is both applicable and can successfully detect DOM-XSS vulnerabilities, the thesis author developed a tool that is able to open, control a web browser and fill inputs found in the loaded webpage. To achieve that there are two ways. The first one is to create a built-in browser from scratch using web views or something similar which would allow the developed tool to fully control it, but it would end-up being different than modern web browsers that users actually use, as each browser has a certain way in parsing and rendering received HTML. Even some browsers like Chrome have special built-in plugins to detect XSS called XSS-auditor [19]. Also Firefox automatically encodes certain meta-character in URL by default.

That leads to the second option where the designed tool would use a third party tool to communicate with an actual browser. There are unit testing libraries and tools that were developed to allow web developers to write test cases to test their website's functionality. One of these tools, Selenium [20], gives the full power of opening an actual web browser and gives total control of opening web pages, filling input fields in the page and even checking if any pop-up alerts were triggered by the webpage. Web developers use Selenium to write test scripts that will open an actual web browser using its correspondent web driver and follow a test case scenario to check their website functionality. The author's developed tool will use Selenium to open and control an actual web browsers which allows the tool to have the option of running scan in multiple different browsers.

The tool will use Selenium to open a browser which returns a web driver object that allows the tool to control the browser. Then the tool loads the URL that needs scanning into the browser. The browser communicates with the webserver by sending an HTTP request and wait for the HTML in the response to render it. Once rendering is done, it send the rendered code to the tool. The tool detect all possible input within the rendered response including inputs inside forms and outside forms as well as URL parameters. Then it injects unique

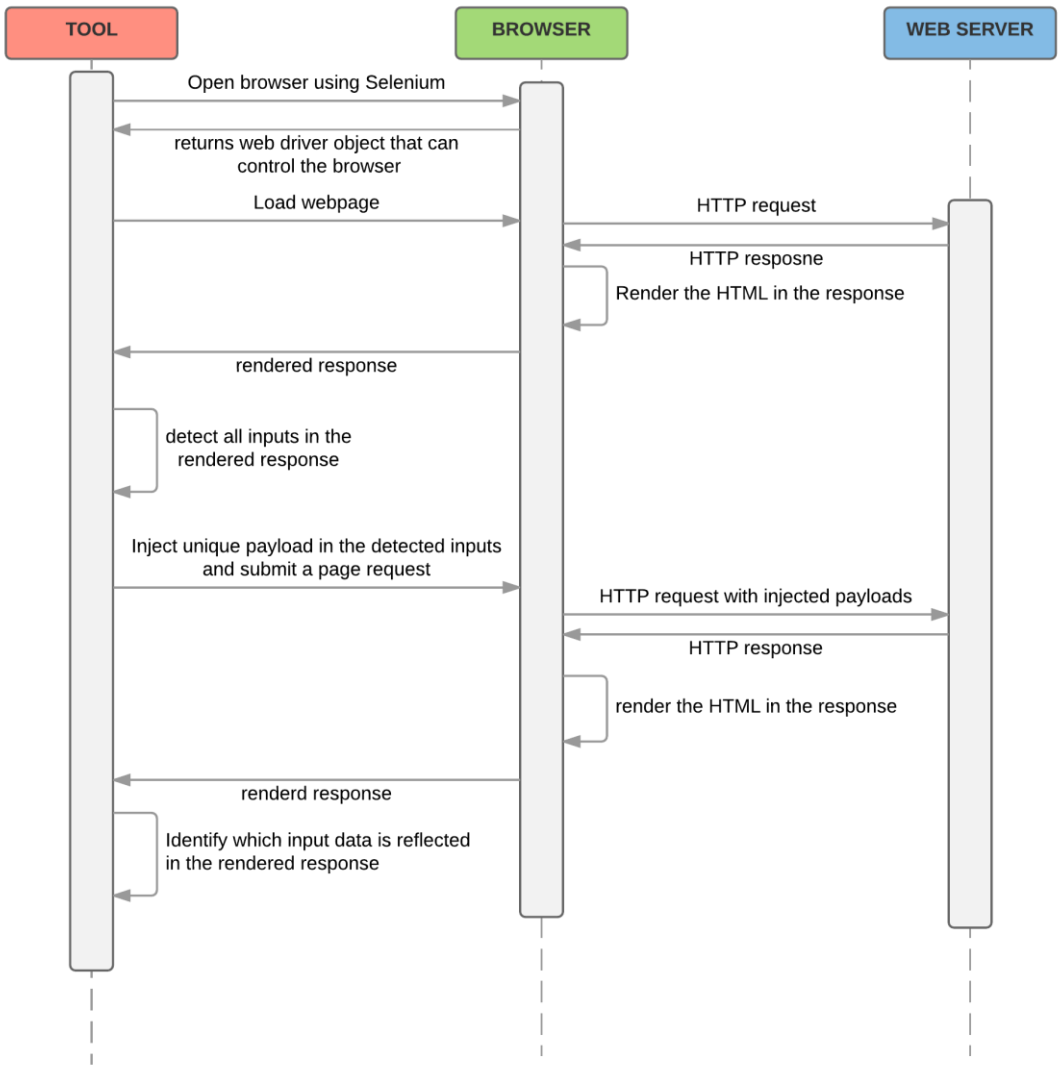

Figure 12. Sequence of detecting possible vulnerable inputs in a webpage by thesis tool.

payload in each input in the form and submit a form request. The browser will take care of submitting the request which will send HTTP request that contains the injected payloads and then received the response. The browser once again render the response and send it to the tool which checks which of the inputs data were reflected in it. See Figure (12).

At this point the tool has list of all possible vulnerable inputs in the webpage. The next step is to check if the website encode/filter injected data in these inputs which could be done by re-injecting payload along with specific characters/words that are needed to inject malicious code that would cause cross-site scripting vulnerability. The browser takes care of sending a request to web server and send back the rendered response to the tool. The tool crafts a list of possible payloads that could be injected in each input based on the position of the data reflected in the response and the filters identified for that input. Then it injects each payload for the input separately and the browser takes care once more of submitting forms with the

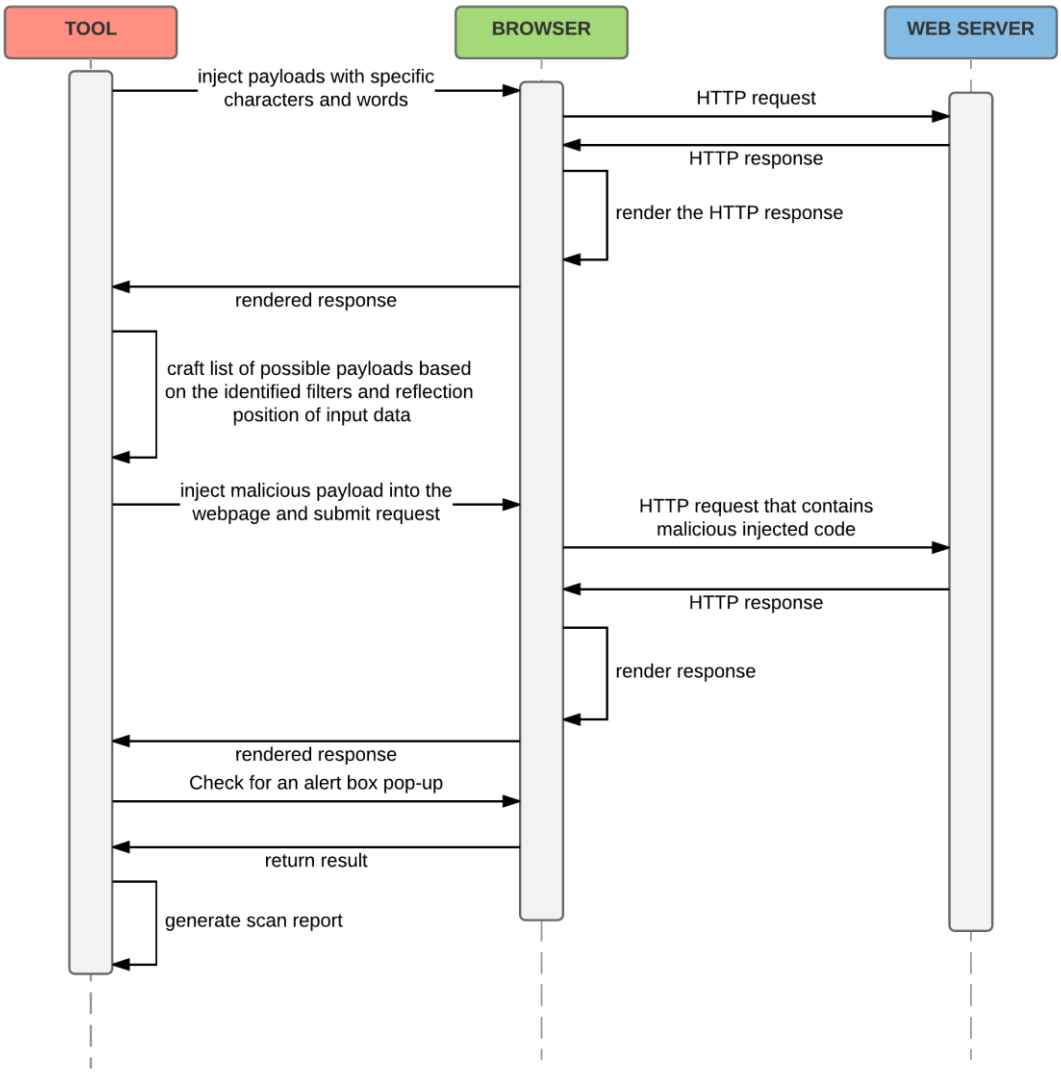

Figure 13. Sequence diagram showing how thesis tool injects malicious payload and confirms its execution.

malicious payload and send back the rendered response to the tool. The tool checks if each payload manage to pop-up an alert box or not. Finally the tool generates a report that contains scan results including possible vulnerable inputs, vulnerability found, payload used and scan metrics like number of request sent and scan duration. See Figure (13).

The drawback of using Selenium is that it still has some compatibility issues [21] which makes it so that Selenium only runs specific versions of web browsers that are both compatible with the its version and the version of the web driver that it user to control the browser. For the tool to run a scan it will open a browser and keep loading the scan target webpage while populating its input with different payloads then tries to check if the injected payload gets executed or not. Due to these issues and the noise that would be generated by the tool while running scan, the thesis author decided to make the tool a web-based one. Where users can just submit scan requests with the desired webpages and do not have to worry about any compatibility issues, complicated installation steps or system noise generated by the scan.

Having the developed scanner as a web-based tool will add some overhead issues that need special handling like adding authentication in order to attribute each scan to an actual logged in user, since the attack will be generated from the website server. Also adding authentication to a website will require using SSL communication to secure user credentials. Implementing database in order to save user account details as well as scan results done by each user. Since the attacks is generated from the tool web server, it is not enough to attribute scans to specific account so the author uses domain control validation (DCV) to confirm the user ownership of the domain. Domain control validation is done by asking the user to upload an empty file with unique name to the domain root, hence confirm ownership of the domain. Finally since the scan will probably takes long time to finish (more than 2 seconds at least), scan requests should be pushed to a queue and handled one at a time.

4 Implementation:

The previous section explained the methodology of this thesis and why the proof of concept tool that detects DOM-XSS will be web-based one. This section will explain implementation of the tool from setting up the server, identifying all the used tools, plugins or libraries and implementing the XSS attack process.

4.1 Server setup

The tool is hosted in DigitalOcean [22] VPS under the domain (http//:alert1.me). Due to budget restraints the web server has the minimum technical specifications that are sufficient enough to run the tool which are:

- 4GB memory/2CPU.
- 20GB disk.
- OS: Ubuntu 16.04.1 x64.

There are set of tools that needs to be installed on the VPS in order to be able to implement the tool. We will mention these applications and their use, but will not go in details of the installation and configuration steps since it would be out of thesis scope. The needed tools are as shown in Table (1).

Table 1: Required tools on the VPS for the thesis tool.

Tool	Usage
Apache2 v2.4.18	To be used as the tool web-server
MySQL v14.14	To be used as the database server for storing user account details and any scan data.
Postfix v3.1.0 [23]	To serve as SMTP server to allow sending account confirmation emails as well as add the possibility of sending notifications e-mail after scan is done.
Python v3.5.2	To be used as the main programming language for the tool.
Python-BeautifulSoup v4.4.1[24]	To help parsing HTML.
Django [25] v1.10.1	To be used as the web development framework.
Django-rq [26]v0.9.2	To queue scan request.
Django-Redis [27] v4.4.4	Cache server used to push scan requests to queue.
Selenium v2.53.6	To open and control web browsers using their correspondent web driver.
Chromium [28] v51.0.2704.79	To be used as an alternative browser.
Firefox v48	To be used as the main testing web browser.

PyVirtualDisplay v0.2 [29]	Python wrapper for XVFB [30] which is an X server that can run on machines with no display. It will allow us to run browsers heedlessly in Ubuntu server.
Docker 1.12.1	To act as a container for Selenium node.

4.2 Database design

In this section the author will design the database tables and their attributes to store any data processed by the website or the tool. Starting with the set of tables needed to account authentications which will be automatically generated by Django [31] and the author will not explain these tables or their attributes as they are out of scope. On the other hand, the website need tables to store scan details and generated reports.

The EER diagram of database is shown below in Figure (14). Although auth_user table belongs to the tables automatically generated by Django but it was left in the diagram since the scan table will use the primary key of the auth_user table (ID) as a foreign key. The attacks_scan table saves records of different domains that the user scanned. Records on that table are unique by design as there should be no duplicate domain that belongs to the same user. The tables have the following attributes:“id“, “domain“ to save domain URL, “token“ to save confirmation token of the domain, “time_added“ to save the time when scan was added, “spider“ if true tool will run spider web scan on the domain after domain ownership is confirmed and finally “user_id“ which is foreign key of the auth_user table.

For each domain entered by the user to scan, there might be list of pages at this domain. The tool allows manual adding of pages or doing web spider scan to detect all possible pages and then run scan on them. For every new page added by a user, the tool should check the record of the domain and create a new record of the page table that corresponds to its domain. As the attacks_scan primary key is a foreign key in the attacks_page table. The attacks_page table has the following attributes: :“id“, “page_url“, “time_added“, “time_scan_started“, “time_scan_ended“, “domain_id“ as foreign key of table scan to connect pages to domains and finally “scan_status“ which is either in queue, processing or scanned.

Finally, to save reports of the scan for each page, we created table attacks_reports. The attacks_page primary key is a foreign key in the attacks_report table which means that every record of a report corresponds to a record in attacks_page which in turn corresponds to a record in table attacks_scan that corresponds to a specific user. The attacks_report table has the following attributes: “number_request“, “scan_duration“, “vulnerability_found“ and “crafted_payload“. Also, it has attributes that relates to scan result like: “hash_access“, “url_parameters“ and “page_input“. These attributes provide information on possible injection point that can help penetration testers to fasten their testing process.

4.3 Site map

The website developed has simple structure. Starting with the landing page that gives a quick brief about cross-site scripting vulnerabilities and how scanners detect them specially DOM-XSS. Then it explains that the tool is an academic one that proposes an alternative approach in detecting DOM-XSS. The scan page is where user can add URL so that the tool

can scan them. Contact details page contains the details of the thesis author in order to contact him for any future collaboration. Finally all the web pages that are necessary for account authentication like signup, login, forgot password and reset password.

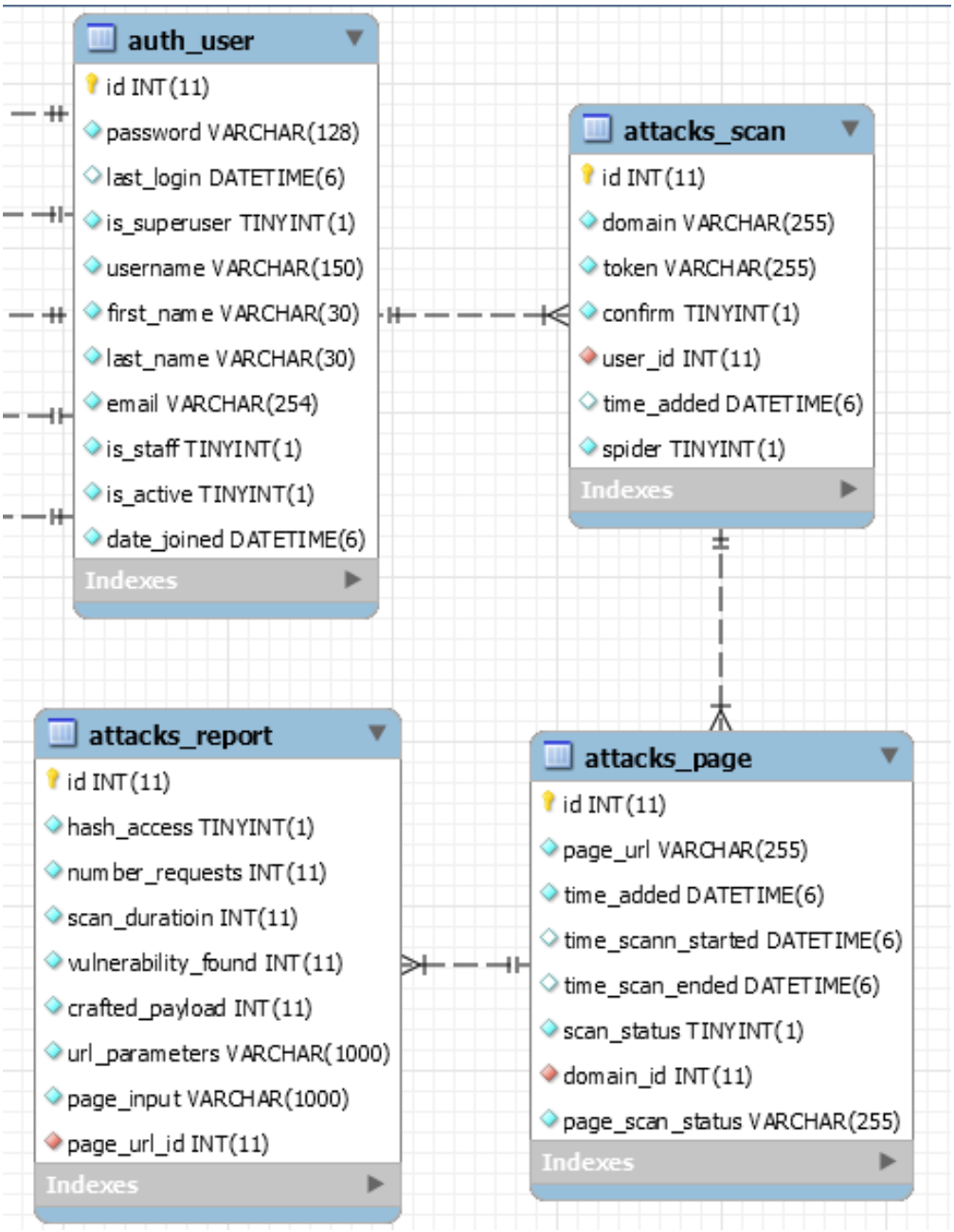

Figure 14. Database EER diagram.

4.4 Scan process

Scan process is the process that handles scan requests by the user before the actual tool run any scan. It starts with dealing with the user request to initiate new scan and explains all the possible scenarios for user interaction with the web-tool. On the other hand the process also includes how the server deal with scan request and all the possible scenarios to handle it.

We will explain first user interaction with the web-tool to initiate new scan. As shown in Figure (15), when a user opens the scan page to initiate a scan, there will be three options to choose from: first single page scan where user can add single URL for the tool to scan then domain scan where user can provide top domain URL and then server will run web spider scan to find all the possible pages that are linked to that top level domain; finally URLs from file where a user can add a file that contains list of URLs where each URL is in a separate line.

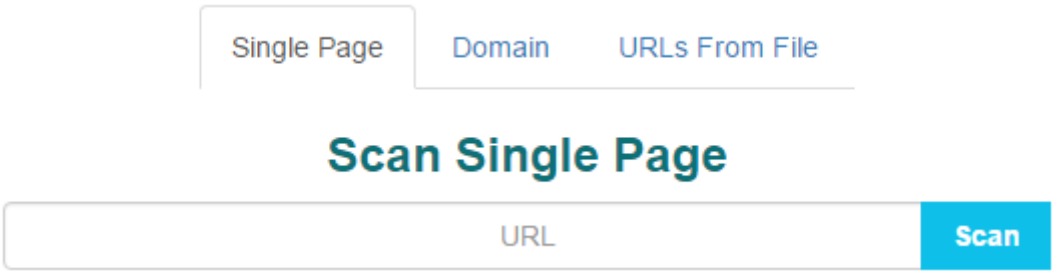

Figure 15. Scan options available to user.

When user initiates a scan process using one of the previously mentioned options it sends scan request to server which will handle the request based on the URL submitted. For any submitted URL there are three possible cases. First the domain is new one where domain in the submitted URL is new and user will be asked to confirm domain control validation (DCV) by uploading HTML page with uniquely random generated name that was provided by the tool to his domain root. See Figure (16). This allows only the domain owner to run scan on their websites since the scan requests will be sent from the tool web server.

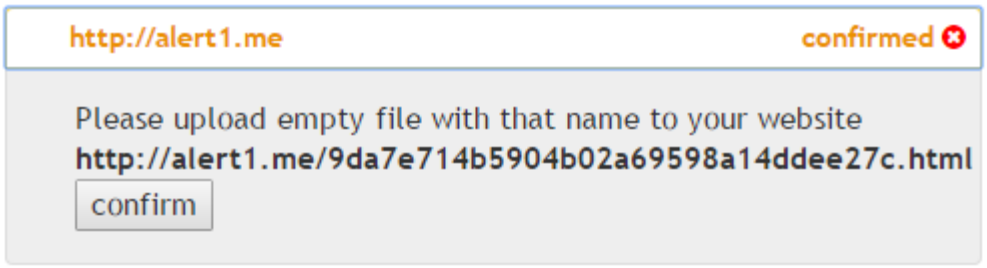

Figure 16. Result of adding URL with new domain.

The other case is that the domain exists already in the database for that logged in user and that domain ownership is confirmed. In that case, if the URL is new, a new page record will be created that belongs to that domain and then it will be added to scan queue. Otherwise, if the URL exists in the database and not in the scan queue, it will be added to scan queue. See Figure (17).

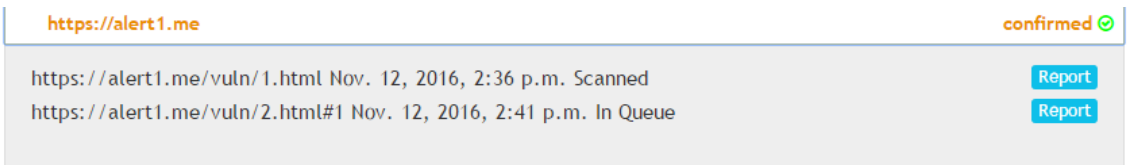

Figure 17. Result of adding URL with previously confirmed domain.

Finally if the domain also exists in the database and is not confirmed then the URL will be processed the same way as in existing confirmed domain, but it will not be added to queue. Or in case of domain scan the domain record attribute “spider” will be set to true.

For the third option found in the scan page where user can add multiple URLs from a file, each URL will be considered as separate scan entry; hence will have one of the previously mentioned cases.

For a user to confirm domain ownership, he should create an empty file with the unique generated file name in the domain root. After that on the tool scan page, he should click on the confirm button under that domain which will send a confirmation request to the backend where it will check if the file exists by sending HTTP Request and checks if the HTTP response equals 200. After domain confirmation, all pages that correspond to that domain will be added to scan queue and if the domain attributes spider is set to true, it will run web spider scan to detect all pages linked to the top level domains.

User can check each page scan status under the domain as there are three options: “In queue” when page is in queue to get scanned, “processing” when scan process for the page is ongoing or “done” where scan is done and reports were successfully generated.

On the other hand, when the server receives a scan request, first the request needs to be validated to avoid any crafted or altered request and ensure server security then it pass the request data to scan function.

As shown in Figure (18), the scan request is considered valid if it was of type POST which was just a design decision. The request should also be sent from a logged-in user which is handled by Django itself. Then it checks if the request has log-in cookie maps to an existing login session.

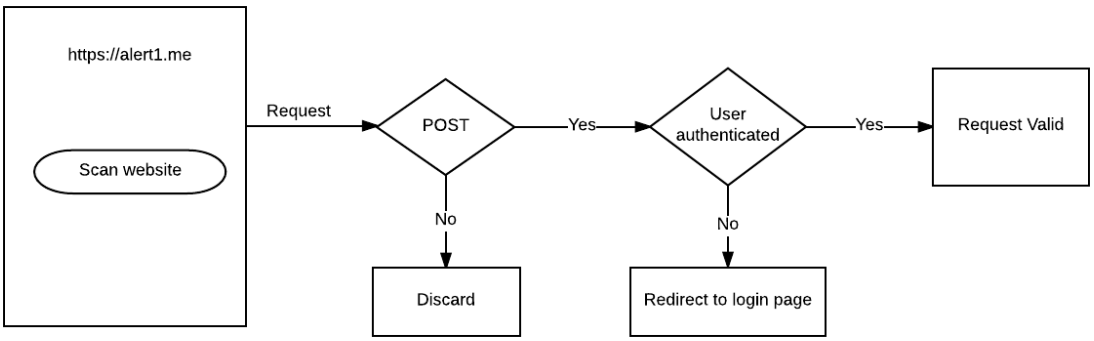

Figure 18. Scan request validation.

Scan request has two parameters which are “URL“ and “type“. As shown in Figure (19), request parameters are considered valid if the submitted URL is valid which is confirmed by checking it against URL regular expression found in Django’s URL validator [32]. Then it checks if the URL exists by sending a GET request and checks if the response code is 200 and finally if the type parameter value is either 0, 1 or 2 which corresponds to single URL, domain and file scan type respectively.

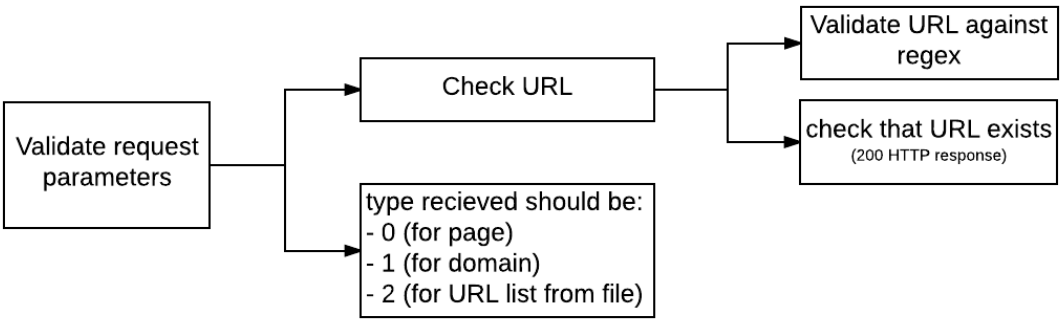

Figure 19. Request parameter validation.

If any of the request parameters are not valid, request will be discarded. After the request and its parameters are validated, the request parameters are passed to add scan function to process it. The add scan process decides the correct scenario for processing scan URL. As shown in Figure (20) below, the process starts by checking if the domain exists in user's domain records. If not, it adds new domain record that is linked to that user and then creates new page record that is linked to that domain.

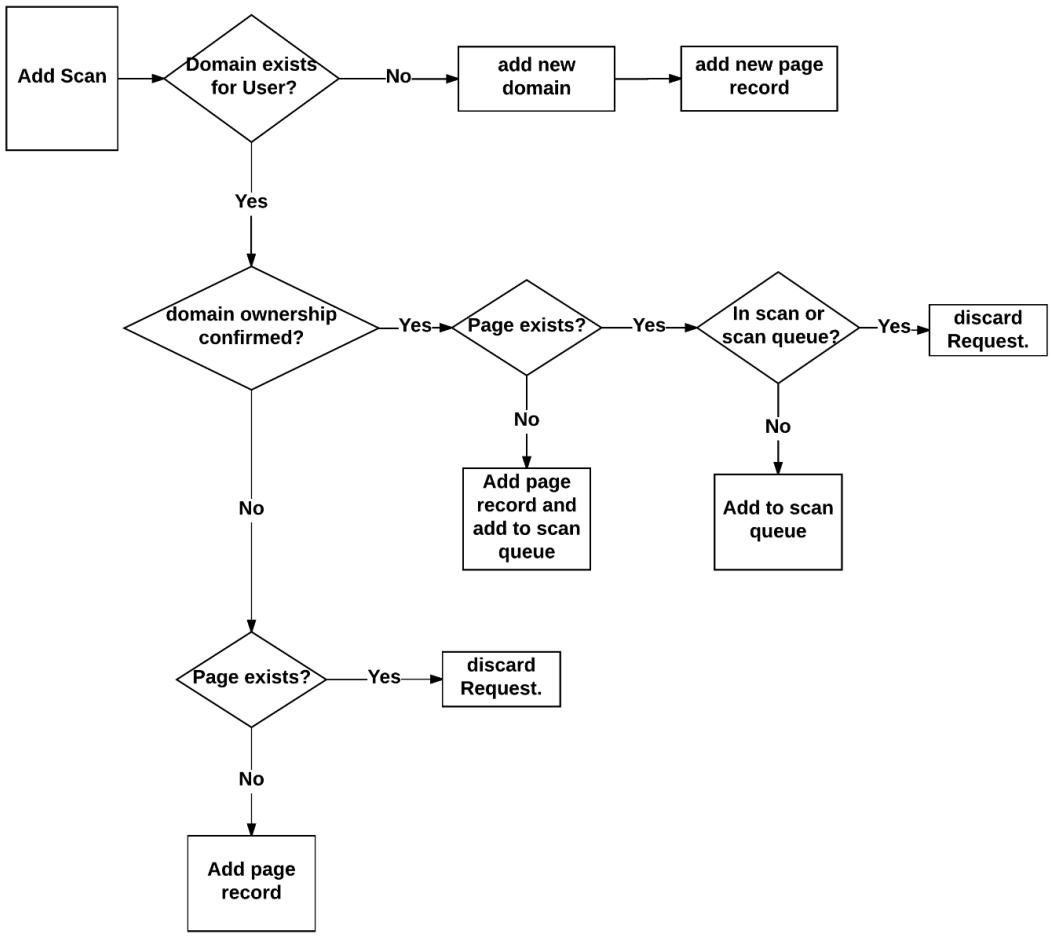

Figure 20. Add scan process.

If the domain already exists for that user, then it checks if the domain was confirmed or not. If it was confirmed and there is not page record with that URL, then it adds new page record and add it to scan queue. On the other hand if page exists, it checks that page status. If it is not in queue or in scan, it means that user was re-scanning an existing page. In that case, it adds that page record to scan queue. But if the page status shows it is in queue or already getting scanned, then the request will be discarded.

If the domain exists and was not confirmed then it checks if the added URL was in the domain's page records or not. If the page exists already then request is discarded, otherwise a new page record for that domain is created.

The scan queue was implemented by using Redis cache server and Django-rq which enables to push function calls from Django that gets queued and executed by the Django-rq process. That approach handles the long time needed by the attack process by pushing all scan request to queue and update scan status and results once they are ready.

Figure (21) shows how Django pushes page's record URL and its ID to scan queue and also update the scan status of that record to 'In queue'. On the other hand Figure (22) shows how Django-rq process pops-up jobs from the queue. First, it updates the scan status for that page record to 'processing'. Then it initiate the scan by calling the attack sequence function. Once scan is done, the page record status is updated to 'scanned'. Then the process checks if there were more jobs in the queue. The maximum work time for any job popped from queue is 600 seconds which is long enough for any scan to finish.

```
def addPageToQueue(url,pageID):
    #add pages to queue
    queue = django_rq.get_queue('pageScan')
    queue.enqueue(scanPageURL, url=url,pageID=pageID)
    page.objects.get(id=pageID).page_scan_status='In Queue'
```

Figure 21. Function that add pages to scan queue

```
@job('default', timeout=600)
def scanPageURL(url,pageID):
    #get page record
    pageRecord= page.objects.get(id=pageID)
    pageRecord.time_scann_started=timezone.localtime(timezone.now())
    pageRecord.page_scan_status='Processing'
    pageRecord.save()
    pageReport=report(page_url=pageRecord)
    pageReport.save()
    attackSequence(url,pageID)
    pageRecord.time_scan_ended=timezone.localtime(timezone.now())
    pageRecord.page_scan_status='Scanned'
    pageRecord.save()
```

Figure 22. Job that pops-up scan request from the queue and calls attack function.

Scan process is also responsible on running the crawler in order to scan top level domain for URLs. Since crawler process can consume long time, it was also done using queue. The crawler returns list of found URL which is then processed to check if each URL exists before

in the database for the user requested the scan. Then it adds the URL to queue only if the URL was not already in queue. See Figure (23).

```
@job('default', timeout=600)
def webSpiderCrawl(domainID):
    #get domain record
    domainRecord= scan.objects.get(id=domainID)
    #get the list of URLS from crawler
    #crawler checks all links in the top level domain that has the same domain
    #then check every found page under that domain for sub links
    #it keeps going through unchecked pages until all pages are checked
    # then it returns list of URLs found
    URLList=crawler(domainRecord.domain)
    #Loop through the URL list to add the non-existed one in the database
    for url in URLList:
        if page.objects.filter(domain=domainRecord,page_url=url).exists():
            pageRecord=page.objects.get(domain=domainRecord,page_url=url)
            if domainRecord.confirm:
                # the page is only added to queue if the it isn't
                #already in queue or under processing
                if pageRecord.page_scan_status!='In Queue' and pageRecord.page_scan_status!='Processing':
                    pageRecord.page_scan_status='In Queue'
                    pageRecord.save()
                    addPageToQueue(url,pageRecord.id)
        else:
            pageRecord= page(domain=domainRecord,page_url=
                url,time_added=timezone.localtime(timezone.now()))
            pageRecord.save()
            if domainRecord.confirm:
                pageRecord.page_scan_status='In Queue'
                pageRecord.save()
                addPageToQueue(url,pageRecord.id)
```

Figure 23. Job that pops-up web spider requests from queue and handles results.

4.5 Attack process:

In the previous steps a URL that needs scanning was validated, attributed to a certain user who confirmed ownership of his domain and finally pushed to scan queue. In this section, steps for attack URL will be explain in details. The attack process receives the URL that needs scanning then it follows the steps defined previously in methodology section to detect XSS which are: identify possible inputs in page, properly inject unique payload into it, check which of these input reflect data on the page, analyse reflection case, identify existing filter, craft payload based on previous data, inject that crafted payload and finally check if the injected code was executed.

In order to have a modular efficient code, we reversed the implementation process by splitting the full task into smaller ones then code each task separately. This approach was used to facilitate constant editing during thesis implementation and to ensure that more features can easily be added in the future. In the following sections we will explain functions and classes created for each small task and in the end we will explain how they are integrated together to create the attack process.

Initialization

The first step in developing the attack process was to create a class that initializes global variables which will be used during the whole process. Initializing the class takes the page ID in order to load the database record of the page to allow updating scan status and create reports. The class also sets up a virtual display so that browser can run in it and finally opens a browser by creating an instance of Selenium's web driver that allow us to control the browser. See Figure (24).

```
class environmentSetup:
    def __init__(self,pid):
        self.display = Display(visible=0, size=(800, 600))
        self.display.start()
        self.chromeDriver= webdriver.Chrome('/usr/bin/chromedriver')
        self.pageToProcess= page.objects.get(id=pid)
        self.pageReport=report.objects.get(page_url=self.pageToProcess)
    def __del__(self):
        self.chromeDriver.quit()
        self.display.stop()
        self.pageReport.save()
```

Figure 24. Environment setup class

Detect inputs

In this part, we will also create a class that has functions for detecting page inputs and attributes that hold input detection results. That allows an object of that class to have attributes with the input detection results.

As shown in Figure (25), the class objects are initialized with two parameter: URL and the object of global environment class. The initialize function starts with detecting parameters found in the URL and save them in class attribute named 'parameterURL' as a dictionary where the URL parameter name is the key. Then it checks if there was hash (#) in the URL and save the result to a class attribute of type bool. It also detects inputs in the page by first opening the page and rendering the response, parse that response by BeautifulSoup to facilitate pulling inputs from the page source then find all the forms in that response and save them. Finally to detect any inputs that are not inside form tags, it will first remove all the forms found from the rendered source then look for any input tag and save them to a list.

```
#input detection class
class inputDetection:
    def detectInputs(self):
        # get the rendered response
        response =self.env.driver.page_source
        # turn it to soup object in order to parse HTML tags easily
        soup = BeautifulSoup(response, 'html.parser')
        # get all forms tags
        formList=soup.find_all('form')
        # remove all found form from the response in order to find inputs withoud forms
        respWithoutForms=str(soup)
        for form in formList:
            respWithoutForms=respWithoutForms.replace(str(form),'')
        regex = r"<input[^>]*"
        self.nonFormInputList = re.findall(regex, respWithoutForms, re.IGNORECASE)
        return formList
    def __init__(self, url,arg):
        super().__init__()
        self.env=arg
        self.url = url
        # get URL parameters and save it to dictionary
        self.parameterURL= dict(parse_qsl(urlparse(url).query))
        self.hashURL='#' in url
        self.nonFormInputList=[]
        self.formList=self.detectInputs()
```

Figure 25. Input detection class.

Check reflection

To check reflection the code needs to inject unique token that might not be found in the rendered response. The token generates random 25 digit of hex values (UUID) which ensure token uniqueness [33] as it is used to generate unique file name for the user to confirm the domain ownership as well as unique payload in testing reflection. See Figure (26)

```
##create token
def createToken():
    return str(uuid.uuid4().hex)
```

Figure 26. Token generator funciton.

After sending a request with the unique generated token, the rendered response needs to be checked to confirm if the token was reflected or not. As shown in Figure (27), function 'tokenReflection' takes the token and rendered page source as its parameters and then parses all the reflection occurrences of the token using regular expressions [34]. The expression detects what you might call reflection string which is the first HTML tag surrounding the token. So if the token was part of a tag attribute like <img src="token">, it will parse the whole image tag. But if it was between two tags like token</>, then it will parse the opening and closing paragraph tag and save it as reflection string. Then it passes each reflection string as well as the injected token to 'detectTokenCase' function which returns value that represent the reflection case based on the position of the token in the HTML. There are three possible cases for a token to appear in HTML. First case (case 0) where the token is inside HTML tag like <img src="token">. Second case (case 1) where the token is between two HTML tags like token . Third case (case 2) where token is inside script tags like <script> token </script>. Finally the function add both reflection string and token case into a dictionary with reflection string as a key then return that dictionary.

```
def detectTokenCase(token,reflectionString):
    # check if token is an attribute inside HTML tag <tag token>
    if re.findall('<[^>]*'+token+'[^>]*>',reflectionString):
        return 0
    # check if token is between tags <tag> token </tag>
    if re.findall('(?<=>)[^<]*'+token+'[^<]*(?=<)',reflectionString):
        # check if the tag is script tag
        if '<script' in reflectionString:
            return 2
        else:
            return 1
def tokenReflection(token,response):
    # find all reflections strings of the token
    reflectionList=re.findall('<(?<=<)[^<]*'+token+'[^>]*>',response)
    tokenDict=dict()
    # set tokenDict value = the token case
    if reflectionList:
        for reflection in reflectionList:
            tokenCase=detectTokenCase(token,reflection)
            tokenDict[reflection]=tokenCase
    return tokenDict
```

Figure 27. Token reflection detection

Identify filters

At this point, the tool managed to identify the possible vulnerable inputs that their data gets reflected in the rendered response and needs to check if there was any encoding or filtering that gets applied on injected data.

There are three functions that helps in identifying possible filters on characters for each reflection string found in the rendered response. The first function identifies character needed to inject XSS payload that could exploit that specific reflection based on the reflection case. The thesis author identified two main meta-character that are used to escape the reflected string if the reflection case is of type 0 where the token is inside an attribute of HTML tag. In that case double quotes (") and single quote (') are the only way to escape the string and inject a malicious code. On the other hand if the reflection was of case 1 where the reflected token is just text inside the tag then less-than sign (<) and greater-than sign (>) are the only way to inject malicious code. If the reflection case is of case 2, then all of the four character might be needed to craft a payload depending on the scenario.

The second function takes the full reflection dictionary and return list of all the characters that need to be checked by the attack function in order to craft load. The function mainly loop through the dictionary and then uses the first function to decide which character is needed for each reflection. These two functions were split since only the first one will be needed while crafting the payload.

The third function takes the reflection dictionary, token, filter dictionary and the character that is being tested to check whether the website filter it or not. The function checks if the character injected after the token was reflected directly, encoded, escaped or any other scenarios.

First, the function creates a sub-dictionary for each reflection string to add filter cases of characters for each reflection string. Then it checks whether the token was followed directly by the character. In that case it adds an entry in the filter dictionary for that reflection string with the character being tested as a key and with value 0. On the other hand to check if the character was encoded, it encodes the character and checks if that encoding exists after the token which makes value added in the dictionary for the character of the reflection string equal 1. If the website injects reverse solidus (commonly known as backslash) before the injected character, then the value will be 2 in that case. Otherwise it injects a value of 3 to the filtered dictionary with keys reflection string and character which indicates unknown case which might include total filtering of the character or double encoding, etc. See Figure (28).

Craft payload

At this point the tool has already identified possible vulnerable inputs, dictionary of reflection strings with their reflection case type and filters for each occurrence of data reflection in the rendered response. All that data will be used to craft possible payloads that should be able to inject code that gets executed. In this version of the thesis the payload will mainly concentrate on popping up an alert box.

The craft payload function takes three arguments: the reflected dictionary which has all the reflection string mapped to their reflection case, the injected token and filter dictionary which has list of sub-dictionaries that has the value of filter case for each reflection string. The function is not that sophisticated at the moment of writing the thesis as it does not craft specific payload for the HTML tag itself but uses three of the most popular events among HTML tags which are on error, on load and on focus as it also passes the auto-focus attribute.

```
# identify characters that needs to be checked based on the reflection case
def identifyXSSCharForReflectionCase(case, reflectionString,token):
    if case==0:
        return ['"',"'"]
    elif case==1:
        return ['<','>']
    else:
        return[]
# get list of characters to check for the full page
def identifyXSSChars(reflectedDict,token):
    XSSCharList=[]
    for reflectionString in reflectedDict:
        charList=identifyXSSCharForReflectionCase
        (reflectedDict[reflectionString], reflectionString,token)
        for charFound in charList:
            if charFound not in XSSCharList:
                XSSCharList.append(charFound)
    return XSSCharList
# Return: 0 for no encoding, 1 URL encoding,
# 2 single reverse solidus escape, 3 for html encoding or others
def identifyCharFilterCase(reflectedDict,token,char):
    filterDictionary=dict()
    for reflectionString in reflectedDict:
        filterDictionary[reflectionString]=dict()
        filterDictionary[reflectionString]['case']=reflectedDict[reflectionString]
        #check if it exists without encoding
        if token+char in reflectionString:
            filterDictionary[reflectionString][char]=0
        elif token+urllib.parse.quote(char) in reflectionString:
            filterDictionary[reflectionString][char]=1
        elif token+'\\'+char:
            filterDictionary[reflectionString][char]=2
        else:
            filterDictionary[reflectionString][char]=3

    return filterDictionary
```

Figure 28. Identify filter functions.

The last attribute found in the payload which is ‘x=none’ is just a dummy attribute that will get ignored by most browser as the rest of the tag would still have closing single or double quotation. If the dummy attribute was not added the browser would discard the injected event itself. See Figure (29).

At this point, the craft payload function also focuses on crafting payloads for cases 0 and 1 which are when token is inside HTML tag as an attribute or between two HTML tags respectively.

Check injection

Check injection function takes the object from environment class which contains the web driver object that control the website. Then it uses wait function that is implemented in web driver which waits until an event in the browser happens which in our case is an alert box. If the alert box was found then it accepts it and returns true. If it was not found or any exception occurs then it returns false. See Figure (30).

```
def craftPayload(filterDict,possiblePayloads):
    for reflecttionString in filterDict:
        # if the token is inside HTML tag as an attribute <tag src="token">
        if filterDict[reflecttionString]['case']==0:
            # if no-encoding
            if "'" in filterDict.get(reflecttionString,{}):
                if filterDict[reflecttionString]["'"]==0:
                    possiblePayloads.append
                    ("""' onerror=alert(1) onload=alert(1) onfocus =alert(1) autofocus x=none""")
                    possiblePayloads.append("""'><img src=x onerror=alert(1)>""")
                    possiblePayloads.append("""'><script>alert(1)</script>""")
                if filterDict[reflecttionString]["'"]==2:
                    possiblePayloads.append
                    ("""\\' onerror=alert(1) onload=alert(1) onfocus =alert(1) autofocus x=none""")
                    possiblePayloads.append("""\\'><img src=x onerror=alert(1)>""")
                    possiblePayloads.append("""\\'><script>alert(1)</script>""")
            if '"' in filterDict.get(reflecttionString,{}):
                if filterDict[reflecttionString]['"']==0:
                    possiblePayloads.append
                    (""" onerror=alert(1) onload=alert(1) onfocus =alert(1) autofocus x=none""")
                    possiblePayloads.append(""""><img src=x onerror=alert(1)>""")
                    possiblePayloads.append(""""><script>alert(1)</script>""")
                if filterDict[reflecttionString]['"']==2:
                    possiblePayloads.append
                    ("""\" onerror=alert(1) onload=alert(1) onfocus =alert(1) autofocus x=none""")
                    possiblePayloads.append("""\"><img src=x onerror=alert(1)>""")
                    possiblePayloads.append("""\"><script>alert(1)</script>""")
        # if the token is between HTML tags <tag> token</tag>
        elif filterDict[reflecttionString]['case']==1:
            possiblePayloads.append("""<script>alert(1)</script>""")
            possiblePayloads.append("""<img src=x onerror=alert(1)>""")
            possiblePayloads.append("""\\<img src=x onerror=alert(1)\\>""")
```

Figure 29. Craft payload function.

```
def checkInjection(env):
    #check alert with chrome
    try:
        WebDriverWait(env.chromeDriver, 3).until(EC.alert_is_present())
        alert = env.chromeDriver.switch_to_alert()
        alert.accept()
        return True
    except TimeoutException:
        return False
    except :
        return False
```

Figure 30. Check injection function.

Main attack function

The main attack function initializes an object of the environment setup class. Then loads the URL in the browser to update the URL value. As in some cases the server would automatically add parameters to the before sending back the response to browser. Then it detects page inputs by creating an object of page input class with page's URL that needs scanning. It splits the attack process into three different attacks that target three different types of inputs in the page: hash, URL parameters and rest of the page inputs. See Figure (31).

Hash attack

In this section the environment variables were initialized and the tool tries to check if the hash value reflects in the rendered response or not. It starts by injecting a token in the hash URL. Then open the webpage and check if the token was reflected by calling the token

```
def attackSequence(url,PID):
    #init golbal variables
    env=environmentSetup(PID)
    #load the URL in order to get automatically added URL parameters
    env.chromeDriver.get(url)
    url=env.chromeDriver.current_url
    #detect URL inputs
    inputsInPage=inputDetection(url,env)
    # if hash exist launch hash attack
    hashAttack(url,env)
    #url parameters
    if inputsInPage.parameterURL:
        urlParameterAttack(url, inputsInPage.parameterURL,env)
    #check page inputs
    pageInputsAttack(url,inputsInPage,env)
```

Figure 31. Main attack function.

reflection function. If there were reflection cases, then it identifies all characters that would be needed to carry out an XSS attack by calling 'identifyXSSChars' function. Then it injects each of the character after the token and pass the rendered response to 'identifyFilterCase' function which creates dictionary of sub-dictionaries to all the character filter cases based on each reflection string found in reflection dictionary. The craft payload function uses all this data to craft a list of payload as mentioned in the previous section.

URL parameters attack

The function injects token in each parameter found in the URL then follows the same procedure found in the hash attack function by checking reflection list for each injected token and craft list of payload for each parameter. Then it checks if any of these injections managed to pop-up an alert box successfully. See Figure (32).

Page inputs attack

It checks each form found in the page and detect all possible input inside it and then it injects unique input for each one of them. The function checks every form separately in order to guarantee that the form will be submitted correctly. This part of the tool is still under development as fuzzing and submitting forms are more complicated; as some forms might have lots of restriction on input type or even have CAPTCHA to check if the form was submitted by human and protect website against bot attacks.

```
def urlParameterAttack(url,inputsInPage,env):
    # loop through each parameter
    for parameter in inputsInPage.parameterURL:
        possiblePayloads=[]
        token=createToken()
        #get the rendered response
        injectURL=url.replace('='+inputsInPage.parameterURL[parameter],'='+token)
        # print('injectedURL'+str(injectURL))
        env.driver.get(injectURL)
        renderedRes=env.driver.page_source
        reflectedDict=tokenReflection(token,renderedRes)
        # for every reflection string in the rendered respinse
        # find the characters needed to be checked for XSS
        # injection then check if there is filters in the page and get the list
        # of possible payloads that could be injected in that parameter
        if reflectedDict:
            charToCheckFilters=identifyXSSChars(reflectedDict,token)
            for char in charToCheckFilters:
                injectURLWithFilterChar=injectURL.replace(token,token+char)
                env.driver.get(injectURLWithFilterChar)
                renderedRes=env.driver.page_source
                reflectedDictWithFilters=tokenReflection(token, renderedRes)
                filterDict=identifyCharFilterCase(reflectedDictWithFilters,token, char)
                craftPayload(filterDict,possiblePayloads)
        # check the same as before using only HTML found in HTTP response
        httpRes=env.requests.get(injectURL).content.decode('UTF-8')
        reflectedDictHTTP=tokenReflection(token,httpRes)
        if reflectedDict:
            charToCheckFilters=identifyXSSChars(reflectedDict,token)
            for char in charToCheckFilters:
                injectURLWithFilterChar=injectURL.replace(token,token+char)
                httpRes=env.requests.get(injectURLWithFilterChar).content.decode('UTF-8')
                reflectedDictWithFilters=tokenReflection(token, httpRes)
                filterDict=identifyCharFilterCase(reflectedDictWithFilters,token, char)
                craftPayload(filterDict,possiblePayloads)
        # check if any of the payload manage to exploit a vulnerability
        for payloadFound in possiblePayloads:
            injectURL=url.replace('='+inputsInPage.parameterURL[parameter],'='+payloadFound)
            env.driver.get(injectURL)
            print (injectURL)
            inj=checkInjection(env)
            print ("injection success:"+str (inj))
```

Figure 32. URL parameters attack function

5 Test Cases:

In this section the author sets up some test cases which have DOM-XSS in order to validate the approach followed by this thesis to detect DOM-XSS vulnerabilities. The cases were chosen from either online DOM-XSS challenges or based on common developer's approaches in developing websites that cause DOM-XSS. The cases covers possible scenarios for reflected DOM-XSS. Since developer's approaches are identified based on the developing experience of the thesis author, he used snippets of public code found on popular development community forums.

5.1 Case 1:

The first case is taken from the 3rd XSS challenge created by Google [35] which addresses the fact that data after fragment identifier (commonly known as URL hash, #) are not passed in the request URL to server but only processed at the client side. Usually URL hash is used to access elements in the page using their ID or sometimes to pass data within the page while preventing the page from reloading. In the challenge, when user change tabs, the anchor tag adds the correspondent tab number after the hash in the URL. Then there is a script function in the page that monitors tab changes and parses data found in the URL hash and uses it as a part of the source attribute of an image tag and then injects that image in the page dynamic content. See Figure (33).

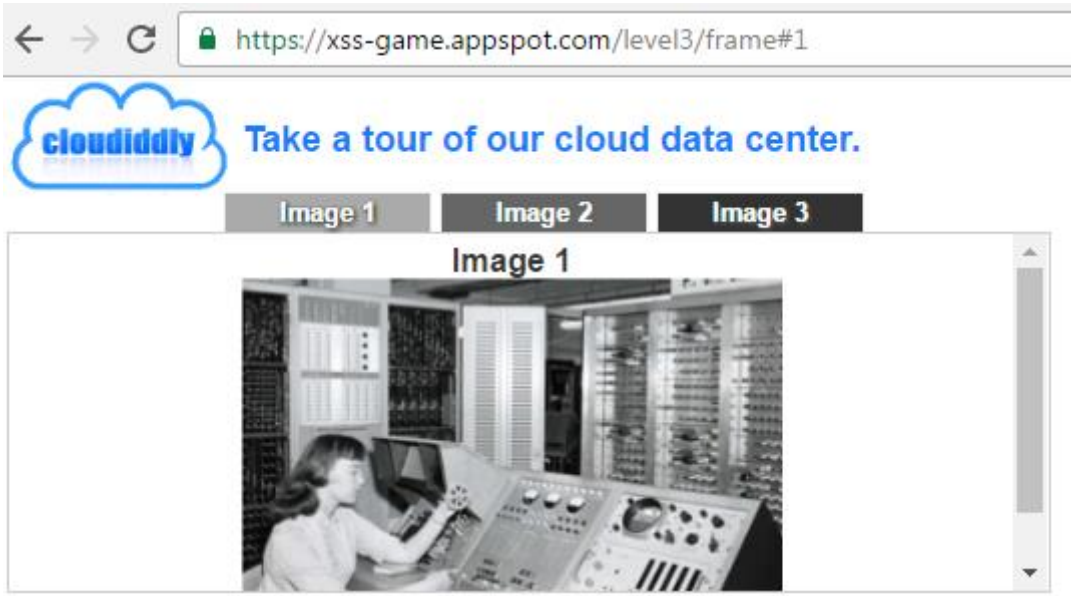

Figure 33. 3rd XSS challenge by Google.

Figure (34) shows that the HTML from the received response does not have any image tags inside the tab content div. The image tag is injected only when the browser render that response. As on window load event the script parse the tab number found after the URL hash and send it to choose tab function. If the URL does not have any hash data, it will send 1 by default. Choosing tab function takes that tab number and use it as part of the source of an image tag. The script does not encode data in the URL hash, which allow crafted data in the URL hash to be injected as part of source attribute inside image tag. This allows hacker to inject malicious string that starts with a single quote that can escape the string of source attribute then add another attribute like (onerror) and inject malicious javascript within that attribute value that will get executed when rendering the image fails due to wrong source

value. Figure (35) shows an example of a malicious crafted link by a hacker for the first case that will pop-up an alert box once the victim clicks on it.

```
<img id="logo" src="">
      <span>Take a tour of our cloud data center.</span><br>
    </div>

    <div class="tab" id="tab1" onclick="chooseTab('1')">Image 1</div>
    <div class="tab" id="tab2" onclick="chooseTab('2')">Image 2</div>
    <div class="tab" id="tab3" onclick="chooseTab('3')">Image 3</div>

    <div id="tabContent"> </div>
<script>
      function chooseTab(num) {
        // Dynamically load the appropriate image.
        var html = "Image " + parseInt(num) + "<br>";
        html += "<IMG src='https://xss-game.appspot.com/static/level3/cloud" + num + ".jpg' />";
        $('#tabContent').html(html);

        window.location.hash = num;

        // Select the current tab
        var tabs = document.querySelectorAll('.tab');
        for (var i = 0; i < tabs.length; i++) {
          if (tabs[i].id == "tab" + parseInt(num)) {
            tabs[i].className = "tab active";
            } else {
            tabs[i].className = "tab";
          }
        }

        // Tell parent we've changed the tab
        top.postMessage(self.location.toString(), "*");
      }

      window.onload = function() {
        chooseTab(self.location.hash.substr(1) || "1");
      }
```

Figure 34. HTML of 3rd XSS challenge by Google.

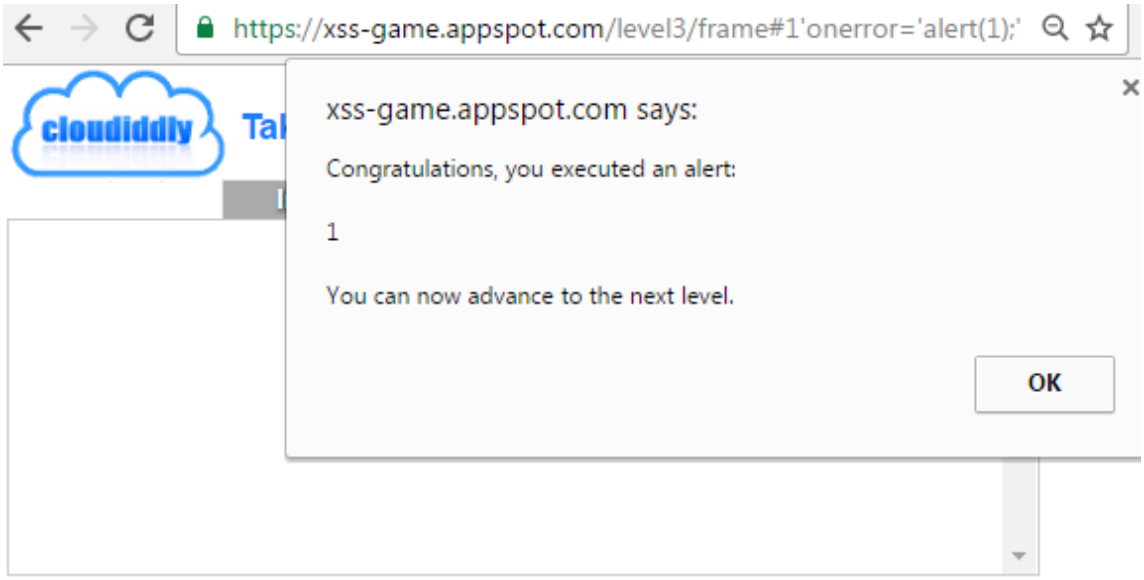

Figure 35. Exploit of 3rd XSS challenge by Google.

The webpage for that case can be found at- https://xss-game.appspot.com/level3/frame#1

5.2 Case 2:

The second case addresses the approach of passing data to front-end through URL parameters. The HTML response contains scripts that parse these data and form the webpage based on it. When browser load the web page it renders the HTML response causing page scripts to be executed which will then parse data found in the URL and inject it somehow inside the page dynamic content. If parse data is not properly code, hacker can inject malicious script that will then be executed by the browser as a legitimate page scripts.

The case scenario is a web page that shows profile data of the logged-in user. The server passes user ID as URL parameter. As some websites save user profile pictures with their ID (‘/images/<userID>.jpg). The page has script that takes the parse URL parameter then use that data to form the image source attribute as the first case. See Figure (36).

Figure 36. Case 2 scenario.

That approach of passing values through URL parameters and that specific scenario of loading profile picture based on the passed data can be found in multiple websites whether it is done on the backend or frontend.

In Figure (37), the function that parses the URL parameter was taken from an answer on Stackoverflow [36] with 5440 vote as the thesis was written, which makes that code a very popular one that might have been used by thousands of developers on different websites. The problem with that snippet of code is that it misses doing proper encoding and that not every developer who will use the function understands that data returned by that function still needs proper encoding or it will cause security vulnerability.

The other part of the problem that is causing the XSS vulnerability is adding that data to page using HTML function in JQuery [37] or innerHTML in javascript [38] without doing proper encoding. There are multiple of resources including Stackoverflow [39] and others [40] offering that solution to developer without explaining possible security risks behind it.

Figure (38) shows an example of a malicious crafted link by a hacker for the first case that will pop-up an alert box once the victim click on it.

```
<!DOCTYPE HTML>
<html>
<head>
<script   src="js/jquery-1.12.4.js"  ></script>
</head>
<body>
<h1></h1>
<div id="res"></div>
<script>
function getUrlParameter(sParam) {
    var sPageURL = decodeURIComponent(window.location.search.substring(1)),
        sURLVariables = sPageURL.split('&'),
        sParameterName,
        i;
    for (i = 0; i < sURLVariables.length; i++) {
        sParameterName = sURLVariables[i].split('=');
        if (sParameterName[0] === sParam) {
            return sParameterName[1] === undefined ? true : sParameterName[1];
        }
    }
};
$(document).ready(function(){
    s=getUrlParameter('accID')
    if (s){
        profImg='<img src="imgs/'+ s +'.jpg" width="300pc"/>'
        $('#res').html(profImg);
    }
});
</script>
</body>
</html>
```

Figure 37. Case 2 HTML response.

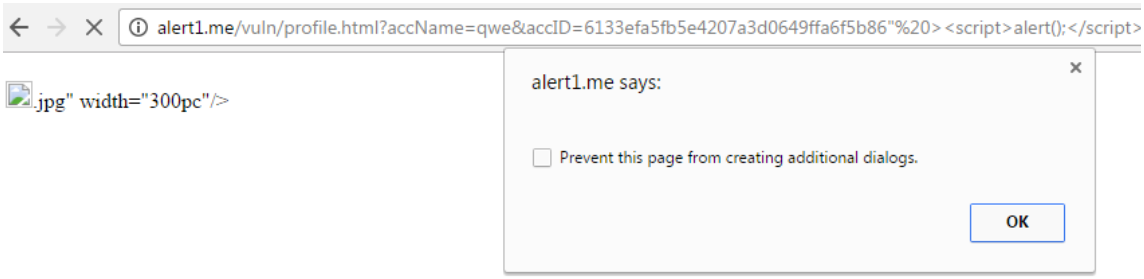

Figure 38. Exploit of Case 2 scenario.

The webpage for that case can be found at- http://alert1.me/vuln/profile.html?accName=qwe&accID=6133efa5fb5e4207a3d0649ffa6f5b86

5.3 Case 3

In this case, data will be passed through URL parameter like in case 2. But once the browser loads the page, these data will be sent back to server again through an ajax [41] request. The web server will reply to that request with a json response [42]. Then the page script will somehow process that response and injects part of the data into the page dynamic content.

Such an approach is being widely used in modern websites in order to minimize page loading time and increase performance of the website. As this approached introduced page dynamic loading where the developers do not have to send back all data requested by user but split them into chunks and load them on request. This approach is user in social network

website to load posts in a web page, as loading all the posts at once will take long time and will probably crash the browser.

In our case, there is a search bar that allows user to search for keywords in the website. When a user searches for a keyword the form will send GET request which adds the keyword to a URL parameter. Then after the browser loads that page, script in the page will parse the URL parameter and send that data as an ajax request to backend. The backend will search database for that keyword and send back search results to frontend. Then frontend scripts will add the search results to the page dynamic content. If all these previous steps do not properly encode data parsed from URL parameter, there will be a DOM-XSS vulnerability. See Figure (39).

In this case, the author used the same function from case 2 to parse the search URL parameter. The page script sends an ajax request that include the search parameter using JQuery and then populate the page with the results received from the server of that request. All these

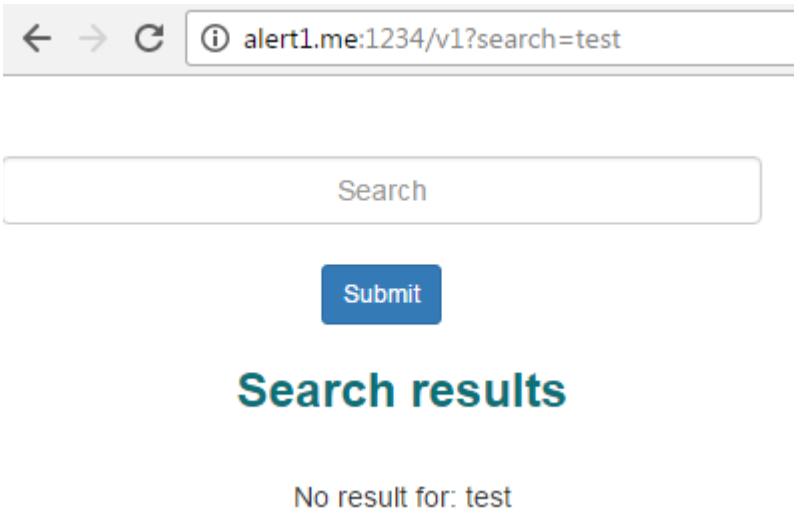

Figure 39. Case 3 scenario.

steps do not do any proper encoding of the data entered by the user. See Figure (40) for the case code. On the other hand, Figure (41) shows an example of a malicious crafted link by a hacker for the third case that will pop-up an alert box once the victim click on it.

```
<div class="row text-center" >
    <div id="result">
    </div>
</div>
<script>
function getParameterByName(name, url) {
    if (!url) {
      url = window.location.href;
    }
    name = name.replace(/[\[\]]/g, "\\$&");
    var regex = new RegExp("[?&]" + name + "(=([^&#]*)|&|#|$)"),
        results = regex.exec(url);
    if (!results) return null;
    if (!results[2]) return '';
    return decodeURIComponent(results[2].replace(/\+/g, " "));
}
$(document).ready(function(){
  s=getParameterByName('search',window.location.href);
  if (s){
        $.ajax({
            url:  'http://alert1.me:1234/vuln1',
            type: 'POST',
            contentType: 'application/json; charset=utf-8',
            data: JSON.stringify({'search':s}),

            error: function() {
                console.log('error');
            },
            success: function(data) {
                console.log(data);
            }
        });
    $('#result').html('You have searched for: '+s);
  }

});
</script>
```

Figure 40. Case 3 code.

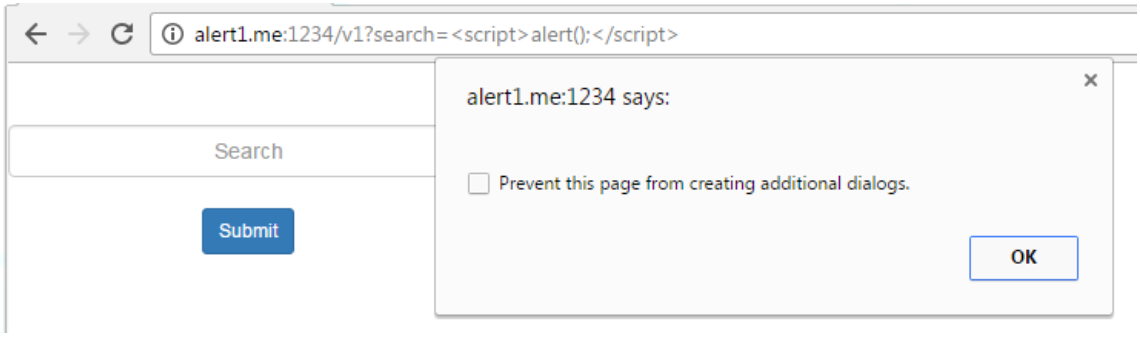

Figure 41. Exploit of Case 3 scenario.

The webpage for that case can be found at- http://alert1.me/v1?search=test

6 Results

In this section, the author scans the previously mentioned test cases by the tool as well as different web scanners. Then he compares the results of each test cases from different scanners. The comparison metrics will be vulnerability detection (true positive – TP), false detection (false positive - FP), scan duration and number of request sent during the scan process.

Although test cases might not cover every possible scenarios that causes DOM-XSS vulnerabilities, but they cover wide range of possible reflected DOM-XSS scenarios that can proof that the tool could detect the vulnerability.

In order for that comparison to be correct, the scanners needed to be picked carefully and to specific criteria. As there is a big list of general web scanners that scans for different web vulnerabilities as well as lots of academic and experimental scanners. So the first criteria of choosing the scanner is that it has XSS detection profile or capability. Next, the scanner's licence should be either open-source, free to use or at least has a trial period in case of commercial scanners. At this point there were still lots of scanners to be considered, so the author tried to favour the most recommended ones recognized by OWASP [43].

The tool results was compared against 2 scanners that has modules for DOM-XSS detection: OWASP Xenotix XSS exploit framework and Acunetix which both use taint-tracking and dynamic taint-tracking respectively. Comparison also includes another 2 scanner that use the generic method of only analysing the received HTML response: W3af and Vega.

Results of scanning Case 1

Table 2. Result of scanning case 1 by different web scanners.

Scanner name	Vulnerability found (TP)	False vulnerabilities (FP)	Scan duration (seconds)	# requests
Thesis tool (X55)	1	0	97	19
Acunetix WVS	1	0	265	41
Xenotix	1	0	17	N/A
Vega	0	0	47	23
W3af	0	0	45	N/A

Results of scanning Case 2

Table 3. Result of scanning case 2 by different web scanners.

Scanner name	Vulnerability found (TP)	False vulnerabilities (FP)	Scan duration	# requests
Thesis tool (X55)	1	0	145	17
Acunetix WVS	0	0	65	96
Xenotix	1	0	12	N/A

Vega	0	0	65	34
W3af	0	0	73	N/A

Results of scanning Case 3

Table 4. Result of scanning case 3 by different web scanners.

Scanner name	Vulnerability found (TP)	False vulnerabilities (FP)	Scan duration (seconds)	# requests
Thesis tool (X55)	1	0	169	33
Acunetix WVS	0	0	38	88
Xenotix	0	0	13	N/A
Vega	0	0	57	76
W3af	0	0	45	N/A

The tool results show that it consumes more time than other tools but it was successful to find the DOM-XSS vulnerability. The consumed time is expected due to the overhead needed to open a browser and time of loading the webpage itself. Time consuming could be an arguable metrics against the value of finding the bug itself in an approach that could be integrated in any web scanner. In conclusion, the results prove that the methodology could successfully detect DOM-XSS vulnerability.

7 Summary

In this section, the thesis author drew conclusions based on the results of scanning test cases with different web scanners and the tool developed by the thesis author. The author also discussed importance of developing such tool with different penetration testers and defined possible valuable information that would help them to carry-out manual tests. Also, the author defines possible future work that could enhance the tool detection for XSS vulnerabilities.

7.1 Conclusion

This thesis proposes an alternative methodology to automate detection of DOM cross-site scripting vulnerabilities by adding an extra layer between receiving the HTML code from the server response and analysing it for vulnerabilities. The layer is an actual browser that renders the received HTML response. Rendering the response allows the execution of any scripts in the page which can change content of the page and might cause DOM-XSS vulnerabilities. Then the tool scans that rendered response for possible cross-site scripting vulnerabilities. Since the tool's methodology is a build-up on the methodology followed by general web scanners to detect XSS, it can be integrated in any web scanners allowing them to effectively discover all kind of cross-site scripting.

The author developed a web-based tool based on the proposed methodology to proof that it is a practical one. He also created 3 scenarios that cover wide range of possible reflected DOM-XSS scenarios. Test cases were based on either online hacking challenges that contained DOM-XSS vulnerabilities or popular code found on developer's community forums. The author compared results from scanning these 3 test cases by the tool as well as other web scanners. The results showed that the tool is capable of finding reflected DOM-XSS against the other tools. The drawback of the tool is that it consumes time due to the actual fact that it has to open an actual browser and load pages through it.

Finally, this thesis methodology could contribute to enhance general web scanners to allow them discover all kind of XSS including DOM without the need to run dynamic taint scan on the page's scripts. It also contributes a tool that can detect reflective DOM-XSS vulnerabilities and have successfully claimed bounty from TransferWise for DOM-XSS vulnerability found in their website through their un-official bug bounty program.

7.2 Future work

Since the developed tool by the thesis author was meant as a proof of concept of the followed methodology, the tool needs more future developing contribution in the following criteria.

Crafting payload

The craft payload function generates proper list of payloads based on the position of the reflected token in the rendered response and list of filtered/encoded characters/words detected by the tool. The function needs more work to cover more possible scenarios and combination of HTML tags along with their possible attributes as well as the filtered/encoded characters/words.

Confirm injection

The craft payload function creates a list of payload that try to launch an alert box as a proof of concept of XSS vulnerability. Such an approach might not always work as some XSS firewall or filters might detect the alert keyword and deny the request. Future work could concentrate on different approaches of confirming that injection was successful by using

other resources accessible by script like using “console.log () “which output string or object to browser’s console.

Different browsers

The tool uses an actual browser to render the web page which allows any script in the page to be executed. At the moment, the tool uses only Chromium browser, which served well as a proof of concept. But future work should include using different browsers to scan the page and give detailed results of the scan on each browser. Because certain payloads might not work in specific browsers as some browsers have XSS detection plugin like XSS auditor in Chrome, XSS filter in Microsoft Edge or even in Firefox which encodes the URL by default.

Fuzzing forms

Checking forms in page and fuzzing them with data to check if their fields are vulnerable is still under development. Form submission is an actual hard process as most forms in real websites would have constrains on some field whether only digits or letters or specific format. Also, they might have CAPTCHA to challenge bots submissions which makes it even more complicated process.

8 References

[1] "Excess XSS: A comprehensive tutorial on cross-site scripting", Excess-xss.com, 2016. [Online]. Available: http://excess-xss.com/. [Accessed: 30- Nov- 2016].

[2] "Cross-site scripting (XSS) - OWASP", Owasp.org, 2016. [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS). [Accessed: 30- Nov- 2016].

[3] "[DOM Based Cross Site Scripting or XSS of the Third Kind] Web Security Articles - Web Application Security Consortium", Webappsec.org, 2016. [Online]. Available: http://www.webappsec.org/projects/articles/071105.shtml. [Accessed: 30- Nov- 2016].

[4] "Links in HTML documents", W3.org, 2016. [Online]. Available: https://www.w3.org/TR/html4/struct/links.html#h-12.1. [Accessed: 30- Nov- 2016].

[5] Owasp, "Server XSS/Client XSS vs. Stored/Reflected XSS categories". 2013. [Online]. Available: https://www.owasp.org/images/c/c6/Server-XSS_vs_Client-XSS_Chart.jpg [Accessed: 30- Nov- 2016]

[6] D. Stuttard and M. Pinto, The web application hacker's handbook, 1st ed. Indianapolis: Wiley, 2011.

[7] Edgescan, "2015 Vulnerability Statistics Report", [Online]. Available: https://www.edgescan.com/assets/docs/reports/2015-edgescan-Stats-Report-(2015)-v5.pdf [Accessed: 30- Nov- 2016].

[8] Whitehat security, "Website Security Statistics Report", 2015, [Online]. Available: https://info.whitehatsec.com/rs/whitehatsecurity/images/2015-Stats-Report.pdf [Accessed: 30- Nov- 2016].

[9] "List of Bug Bounty Programs INTERNATIONAL 477+ OFFICIAL - Bug Bounty Sheet VULNERABILITY LAB", Vulnerability-lab.com, 2016. [Online]. Available: https://www.vulnerability-lab.com/list-of-bug-bounty-programs.php. [Accessed: 30- Nov- 2016].

[10] "What is the Document Object Model?", W3.org, 2016. [Online]. Available: https://www.w3.org/TR/WD-DOM/introduction.html. [Accessed: 30- Nov- 2016].

[11] P. Sexton, "How a webpage is loaded and displayed", Varvy.com, 2016. [Online]. Available: https://varvy.com/pagespeed/display.html. [Accessed: 30- Nov- 2016

[12] J. Bau, E. Bursztein, D. Gupta and J. Mitchell, "State of the Art: Automated Black-Box Web Application Vulnerability Testing", 2010 IEEE Symposium on Security and Privacy, pp. 322-354, 2010.

[13] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, P. Saxena. “DEXTERJS: Robust Testing Platform for DOM-Based XSS Vulnerabilities”, In ESEC/FSE Conference, ACM, 2015.

[14] S. Di Paola. “DominatorPro: Securing Next Generation of Web Applications”. https://dominator.mindedsecurity.com, 2012.

[15] P. Saxena, S. Hanna, P. Poosankam, and D. Song. “FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications”. In Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 2010.

[16] "OWASP Xenotix XSS Exploit Framework - OWASP", Owasp.org, 2016. [Online]. Available: https://www.owasp.org/index.php/OWASP_Xenotix_XSS_Exploit_Framework. [Accessed: 22- Dec- 2016].

[17] S. Lekies, B. Stock, and M. Johns. “25 Million Flows Later - Large-scale Detection of DOM-based XSS”. In 2013 ACM SIGSAC Conference on Computer and Communications Security, Berlin, Germany, ACM, 2013.

[18] "Better DOM-based XSS Vulnerabilities Detection - Acunetix", Acunetix, 2016. [Online]. Available: http://www.acunetix.com/websitesecurity/improving-dom-xss-vulnerabilities-detection/. [Accessed: 22- Dec- 2016].

[19] "Understanding XSS Auditor - Virtue Security", Virtue Security, 2016. [Online]. Available: https://www.virtuesecurity.com/blog/understanding-xss-auditor/. [Accessed: 30- Nov- 2016].

[20] "WebDriver: Advanced Usage — Selenium Documentation", Seleniumhq.org, 2016. [Online]. Available: http://www.seleniumhq.org/docs/04_webdriver_advanced.jsp. [Accessed: 30- Nov- 2016].

[21] "SeleniumHQ/selenium", GitHub, 2016. [Online]. Available: https://github.com/SeleniumHQ/selenium/issues. [Accessed: 30- Nov- 2016].

[22] "DigitalOcean: Cloud computing designed for developers", DigitalOcean, 2016. [Online]. Available: https://www.digitalocean.com/. [Accessed: 30- Nov- 2016].

[23] "The Postfix Home Page", Postfix.org, 2016. [Online]. Available: http://www.postfix.org/. [Accessed: 30- Nov- 2016].

[24] "beautifulsoup4 4.5.1 : Python Package Index", Pypi.python.org, 2016. [Online]. Available: https://pypi.python.org/pypi/beautifulsoup4. [Accessed: 30- Nov- 2016].

[25] "Django documentation | Django documentation | Django", Docs.djangoproject.com, 2016. [Online]. Available: https://docs.djangoproject.com/en/1.10/. [Accessed: 30- Nov- 2016].

[26] "Django-RQ", GitHub, 2016. [Online]. Available: https://github.com/ui/django-rq. [Accessed: 30- Nov- 2016].

[27] A. Antukh, "django-redis documentation", Niwinz.github.io, 2016. [Online]. Available: https://niwinz.github.io/django-redis/latest/. [Accessed: 30- Nov- 2016].

[28] "The Chromium Projects", Chromium.org, 2016. [Online]. Available: https://www.chromium.org/. [Accessed: 30- Nov- 2016].

[29] "PyVirtualDisplay 0.2.1 : Python Package Index", Pypi.python.org, 2016. [Online]. Available: https://pypi.python.org/pypi/PyVirtualDisplay. [Accessed: 30- Nov- 2016].

[30] "XVFB", X.org, 2016. [Online]. Available: https://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml. [Accessed: 30- Nov- 2016].

[31] "User authentication in Django | Django documentation | Django", Docs.djangoproject.com, 2016. [Online]. Available: https://docs.djangoproject.com/en/1.10/topics/auth/. [Accessed: 30- Nov- 2016].

[32] "Validators | Django documentation | Django", Docs.djangoproject.com, 2016. [Online]. Available: https://docs.djangoproject.com/en/1.10/ref/validators/. [Accessed: 30- Nov- 2016].

[33] "RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace", Tools.ietf.org, 2016. [Online]. Available: https://tools.ietf.org/html/rfc4122#page-14. [Accessed: 30- Nov- 2016].

[34] "Regular-Expressions.info - Regex Tutorial, Examples and Reference - Regexp Patterns", Regular-expressions.info, 2016. [Online]. Available: http://www.regular-expressions.info/. [Accessed: 30- Nov- 2016].

[35] "Xss-Game", Xss-game.appspot.com, 2016. [Online]. Available: https://xss-game.appspot.com/level3. [Accessed: 30- Nov- 2016].

[36] "How can I get query string values in JavaScript?", Stackoverflow.com, 2016. [Online]. Available: http://stackoverflow.com/questions/901115/how-can-i-get-query-string-values-in-javascript?rq=1. [Accessed: 30- Nov- 2016].

[37] "JQuery API Documentation", Api.jquery.com, 2016. [Online]. Available: http://api.jquery.com/html/. [Accessed: 30- Nov- 2016].

[38] "HTML DOM innerHTML Property", W3schools.com, 2016. [Online]. Available: http://www.w3schools.com/jsref/prop_html_innerhtml.asp. [Accessed: 30- Nov- 2016].

[39] "How to append data to div using javascript?", Stackoverflow.com, 2016. [Online]. Available: http://stackoverflow.com/questions/5677799/how-to-append-data-to-div-using-javascript. [Accessed: 30- Nov- 2016].

[40] V. Liu, "How to dynamically add content to a div", Random Snippets, 2016. [Online]. Available: http://www.randomsnippets.com/2008/04/14/how-to-dynamically-add-content-to-a-div-via-javascript/. [Accessed: 30- Nov- 2016].

[41] "JQuery API Documentation | AJAX", Api.jquery.com, 2016. [Online]. Available: http:///jquery.ajax/. [Accessed: 30- Nov- 2016].

[42] "JSON Example", Json.org, 2016. [Online]. Available: http://json.org/example.html. [Accessed: 30- Nov- 2016].

[43] "Category:Vulnerability Scanning Tools - OWASP", Owasp.org, 2016. [Online]. Available: https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools. [Accessed: 30- Nov- 2016].

Appendix

I. Abbreviation

URL	Uniform Resource Locator
HTML	Hypertext Markup Language
XSS	Cross-site scripting
DOM	Document object model
CSSOM	Cascaded Style Sheet Object Model
VPS	Virtual Private Server
CPU	Central Processing Unit
SMTP	Simple Mail Transfer Protocol
HTTP	Hypertext Transfer Protocol
MiM	Man in the Middle

II. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Wael AbuSeada**,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Alternative Approach to Automate Detection of DOM-XSS Vulnerabilities,

(title of thesis)

supervised by Olaf Manuel Maennel and Raimundas Matulevicius,

(supervisor's name)

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **22.12.2016**