b
UNIVERSITÄT
BERN

Finding and Mitigating Cross-Site Scripting Attack Vectors

Testing different Web Application Security Scanners

Bachelor Thesis

Rafael Burkhalter
from
Rüegsau BE, Switzerland

Philosophisch-naturwissenschaftliche Fakultät
der Universität Bern

09. April 2021

Dr. Mohammad Ghafari
Prof. Dr. Oscar Nierstrasz
Software Composition Group

Institut für Informatik
University of Bern, Switzerland

Abstract

The purpose of this thesis is to determine the efficacy and usability of different popular security scanners for web applications. The main focus lies on testing their ability to find cross-site scripting vulnerabilities, i.e. vulnerabilities arising when user input isn't properly sanitized. To analyze the scanners various criteria are taken into account mainly completeness of the findings, ease of use and installation effort. In a second part an overview on how to analyze a scanner's result and how Cross-Site Scripting attacks can be mitigated is given.

Contents

1
Introduction to Cross-site Scripting

To create dynamic web pages the web-browser Netscape Navigator introduced the scripting language LiveScript (now ECMAScript, commonly known as JavaScript) which could execute client-side scripts and therefore give developers the possibility to create more interactive elements on formerly static pages [33]. With the introduction of JavaScript also came the security flaw now known as Cross-site Scripting (XSS), since the scripts used by Web pages are marked as such by using the HTML-tags '<script>' and '</script>' a malefactor can insert malicious code into the web page that is shown to a user. This works by exploiting different input field, for example a search bar, comments or form fields. If the user supplied input is returned and embedded somewhere on the page the inserted code can be executed Due to the wide spectrum of functions JavaScript can perform, this inserted code can have major impacts, from stealing the credentials of users, who think they're on a trusted website, to automatically installing malware. An example is the infamous Samy Worm, the first mayor XSS Worm, which, in 2005, shut down the popular social network Myspace and infected millions of users profiles in a matter of hours [40]. Note that XSS attacks are also possible with other scripting languages such as PHP. This thesis will focus on JavaScript XSS. However, the problems and mitigation tactics are similar.

To understand XSS, it is important to know what different kinds of XSS exist and how to differentiate them. There are three main types of XSS attack:

1.1 Persistent Cross-site Scripting

Persistent, or stored, Cross-site Scripting is a very devastating attack, in which data from user input fields is somehow stored on a website's server and shown to other users. Such data comes, for example, from a comment section of a website. If those inputs aren't properly sanitized before being sent back to other user's browsers they might be vulnerable because an attacker might inject a script into those fields. This script would then be executed on the computer of every user who would normally see the reflected input data. One such example is the aforementioned Samy Worm which lived in blog posts of users and executed a script that published a new copy of this post on the blog of every user who saw it [36].

1.2 Reflected Cross-site Scripting

Reflected, or non-persistent, Cross-site Scripting is the most common type of XSS. Unlike in persistent XSS a user on the website cannot get directly infected, rather the attacker has to send a URL which contains the malicious script in its payload. Only users who click on this link will see the page, now with the added code. Such attacks are usually used for phishing, cookie stealing and Account Hijacking, since the user, as well as the browser think they are communicating with a trusted website. An attacker with a clever script can dupe the victim into sending them confidential information, by posing as the normal login page of the website but sending the inserted data back to them self instead [36].

1.3 DOM-based Cross-site Scripting

DOM-based Cross-site Scripting is possible if a user-controlled source is brought to a sink, which evaluates and executes the input directly on the client-side, through JavaScript's eval() function for example. Most commonly the source for such attacks is, similar to the reflected XSS, the URL, which an attacker can tweak in such a way that dangerous scripts are executed on the victim's computer. The main difference between DOM-based and reflected XSS is that the vulnerability lies in code that is directly executed in a victim's browser, rather than embedded into the HTML-code of a website [26].

1.4 Impact of Cross-site Scripting

The following explanations are, if not stated otherwise, based on Chapters 1 & 3 of the book CROSS-SITE SCRIPTING by B.B. Gupta and Pooja Chaudhary [41].

To understand how dangerous an XSS vulnerability really is, it is important to know what possibilities an attacker has to exploit a detected weakness in a website. The end goal of each of these attacks is either gaining control over a victim's system by acquiring their confidential information or hijacking the victim's account. Here are five of the most hazardous security issues that arise from XSS vulnerabilities, either because they are very prevalent, since they are easy to exploit, or because their effects are severe.

1.4.1 Phishing

Phishing is a common tactic used to get a user's confidential information. Usually an attacker sets up an imitation of a trusted website where a user has to fill out some form, e.g. their login data. The input is then sent to the attacker instead of, as the user thinks, the website. With a reflected XSS vulnerability a URL can be created, sending the user to the page with the nefarious form. Since the user is still on the same page, they are more likely to be trusting and might give up his data more easily [41].

1.4.2 Cookie Stealing

Cookie Stealing is the practice of acquiring the cookies a browser has saved of a certain website and sending them to the attacker. To get these cookies through XSS an attacker only has to send the cookies of the website to them self instead of the usual server. This can be especially dangerous for sites that use cookies for authentication. A skilled attacker can use the stolen cookies to impersonate the victim [41].

1.4.3 Browser Exploitation

Browser Exploitation is another harmful intrusion into a system's autonomy. When an attacker successfully exploits a browser, they can control certain parts of the browser and might install malware onto it or redirect certain queries to harmful websites. If an XSS vulnerability is found, it's quite easy to automatically trigger the download and install malevolent software via JavaScript [45].

1.4.4 Denial of Service Attack

The goal of Denial of Service (DoS) attacks is to send so many requests to a website's server that it cannot keep up and the site can't function normally anymore. This can cause great damage, as users cannot use the site's services anymore. In XSS such an attack can be executed through a persistent XSS vulnerability, where the attacker puts a certain request for the server, which will be executed by each user visiting the site. With an expensive (in terms of computing effort) enough request and enough users visiting the page, the site could crash [43].

1.4.5 Remote Control on System

Similar to browser exploitation but much more devastating is if an attacker can somehow infect not only the browser but even the victim's whole system. For this to happen the attacker would have to construct a page which would download and install a remote-control software and afterwards execute it. Usually there are checks and safety mechanisms in a browser, as well as a computers operating system blocking such attacks' making this kind of attack very unlikely to succeed, but with ever advancing malware technology it's still a possibility [41].

1.5 Frequency of XSS occurrences

Since XSS is an older type of weakness one would think most websites would be protected against it. However, the National Vulnerability Database (NVD) that provides a list of Common Vulnerabilities and Exposures (CVEs) , provided by the National Institute of Standards and Technology (NIST) of the United States, claims that 13.49% of all vulnerabilities found in 2019 were XSS based [51].

The OWASP Foundation, which tries to improve security throughout the internet actually puts Cross-site Scripting as the 7th highest in their Top Ten List of Application Security Risks [52].

So many occurrences, combined with the devastating effects XSS can have on a website and its users, it is indispensable to accurately identify those weaknesses.

2
Finding XSS Vulnerabilities manually

To find some websites against which the automated web crawlers could be tested, we first had to identify pages where XSS vulnerabilities were present. Our search was focused on reflected XSS weaknesses, as they are quick to check and would usually only leave a small trace in a server's Log-file somewhere. Checking for stored XSS would require the modification of the database or other stored files of each website in some form, which might be seen as a serious attack attempt where the webmaster of the site most likely wouldn't be very forgiving.

2.1 List of all Tested Websites

The websites we tested were of Swiss origin. First, we tested some rather old websites, which were known to not have been updated for some time. Afterwards, we checked some of the bigger retail websites and their subsidiaries. The tests included the following ten websites:

- `https://etoa.ch/`

  Etoa, short for Escape to Andromeda, is an older multiplayer online game where a user has to create an account to play.

- `https://playit.ch/`

  Playit is a game hosting platform where smaller games can be played for free.

- `https://www.staemme.ch`

  Stämme is a multiplayer online game, similar to Etoa. It is still today one of the most popular swiss online games.

- `https://www.coop.ch/de/`

  Coop is one of the biggest retailers in Switzerland.

- `https://www.orellfuessli.ch/`

  Orellfüssli is the biggest book distributor in Switzerland.

- `https://www.migros.ch/`

  Migros is the biggest retailer in Switzerland.

- `https://www.interdiscount.ch/de`

  Interdiscount is a subsidiary of Coop, focusing on the distribution of consumer electronics

- `https://www.sportxx.ch/de`

  A subsidiary of Migros, which sells mostly sporting goods.

- `https://www.manor.ch/`

  Manor is a chain of general purpose stores.

- `https://www.ricardo.ch/`

  Ricardo is a bidding platform where users can buy and sell items.

2.2 Mechanism used to find Reflected XSS

The first thing on each website we checked was the presence of a full-text search field. Afterwards, we looked for input fields which would generate a page-modifying response like login fields or specific search fields. When we found such a field, the first thing tested was, if the users' input would be embedded in the HTML of the newly generated page. If this was the case, we used a few of the designated attack scripts to check if any flaw exists.

The scripts we used are all found in some form on OWASPs XSS Cheat Sheet. They all produce, if successful, a popup message containing the Message 'Rafi'. To accurately find vulnerabilities it is important to pause all forms of Ad Blockers in the browser, as they tend to suppress unexpected pop-up windows.

The basic code used is the following:

<script>alert('Rafi')</script>

The 'script' tag indicates to a browser that the text inside should be interpreted as client-side JavaScript code. If this code is reflected back into the page's HTML-code in an unmodified manner by the website, the browser will execute the code between the two tags.

With this script we found two separate vulnerable pages. The first is on playit.ch with an input field called 'suchbegriff' in the search bar and the second is Etoa's help page (help.etoa.ch) where the flaw appears within a input field called 'faq'.

This code of course will only be reflected by the most out of date or careless of websites. Usually there is some form of XSS prevention in place. The easiest of these defense techniques is straightforward: replacing special characters like angle brackets in the query before reflecting it back to the page.

The second very rudimentary prevention form is putting the query in a so called safe space, i.e. a predefined context where when displaying text, the code won't be parsed as HTML but will only be seen as an instance of plain text.

The OWASP Cheat Sheet [49] has several suggestions on how to circumvent those two methods. For the first one there is an easy way by using the encoding of the page. If the code is sent with the encoding of the special characters instead of the character itself it might not be detected and when sent back to the browser would still be interpreted as the special character. For example, in a HTML-encoded text the strings '&#60' and '&#62' are the ISO-8859-1 encoding of the angle brackets '<' and '>' respectively [27]. For a website which might filter out those two characters the script could be modified as follows:

&#60script&#62 alert('Rafi') &#60/script&#62

This would allow the code to get through a filter undetected, if this filter doesn't check for any characters like '&' or '#'. With this code an XSS vulnerability was found on the main page of Coop in the search bar within an input field called 'text'.

Other suggested encodings for '<' like 0x3c (UTF-8), &lt (XML) were tested but those yielded no additional vulnerable webpages.

Other scripts on the OWASP Cheat Sheet [49] try to escape the safe input fields as explained in the second prevention form. They usually try to achieve this goal by putting string escape sequences ahead of the script. Those sequences usually contain one or more of the characters ', ", > or -. Other variants try to close the safe HTML tags as can be seen in the code:

/*--></title></style></textarea></script><script>alert('Rafi')</script>

It tries to close one or more tags and then start its own script. A weakness was found with this method in an online bookstore's search field (orellfuessli.ch). This was patched before it could be used for any tests.

3
Web Application Security Scanners

To find vulnerabilities more easily, different kinds of Web Application Security Scanners have been introduced to find those attack vectors automatically. For this thesis we analyzed several of these Scanners. The list of those scanners is comprised of OWASPs list of vulnerability scanners (`https://owasp.org/www-community/Vulnerability_Scanning_Tools`) where all scanners that were listed as open-source were taken. The list is complemented with those on a blogpost from 2020 by Pavitra Shandkdhar, a security researcher at the InfoSec Institute specialized in web penetration testing [56]

3.1 Introduction to Automated Scanners

This introduction is based on the information on the blog "Breaking Down Web Application Scanning: Know-How and Know-Why" [59] and information gathered through reading the documentation of the different scanners tested. A Web Application Security Scanner is a program that performs dynamic Black Box Testing for different types of vulnerabilities on a specified site. Those tests are programming language independent since all websites, in the end, are displayed in a similar manner for browsers to interpret. A Web Application Security Scanner usually analyses web sites in three steps, which can also be done concurrently.

3.1.1 Discovery

The first step is called the discovery phase. In this step, the part of the scanner called a spider tries to find as many URL paths as possible within the defined scope. The scope can be either a single page, a folder, the subdomain or the whole domain.

The spider needs a URL as the starting point for a queue of all pages to check. It then recursively analyzes the response body it recieves when requesting the next URL in the queue for new hyper references such as links. If these are within the scope and have not already been discovered it adds them to the queue. It does this for as long as it has URLs in the queue.

Sometimes a scanner also has a dictionary of common URL paths or paths it learned from previous crawls. If present the scanner will check for paths in its dictionary and add found pages to the queue, before running the normal crawl.

3.1.2 Fuzzing

The second phase is called fuzzing. On each page the Fuzzer tries to inject semi-random or deliberately sinister code into the request to find a flaw. For this the scanner might have a list of different payloads it will try to inject into found input fields.

3.1.3 Analyzing

After the Fuzzer transmits a malicious request the scanner analyzes the response to look for the flaw which it intended to find. If a vulnerability is found it is reported to the user.

For example to find a XSS vulnerability a scanner might inject a payload such as >'>>""><tag> during the fuzzing phase. Then it checks the response body for the tags <tag> and </tag>. If they both exist it indicates, that <tag> was parsed as a HTML tag and therefore it has to be closed before any outer tag is closed.

3.1.4 Authenticated Scan

In some applications a user can also define login data with which the Web Crawler can perform those aforementioned steps as an authenticated user as well [59].

3.2 Criterion-Catalogue for judging the Scanners

To analyze the different scanners we take several criteria into consideration. They aren't all weighted equally in the conclusion, since some are more important than others. For each criterion a short explanation is given and a judgment of its importance. The list is sorted by the chronological order the criteria would be tested on a scanner.

3.2.1 Installation and Dependencies

The first criterion has to do with the installation of the application. If the installation process is too complicated, most users won't bother trying the program. Because well over 60% of computers run on the Windows 10 operating system [60] it was tried, whenever possible, to install the applications on this system. If not directly possible a Windows Subsystem for Linux (WSL) was used.

We judge this criteria by looking at how many steps a user has to manually perform to install the scanner and how much time the whole process needs.

3.2.2 Complexity of the Interface and usage

Another very important aspect in terms of popularity is a comprehensible interface. The interface might be a command line or a graphical user interface. If reading several pages of documentation is required before being able to do even the most basic tasks this will be off-putting for users. Not being able to check the progress or the state of the application it might also be a deterrent for using it.

For GUI scanner we check how many steps are necessary before a crawl with our predefined configurations can be performed. For scanners with a command line interface we compare how many arguments a user has to remember. Secondly we look at the documentation provided and see if it is detailed, well-structured and understandable for a new user.

3.2.3 Soundness of the Application

One of the most important criteria of any application is its soundness. If a regular user encounters a lot of bugs and crashes or if it uses up too much of a systems resources such as CPU or RAM, users tend to become frustrated very quickly.

We check how regularly the scanner freezes or crashes. And for all 10 websites, how often a scan had to be restarted because the scanner had a problem during the scan.

3.2.4 Duration of the Analysis

The time difference between the start of the request to analyze and the finalization of the end result is also an important part of an analysis. It might become less relevant the more responsive the scanner is about what it already found and what it still has to do, but even then a user's patience is limited.

We try to measure the time elapsed for each scan.

3.2.5 Complexity of interpreting the Results

Even the most reliable scanner won't be used often if the results it produces cannot be interpreted by the person using it. Therefore it is important that the findings are easily interpreted and as detailed as needed for an accessible user analysis.

To check how detailed the results are we look at the following aspects:

- a sensible grouping of the vulnerabilities
- the exact page where the vulnerability is found
- the request sent to find the vulnerability
- the proof in the response body where the vulnerability is found

3.2.6 Completeness of the Findings

The most important point of course for any scanner is, that the findings are as complete as possible. If a scanner does not produce reliable results and overlooks crucial vulnerabilities it is not a good scanner.

We check if the scanner finds all XSS vulnerabilities we discovered through either manually looking for them or when another scanner found them.

3.3 Setup for the Tests

The first 9 tests were run on a Lenovo Thinkpad with a x64 Intel Core i7-7600 CPU: 2.80 GHZ with 20 GB RAM on a Windows Pro 10 Version 1909 64-bit. When specified the Windows Subsystem for Linux with Ubuntu Version 16.04 was used. For the tests on GoLismero, Nikto/Wikto and Ride a Windows 10 virtual machine with 64GB RAM and 64 virtual cores was used.

3.4 Testing procedure

We attempted to download and install each Web Application Security Scanner of the list. For the installation process the instructions given by the download page were followed. If the page mentions that the scanner should be installed on a Linux system, we attempted the process on the WSL. After a successful installation we ran the application and configured it to only check for XSS vulnerabilities if possible. The next step was to scan the ten websites where the manual checks had been done. Whenever possible we limited the depth of a scan 8 nodes starting from the main pages of each website. If any additional vulnerabilities were found on one of the pages by a scanner we checked manually verify that the vulnerability was not a false positive, meaning a flaw is showing up in the results which is already mitigated in some way.

4 Scanner Tests

4.1 Vega

4.1.1 Description

Vega by Subgraph is a Java-based application that should therefore be very easy to install and run since it runs on the JVM and has a high transportability between operating systems. Vega can, according to their website, run in two modes, either as an automated scanner, which is the mode it was tested in, or as an intercepting page-factory with which more specific and lifelike tests can be performed, because in this mode Vega can intercept and edit requests and responses between the server and browser during a users normal usage of the website [31].

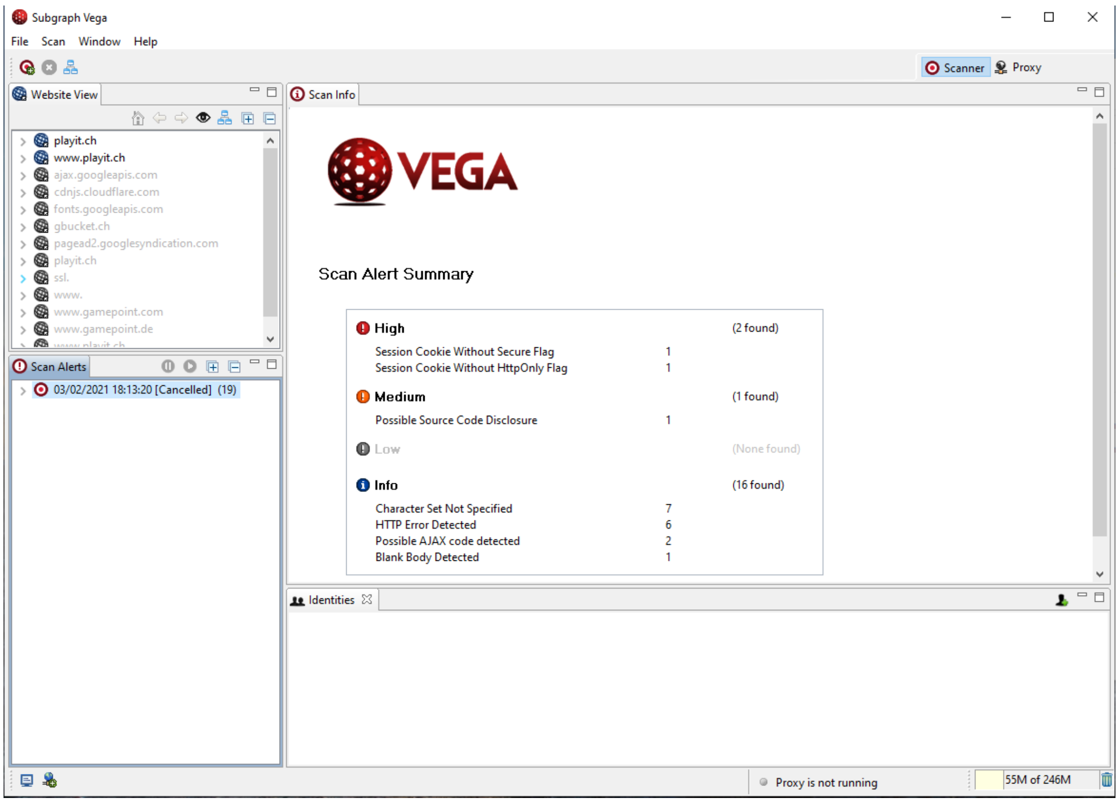
4.1.2 Installation and Dependencies

As a Java application its only dependency is a JRE which was already installed on our computer. For the download (`https://subgraph.com/vega/download`) there are different options for each operating system it runs on. We installed the tool with the wizard from the EXE file. The download page does have a quick paragraph about troubleshooting but we didn't encounter any problems installing and running Vega. The Version we used was: 1.0 Build id: devel-130.

4.1.3 Complexity of the Interface and usage

The user interface is quite simple. On the main window it has the website view showing all the websites that have been scanned before in alphabetical order. Right underneath is a panel that shows all scans including how many vulnerabilities were found per scan. On the right hand side is a big scan info panel which shows details of a selected scan, and beneath that is the authentication information used by this scan. The application has many icon buttons which might not be that intuitive, but thanks to a detailed documentation on the subgraph website it is no problem getting a grip on how to use Vega.

To start a scan, the application has a helpful scan wizard which guides the user through modifying the scan settings. It's possible to choose several websites per scan. The user can choose which attack vectors to test for. To test for XSS only the user can deselect all attack modules except XSS injection, which will make the scan a lot faster. A user could also select authentication data or cookies for the website which should be used if possible, but wasn't used for this test.



playit.ch
www.playit.ch
ajax.googleapis.com
cdnjs.cloudflare.com
fonts.googleapis.com
gbucket.ch
pagead2.googlesyndication.com
playit.ch
ssl.
www.
www.gamepoint.com
www.gamepoint.de

03/02/2021 18:13:20 [Cancelled] (19)

Scan Alert Summary

High		(2 found)
Session Cookie Without Secure Flag	1	
Session Cookie Without HttpOnly Flag	1	
Medium		(1 found)
Possible Source Code Disclosure	1	
Low		(None found)
Info		(16 found)
Character Set Not Specified	7	
HTTP Error Detected	6	
Possible AJAX code detected	2	
Blank Body Detected	1	

Figure 4.1: Vega's interface

4.1.4 Soundness of the Application

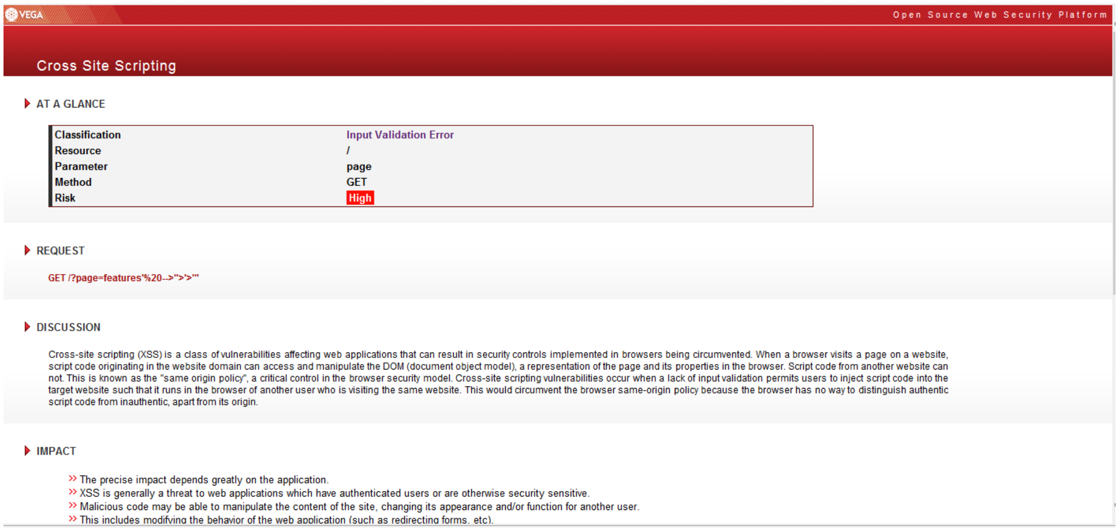
Unfortunately Vega has one major flaw in its UX: it is pretty slow and sometimes non-responsive. Quite often a simple click, for example by choosing a target scope of websites instead of just one to scan, it loads all possible websites which is really time consuming even when the number of sites stored isn't that high. There are more then a few buttons which can be clicked that will freeze the application for several seconds or even minutes. It also still has some minor bugs such as a lock file exception popping up when starting Vega.

4.1.5 Duration of the Analysis

If the analysis scope is limited to XSS only it is reasonably fast. As it doesn't have a built-in clock which would measure the exact time needed and didn't specify when the scan ended, run times had to be approximated. The application doesn't save the number of requests made per website, therefore it's hard to find out how it ranks in this aspect against others.

4.1.6 Complexity of interpreting the Results

The results are shown as a dropdown list under each website scanned. For each website is listed how many vulnerabilities have been found. These are then categorized by threat level in high, medium and low. Here Cross-Site Scripting is always in the high category. For each found vulnerability, it shows the path to the page it was found on, and when such a path is clicked on, it reveals the details of which attack vector and HTTP request were used, i.e. with which payload the attack succeeded. Underneath is a brief description and possible impacts which such a vulnerability in general can have on a website. This is not specific information but a general text for each vulnerability type.



Cross Site Scripting

AT A GLANCE

Classification	Input Validation Error
Resource	/
Parameter	page
Method	GET
Risk	High

REQUEST

```
GET /?page=features'%20-->">'>'"
```

DISCUSSION

Cross-site scripting (XSS) is a class of vulnerabilities affecting web applications that can result in security controls implemented in browsers being circumvented. When a browser visits a page on a website, script code originating in the website domain can access and manipulate the DOM (document object model), a representation of the page and its properties in the browser. Script code from another website can not. This is known as the "same origin policy", a critical control in the browser security model. Cross-site scripting vulnerabilities occur when a lack of input validation permits users to inject script code into the target website such that it runs in the browser of another user who is visiting the same website. This would circumvent the browser same-origin policy because the browser has no way to distinguish authentic script code from inauthentic, apart from its origin.

IMPACT

- The precise impact depends greatly on the application.
- XSS is generally a threat to web applications which have authenticated users or are otherwise security sensitive.
- Malicious code may be able to manipulate the content of the site, changing its appearance and/or function for another user.
- This includes modifying the behavior of the web application (such as redirecting forms, etc).

Figure 4.2: Results of a scan in the Vega application

4.1.7 Completeness of the Findings

Of the three websites with known flaws it only found the two plain text vulnerabilities (etoa.ch and playit.ch). It did not find the more complex one on the Coop website. It did find a second XSS vulnerability on etoa.ch on a GET request. The result shows the following request: GET /?page=features'%20-->">'>'", First is the CRUD keyword of the method used, here GET. Then as it is a GET request it shows the input field page.

4.1.8 Result

All in all the Vega scanner has a lot of potential. Its UI is clean and easy to use. All functionalities are together in one application and the documentation is quite detailed.Its response time for very basic user operations and its failure to find all basic XSS vulnerabilities however are troubling.

4.2 Skipfish

4.2.1 Description

Skipfish by Google is a scanner which advertises itself with three key advantages. First it is written in C with conserving CPU power and memory footprint in mind. Therefore it is described to have a very high performance. As a second plus skipfish claims to be very reliable and self-learning. Lastly due to a modifiable dictionary and other subtle design features it should have fast and well-designed security checks [64].

4.2.2 Installation and Dependencies

Even though it first seemed as if skipfish could be run on the Windows native CLI and some dependencies were already installed on our computer, we had to move to the WSL as the system didn't recognize the skipfish commands used. For WSL however it was simple to install.

4.2.3 Complexity of the Interface and usage

As skipfish is controlled over the command line it is not as easy for a new user to grasp all its functionalities. Thankfully the README file and Kali-Tools keyword outline (`https://tools.kali.org/web-applications/skipfish`) did explain a lot of the different parameters which can be set for running the scan. Similar to Vega we can set authentication data and other HTTP cookies. There are other more subtle functions like limiting the paths skipfish should crawl, or predefining trusted third-party domains to limit false positives. It can also check either only one URL and its subsequent paths or it can be fed a text file with several URLs it then scans in sequence.

A more confusing part of the scan is the dictionary which determines which kind of URL paths are tested in a brute force manner. This means after skipfish does an orderly web crawl it checks out all URL paths which are in its dictionary. The dictionary has a list of common paths, file names and extensions. The scanner also has the ability to modify this dictionary with all the newly learned path extensions during the web crawl. There are three basic dictionaries: minimal, medium and complete coverage. The differences however are rather marginal. They all have between 2172 and 2221 keywords. Here it depends a lot more which brute-force method is used. It can be either no brute-force, only lightweight, which means the scanner will only test limited extensions, and normal dictionary brute force in which it will try most file name/extension pair permutations in the given dictionary. The minimal dictionary already provided by skipfish was used in these tests. As without any dictionary it didn't find any XSS issues. The depth of the crawl was restricted to further speed up the scan, otherwise the scan on playit.ch at least would go on for well over 24 hours.

4.2.4 Soundness of the Application

Since skipfish is implemented in C it runs very smoothly and uses only minimal CPU power and RAM. Larger websites with many pages produced by a page-factory that are essentially the same, like the game pages provided by playit.ch, dupe the brute force algorithm to test all permutations on each of these pages, which takes an enormous amount of time without gaining any new insight.

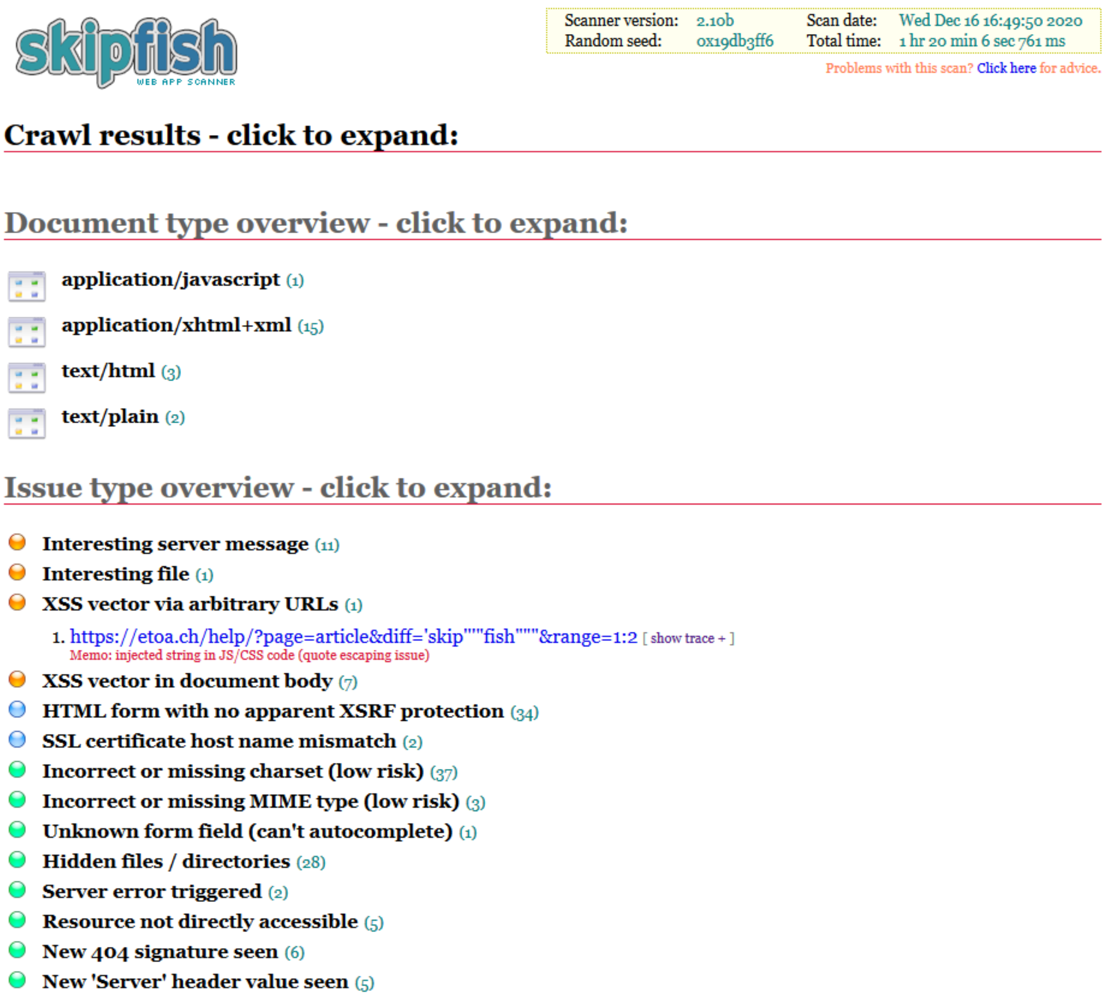
Some websites couldn't be scanned, no matter what we did. All tests started normally and the scanner could retrieve some URLs but then after a few seconds it declared itself to be finished. This could be due to the websites having a security element which prevents too many requests per second from a single source to prevent Denial of Service attacks.

4.2.5 Duration of the Analysis

The request rate is pretty fast but since it's not possible to specify the attack vectors for which the site should be tested, all scans take quite a long time, but due to the verbose progress screen it can be checked immediately not only if the application is still working, but also how many requests have already been made, how many it makes per second, how many nodes still haven't been crawled and if vulnerabilities have been found. On a second page it even shows some of the requests that have been made in the last second. The longer a scan was running the fewer requests per second were made.

4.2.6 Complexity of interpreting the Results

For each scan the application writes its findings into a HTML file in the specified directory. This file shows the crawled websites and which underlying issues were found. It then shows some documents it could access and might be interesting like error-logs, XML files and so on. Afterwards there is an overview over all vulnerabilities ordered by their severity (High impact, medium, low, warning and info). XSS vulnerabilities are categorized as medium impact, it first shows the whole URL with which the attack was successful. With the 'show trace' link the request and the response for each of the vulnerabilities can be examined. It presumes a bit more knowledge than Vega but is still very easy to read and quite understandable.



Scanner version: 2.10b Scan date: Wed Dec 16 16:49:50 2020
Random seed: 0x19db3ff6 Total time: 1 hr 20 min 6 sec 761 ms
Problems with this scan? Click here for advice.

Crawl results - click to expand:

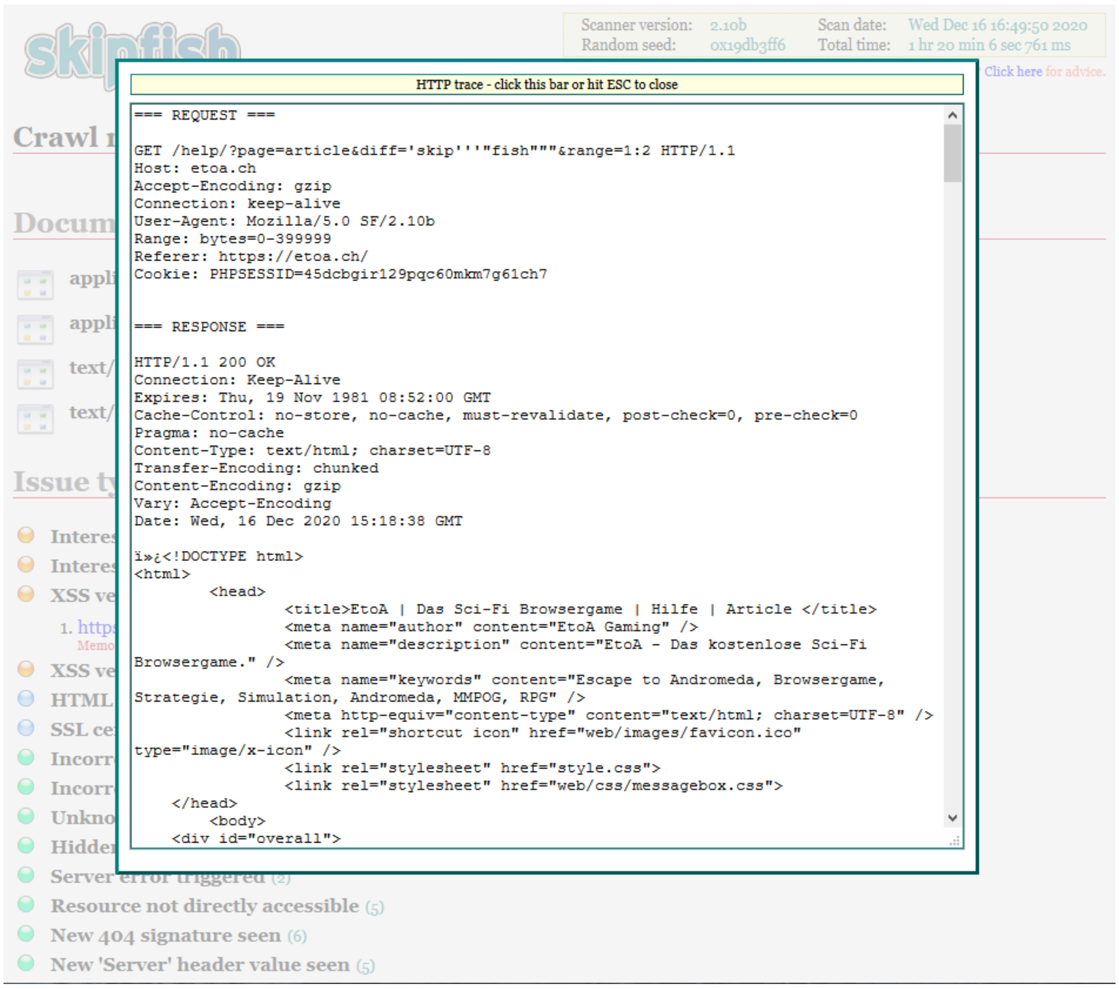
Document type overview - click to expand:

- application/javascript (1)
- application/xhtml+xml (15)
- text/html (3)
- text/plain (2)

Issue type overview - click to expand:

- Interesting server message (11)
- Interesting file (1)
- XSS vector via arbitrary URLs (1)
  1. https://etoa.ch/help/?page=article&diff='skip''''fish''''&range=1:2 [show trace +]
     Memo: injected string in JS/CSS code (quote escaping issue)
- XSS vector in document body (7)
- HTML form with no apparent XSRF protection (34)
- SSL certificate host name mismatch (2)
- Incorrect or missing charset (low risk) (37)
- Incorrect or missing MIME type (low risk) (3)
- Unknown form field (can't autocomplete) (1)
- Hidden files / directories (28)
- Server error triggered (2)
- Resource not directly accessible (5)
- New 404 signature seen (6)
- New 'Server' header value seen (5)

Figure 4.3: All results of a skipfish scan


Scanner version: 2.10b
Random seed: 0x19db3ff6
Scan date: Wed Dec 16 16:49:50 2020
Total time: 1 hr 20 min 6 sec 761 ms
Click here for advice.
Crawl r
Docum
appli
appli
text/
text/
Issue ty
Interes
Interes
XSS ve
XSS ve
HTML
SSL ce
Incorr
Incorr
Unkno
Hidde
Server error triggered (2)
Resource not directly accessible (5)
New 404 signature seen (6)
New 'Server' header value seen (5)
HTTP trace - click this bar or hit ESC to close

```
=== REQUEST ===

GET /help/?page=article&diff='skip'''"fish"""&range=1:2 HTTP/1.1
Host: etoa.ch
Accept-Encoding: gzip
Connection: keep-alive
User-Agent: Mozilla/5.0 SF/2.10b
Range: bytes=0-399999
Referer: https://etoa.ch/
Cookie: PHPSESSID=45dcbgir129pqc60mkm7g61ch7


=== RESPONSE ===

HTTP/1.1 200 OK
Connection: Keep-Alive
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Content-Encoding: gzip
Vary: Accept-Encoding
Date: Wed, 16 Dec 2020 15:18:38 GMT

ï»¿<!DOCTYPE html>
<html>
        <head>
                <title>EtoA | Das Sci-Fi Browsergame | Hilfe | Article </title>
                <meta name="author" content="EtoA Gaming" />
                <meta name="description" content="EtoA - Das kostenlose Sci-Fi
Browsergame." />
                <meta name="keywords" content="Escape to Andromeda, Browsergame,
Strategie, Simulation, Andromeda, MMPOG, RPG" />
                <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
                <link rel="shortcut icon" href="web/images/favicon.ico"
type="image/x-icon" />
                <link rel="stylesheet" href="style.css">
                <link rel="stylesheet" href="web/css/messagebox.css">
    </head>
        <body>
    <div id="overall">
```

Figure 4.4: The results of a vulnerability expanded to see the details

4.2.7 Completeness of the Findings

While skipfish checks for many vulnerabilities and finds a lot of them, the only ones that were examined are the XSS vulnerabilities it found. It did find attack vectors on help.etoa.ch. It also found the same vulnerability on the same search field on six other pages. It did not find the shortcoming on etoa.ch itself. And while it found a lot of apparent problems on coop.ch and playit.ch, it did not find any XSS vulnerabilities. For playit.ch this might be due to the cancellation of the scan after a day.

4.2.8 Result

Even though skipfish might be more complicated to use than Vega it is definitely less frustrating since it runs smoother but still wouldn't run without flaws as it seemed to stop scanning some of the websites prematurely. It also seems to not be designed to find a lot of XSS issues. For a user who wants to check his website thoroughly for all kinds of attack forms it is a good start. If someone only wants to check for XSS vulnerabilities it might be a little too broad.

4.3 Grabber

4.3.1 Description

Grabber is a Python based web application crawler. It is written by Romain Gaucher, a Software Security expert working for the Software Integrity Group in Paris. The program is advertised to be highly modular, meaning it is possible to only test for certain kinds of issues. The attack patterns are written in editable XML files, meaning they can be modified and extended to better fit the user's needs [39].

4.3.2 Installation and Dependencies

On the webpage there are three possible ways to download Grabber. The first is to start it as a regular Python program, the second to start as an EXE and the third is with the whole unpacked source code. we downloaded the first option as Python 2.7 was already installed over WSL on our system. Two dependencies had to be installed via Python pip, but then it could run.

4.3.3 Complexity of the Interface and usage

To configure the starting arguments such as the website to crawl, which attacks to test and how deep the crawler should go, there are two possibilities: either the user modifies a config XML file which then will load on runtime, or they can also set them in the command line right next to the start command. For the attack vectors like XSS themselves there are also different XML files which hold all codes that are tried in requests. This makes it very easy to see what attacks are performed, as well as expand those attacks if necessary. The XML config was used initially but then the direct way over command line was also tried. The scan was set to only check for XSS vulnerabilities.

4.3.4 Problems with running the scan

No scan finished successfully even trying several different websites. Some like etoa.ch it couldn't find, saying it wasn't able to retrieve the URL. For others it started the scan but then had an error when trying to retrieve a given URL, saying it contains illegal characters. Of course it cannot be ruled out, that some mistake was made in the setup. We only modified the start-configurations file, which was given in its base form. Neither the depth (tested between 0 and 16 nodes) nor the website seemed to make a difference. We didn't alter the XSS configuration file as the fault which threw the exception couldn't be identified in there. After some time trying to tweak the configurations and websites to crawl in a way that would work we decided to give up and move on since there most likely is a bug in the tool.

As this version couldn't finish a scan successfully we also tried the EXE but it didn't seem to work either.

Grabber didn't write any results in its result folder, but the suspected reason is, that the program first crawls the whole website and then scans each found page individually. Since it already stopped during the crawl process there weren't any results produced yet.

Even though the concept of Grabber is pretty great with different XML files as configurations, it wasn't able to run as intended. Therefore it can't be judged if it does hold up to the other applications on this list. The documentation also leaves a lot to be desired, probably because the application is mostly intended to be used by its programmer, rather than the general public.

4.4 Zed Attack Proxy

4.4.1 Description

The Zed Attack Proxy or ZAP is a web scanner provided by OWASP itself. It boasts itself as the 'world's most widely used web app scanner'. The basic concept is very similar to the Vega proxy scanner where it can intercept requests and responses, and modify them. It is also written in Java, and has a comprehensive user interface. Its main advantage is that the application is modularly built, which means a lot of functionality can be added through add-ons which are developed by the OWASP community. Another main advantage is the extensive documentation page on `http://www.zaproxy.org/docs` which has introduction videos and a very detailed manual.

ZAP has three main ways to test a website's security. The first is the automated scan which was tested. The second way is a manual explorer with which a user can browse a website manually, and ZAP intercepts and analyses all requests and responses, similar to Vega. The third way is a so called 'Heads Up Display (HUD)'. It tries to overlay security information directly in the browser which could be used when a user tests other functionalities as a simultaneous security test [29].

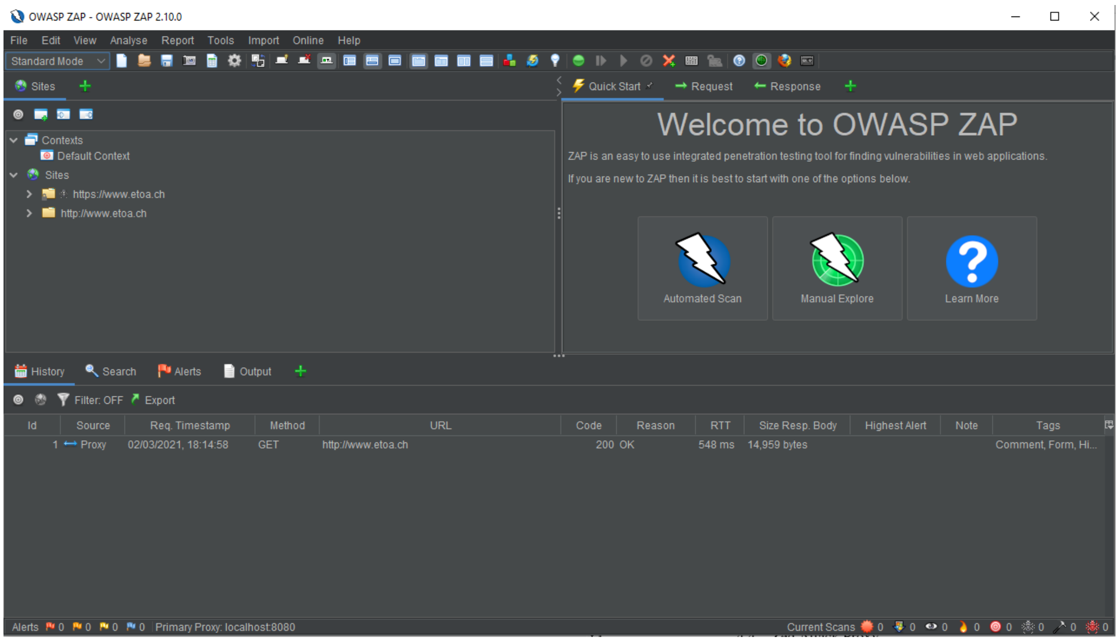
4.4.2 Installation and Dependencies

The only dependency listed is a JRE 8 or higher. Since our system already had JRE 10 installed only the EXE file had to be downloaded and installed via the installation wizard. The version we tested was ZAP version 2.10.0. It started without any problems after the installation.

4.4.3 Complexity of the Interface and usage

The user interface has a similar concept as Vega but is more customizable. It has a menu bar with many different buttons and drop-down menus. There are also three panes. The first on the left hand side is the context pane which shows all the pages crawled. The second, on the right, is the scanning pane which handles the scans. There a user can choose between the automated and manual scan. It's also the pane in which a user can modify requests and responses before they get to the server or browser respectively. The third pane is in the lower half of the window. Through sub-menus it shows and controls different aspects of the scans and their progress. Here options for a new crawl or active scan can be set and all found vulnerabilities are shown via the 'Alerts' menu.

At first the whole interface seems a bit overwhelming with many different icon-buttons and sub-menus. To understand ZAP quite a bit of documentation must be read. Clicking through different dialogs and menus and reading their provided help pages assists so the functionalities are understood. It doesn't help that almost half of the icon-buttons in the main menu-bar are layout related, a matter not really that important for such a prominent placement. The application also has quite a few option dialogs for different scans. Those helped to easily modify the crawl and scan to restrict them so they would search mostly for XSS. ZAP is a bit different than the other applications so far as it does a spider crawl with link discovery first in which the sites would only be checked passively. This means it analyses all requests and responses sent to and from the web application but does not change any of them. Since the passive scan does not include any XSS scanning all tests were shut off that would be made there. This does not impact how fast the crawler scans the web application since the two tasks run on different threads. For each crawl an active scan can be performed which will try to attack the pages listed in the crawl with the attack vectors predefined in a scan policy. To focus only on Cross Site Scripting a new such scan policy was created. For each vector the user can choose a threshold that defines the reporting of potential vulnerabilities from high to low, with low threshold meaning every vulnerability found will be reported which also means more false positive ones, and the 'Strength' over four different levels, low, medium, high and insane where the highest level does the most requests in an attempt to break the system [29]. The XSS attack forms are actually divided in reflected, DOM-based and persistent. In the scan policy used all of them are defined with a low threshold and insane attack to ensure none of the previously discovered targets would get left out. All other attacks were shut off for a better performance.


OWASP ZAP - OWASP ZAP 2.10.0
File Edit View Analyse Report Tools Import Online Help
Standard Mode
Sites
Contexts
Default Context
Sites
https://www.etoa.ch
http://www.etoa.ch
Quick Start
Request
Response
Automated Scan
Manual Explore
Learn More
History Search Alerts Output
Filter: OFF Export
Alerts 0 0 0 0 Primary Proxy: localhost:8080
Current Scans 0 0 0 0 0 0 0 0

Welcome to OWASP ZAP

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications.

If you are new to ZAP then it is best to start with one of the options below.

Id	Source	Req. Timestamp	Method	URL	Code	Reason	RTT	Size Resp. Body	Highest Alert	Note	Tags
1	Proxy	02/03/2021, 18:14:58	GET	http://www.etoa.ch	200	OK	548 ms	14,959 bytes			Comment, Form, Hi...

Figure 4.5: Start screen of the ZAP application

4.4.4 Soundness of the Application

ZAP usually runs smoothly, especially for smaller scopes. It does however have some issues when doing a wide range scan on the websites with more pages. There the crawler did first get slower and then froze completely. At this point the application didn't respond to any further inputs and had to be shut off by force. Since it then couldn't save the session state properly the crawl had to be restarted with a smaller scope.

One additional minor bug that was spotted with DOM based attacks is detailed in the Speed of the Analysis, but as the application itself states, this function is still in Beta and therefore is expected to have some flaws.

4.4.5 Duration of the Analysis

The scans were quite fast. ZAP discovered several thousand URLs per minute during the discovery phase. It did however get significantly slower the longer it ran.

The active scan, where ZAP analyses webpages and tries to inject them with different code snippets, had very different run times for the websites. It only mattered marginally how many requests the scanner made. Far more time was needed for scanning the HTTP-Responses. One major flaw the application has is with DOM-based XSS scanning.

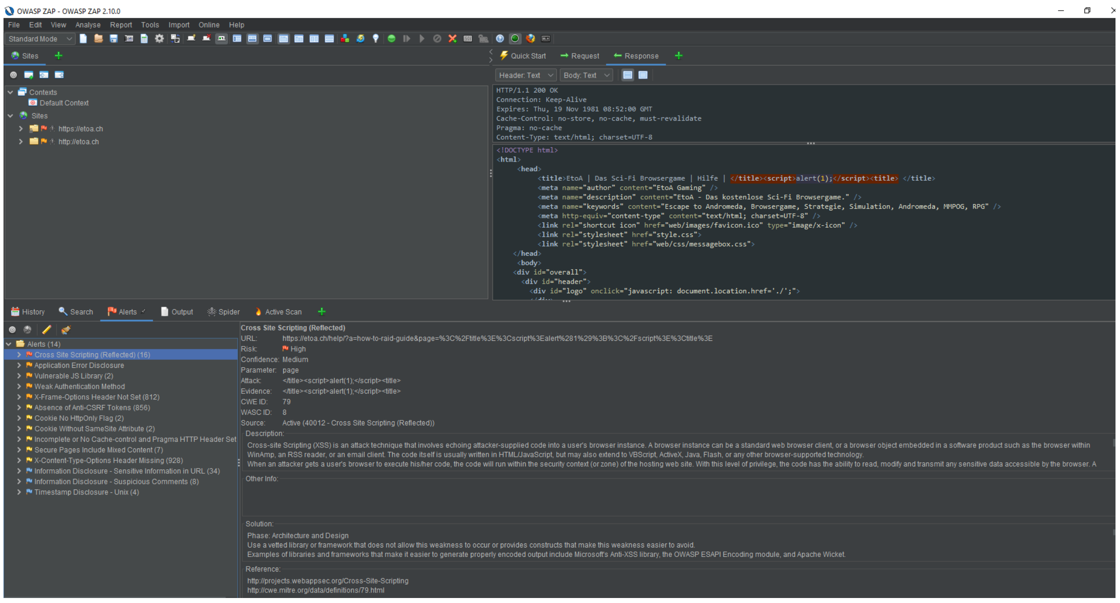
For the first scan (on etoa.ch) the active scan took about 2 minutes for 1513 requests for persistent and reflected XSS attacks. It then somehow took 35 minutes for 12'000 requests in DOM based XSS attacks. This might be because DOM based scanning is still in Beta phase and might still have some flaws. Luckily those requests could be skipped and the scanner ran through the rest without any problems.

After this scan the 'Strength' of DOM based attacks was reduced to 'Low' in an attempt to reduce the amount of requests in this category.

When running the active scan on th Coop website over all discovered URLs, it didn't work because it made so many requests that the program froze again. So the scope was reduced to only check the search sub-path of the website (`coop.ch/de/search`) which then worked. The scanner made 410 requests and took 1 hour and 54 minutes, but again almost all of this time was for the DOM based attacks (1hour and 40 minutes) which didn't even make a request. So after the third scan where it also took the most time by a wide margin we shut off the scan for DOM-based attacks for further scans. Afterward the scans were a lot faster when they ran through without the scanner freezing.

4.4.6 Complexity of interpreting the Results

The results are found in the 'Alerts'-tab. Here for each type of vulnerability a bullet point is listed where, if expanded, all requests which triggered a successful attack are shown. When clicking on one of the requests the request and response can be seen, where the response is split into its header and the body that would be presented. The achieved change in the HTML code which represents the attack is highlighted so a user can easily spot it. In the lower half of the window a more detailed description of the vulnerability including the Evidence found in the response, a brief description of and even possible solutions can be seen.



```
HTTP/1.1 200 OK
Connection: Keep-Alive
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Content-Type: text/html; charset=UTF-8
```

```
<!DOCTYPE html>
<html>
     <head>
          <title>EtoA | Das Sci-Fi Browsergame | Hilfe | </title><script>alert(1);</script><title> </title>
          <meta name="author" content="EtoA Gaming" />
          <meta name="description" content="EtoA - Das kostenlose Sci-Fi Browsergame." />
          <meta name="keywords" content="Escape to Andromeda, Browsergame, Strategie, Simulation, Andromeda, MMPOG, RPG" />
          <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
          <link rel="shortcut icon" href="web/images/favicon.ico" type="image/x-icon" />
          <link rel="stylesheet" href="style.css">
          <link rel="stylesheet" href="web/css/messagebox.css">
    </head>
     <body>
    <div id="overall">
      <div id="header">
        <div id="logo" onclick="javascript: document.location.href='./';">
```

Alerts (14)
- Cross Site Scripting (Reflected) (16)
- Application Error Disclosure
- Vulnerable JS Library (2)
- Weak Authentication Method
- X-Frame-Options Header Not Set (812)
- Absence of Anti-CSRF Tokens (856)
- Cookie No HttpOnly Flag (2)
- Cookie Without SameSite Attribute (2)
- Incomplete or No Cache-control and Pragma HTTP Header Set
- Secure Pages Include Mixed Content (7)
- X-Content-Type-Options Header Missing (928)
- Information Disclosure - Sensitive Information in URL (34)
- Information Disclosure - Suspicious Comments (8)
- Timestamp Disclosure - Unix (4)

Cross Site Scripting (Reflected)

URL:	https://etoa.ch/help/?a=how-to-raid-guide&page=%3C%2Ftitle%3E%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E%3Ctitle%3E
Risk:	High
Confidence:	Medium
Parameter:	page
Attack:	</title><script>alert(1);</script><title>
Evidence:	</title><script>alert(1);</script><title>
CWE ID:	79
WASC ID:	8
Source:	Active (40012 - Cross Site Scripting (Reflected))

Description:
Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.
When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A

Other Info:

Solution:
Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Reference:
http://projects.webappsec.org/Cross-Site-Scripting
http://cwe.mitre.org/data/definitions/79.html

Figure 4.6: Details of a found vulnerability

4.4.7 Completeness of the Findings

For etoa.ch it found 16 reflected XSS vulnerabilities. All on the help page. It apparently didn't find the one on the main page that Vega found even though it did discover this URL in the crawl.

For Coop it actually did find a problem with the search input field, but only with the input ' ";alert(1);" '. If only this string is put into the search field it would not execute the code since the script tags are missing. So a less cautious developer might see it as a false alarm.

In playit.ch it did find the reflected XSS for the search input. It also shows 260 DOM based flaws all stem from the same code in a games window. On the page's HTML a script can be found which uses JavaScript's frowned upon eval() function but seems to accept only certain inputs. It might also have come from the 'suchbegriff' input field of the search as it seems to be used by an innerHTML() function. Since the DOM based alerts don't provide evidence or even the response, it's hard to tell, if this really is what caused the alarm. It also did sound the alarm for orellfuessli.ch where it thought it found a flaw in a registration form. The flaw couldn't be reproduced, so it seems to be a false positive.

4.4.8 Result

With its user interface and detailed documentation a new user can fairly quickly learn most of the basic commands. ZAP did find the more intricate vulnerabilities tested for and is the most reliable. Another convincing feature is the detailed report it generates for each flaw found, highlighting the problem, reporting how confident it is that the problem isn't a false alarm, and even showing solutions.

A positive aspect is the modifiability of the spider and the active scan. Not only can they be run separately, but also the active scan can be restricted to a subgroup of URLs found. The scan policy, which is persistent, makes it easy to only scan for certain kinds of flaws.

The biggest problem with ZAP is its performance. It freezes during bigger scans and crawls, so it's almost impossible to test websites as a whole. Also the shutdown process sometimes takes minutes to end the server and database.

4.5 Wapiti

4.5.1 Description

Wapiti is another general web application security scanner. The Python based application was developed by Nicolas Surribas and a group called ICT Romulus. It can scan for several different kinds of attack forms and runs over the command line. The scan first crawls all URLs, and only after finishing the discovery process tries different attack vectors on them. Through this single threaded scan it preserves resources but the tradeoff is a slower scan in general [28].

4.5.2 Installation and Dependencies

Even though the website states that there should be an EXE file for Windows users on the filesharing platform, we couln't find it anywhere. So we downloaded the regular folder with the source code. The WSL had to be upgraded to Ubuntu 20.04 and an additional Python version (Version 3) had to be downloaded. The whole installation took us a few hours until finally Wapiti could execute. In the end we could successfully launch Wapiti version 3.0.3.

4.5.3 Complexity of the Interface and usage

As a command line application it is again harder to realize all of Wapitis functions. It does have a short README file which gives a very brief overview of all the functionalities. It's short and a user has to have some experience with automated scanners to understand it. The individual arguments with which to launch the scan are listed on a separate page (`https://wapiti.sourceforge.io/wapiti.1.html`). With a bit of trial and error the program with all predefined conditions could run.

4.5.4 Soundness of the Application

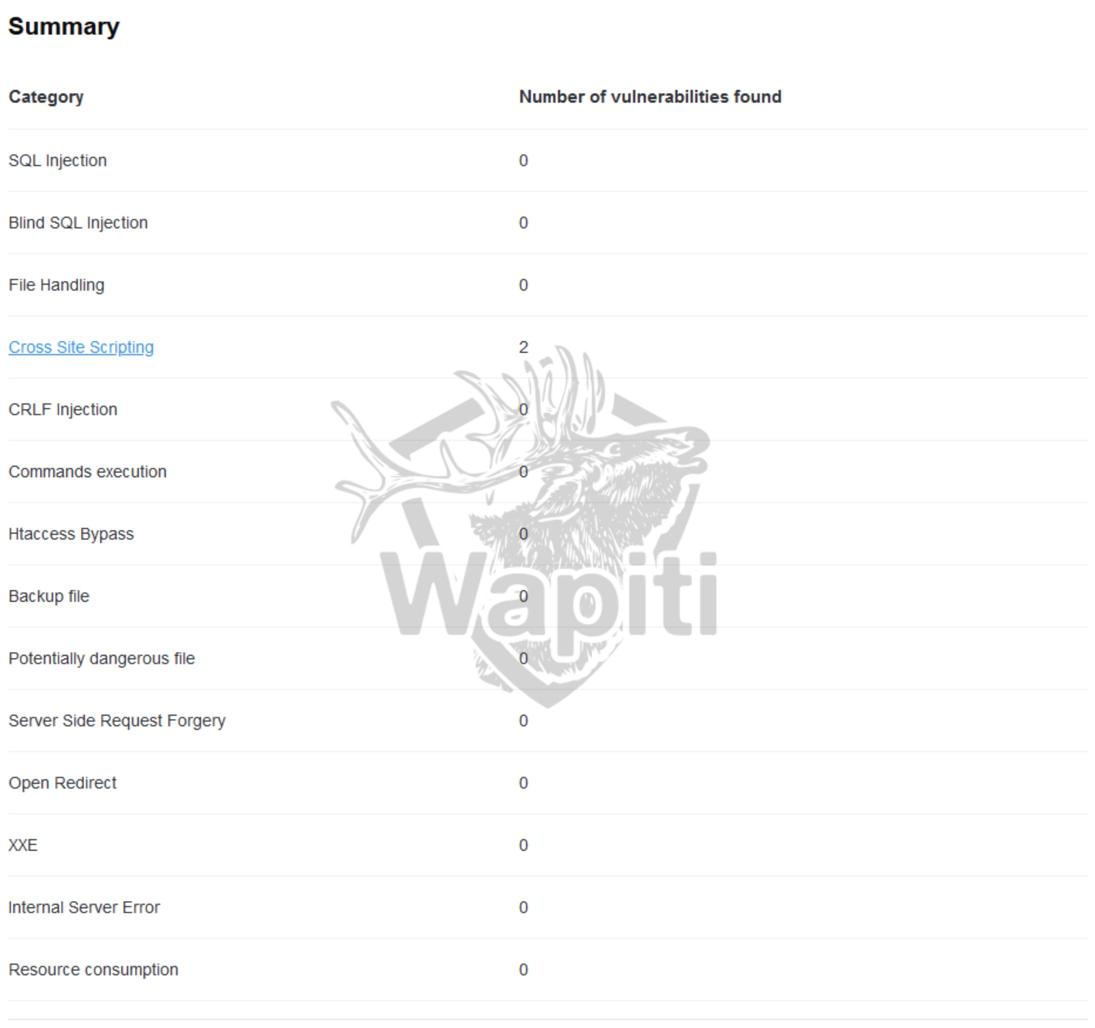
Wapiti did use a bit more of the CPU than other CLI application, but otherwise it ran smoothly. No unexpected behavior was detected at any moment.

4.5.5 Duration of the Analysis

Speed is definitely Wapiti's biggest problem, especially during the crawl. The first scans had to be manually stopped because they took longer then 20 hours. Luckily it is possible to skip further URL exploration and start with the attacks. After scanning the first four pages (etoa.ch, coop.ch, orellfuessli.ch and interdiscount.ch) where three of them had to be manually stopped during discovery, we restricted the depth of the search from 8 to 4 nodes. The other websites also took quite a long time except for the website of Migros, which wouldn't run properly no matter how much we tweaked the starting arguments.

4.5.6 Complexity of interpreting the Results

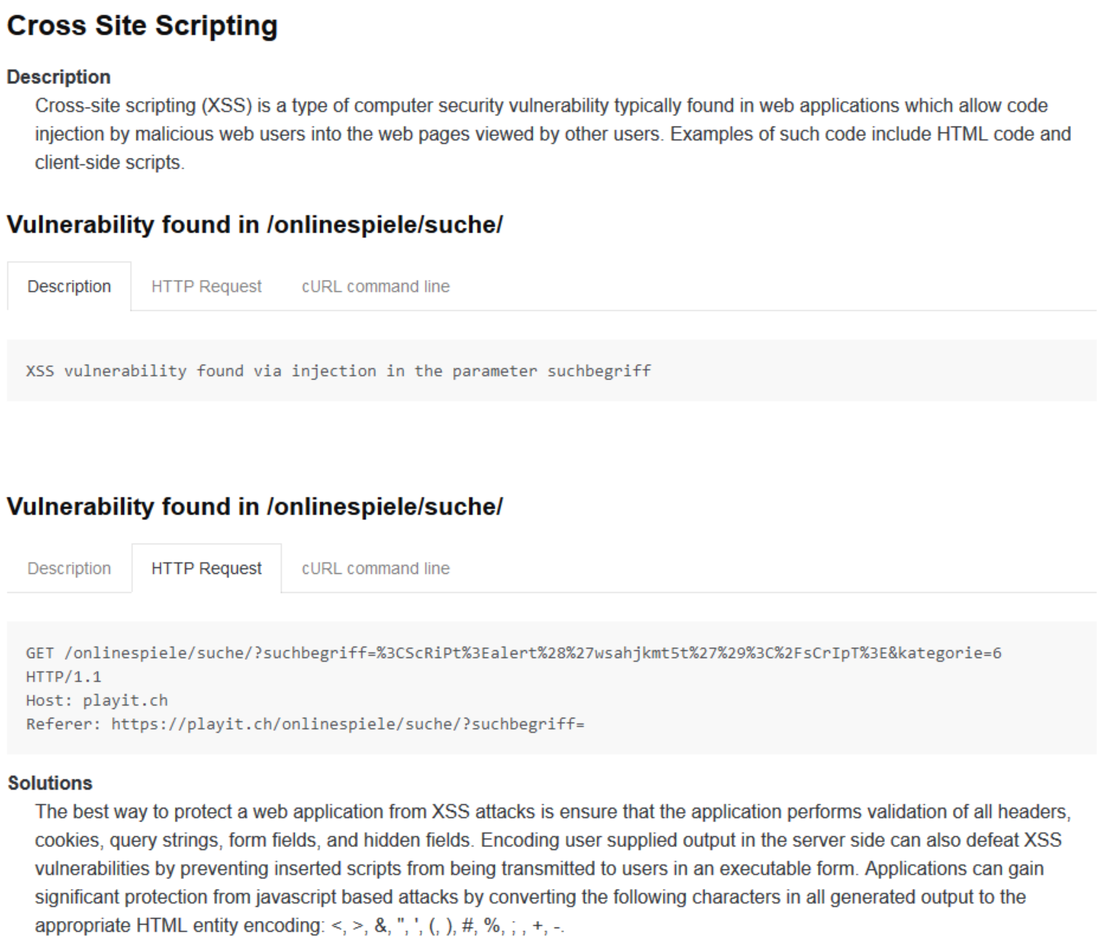
The results can be given in many different formats. The easiest to read would be the HTML. It has a summary over all vulnerabilities and how many it found per category. Underneath are the individual vulnerabilities. Each with a brief description which parameter allowed the attack, the specific HTTP request and the cURL command. At the bottom of all XSS vulnerabilities it lists a general text as a possible solution to fixing these problems through input validation.



Summary

Category	Number of vulnerabilities found
SQL Injection	0
Blind SQL Injection	0
File Handling	0
Cross Site Scripting	2
CRLF Injection	0
Commands execution	0
Htaccess Bypass	0
Backup file	0
Potentially dangerous file	0
Server Side Request Forgery	0
Open Redirect	0
XXE	0
Internal Server Error	0
Resource consumption	0

Figure 4.7: Summary of all vulnerabilities found in a scan



Cross Site Scripting

Description

Cross-site scripting (XSS) is a type of computer security vulnerability typically found in web applications which allow code injection by malicious web users into the web pages viewed by other users. Examples of such code include HTML code and client-side scripts.

Vulnerability found in /onlinespiele/suche/

```
XSS vulnerability found via injection in the parameter suchbegriff
```

Vulnerability found in /onlinespiele/suche/

```
GET /onlinespiele/suche/?suchbegriff=%3CScRiPt%3Ealert%28%27wsahjkmt5t%27%29%3C%2FsCrIpT%3E&kategorie=6
HTTP/1.1
Host: playit.ch
Referer: https://playit.ch/onlinespiele/suche/?suchbegriff=
```

Solutions

The best way to protect a web application from XSS attacks is ensure that the application performs validation of all headers, cookies, query strings, form fields, and hidden fields. Encoding user supplied output in the server side can also defeat XSS vulnerabilities by preventing inserted scripts from being transmitted to users in an executable form. Applications can gain significant protection from javascript based attacks by converting the following characters in all generated output to the appropriate HTML entity encoding: <, >, &, ", ', (,), #, %, ; , +, -.

Figure 4.8: Details of two vulnerabilities showing a description and the request made

4.5.7 Completeness of the Findings

Wapiti did find both problems in etoa.ch. It actually found and reported the flaw on the help site over 600 times.

It didn't find any XSS vulnerabilities on the website of Coop during the scan which was manually stopped, a new scan was started specifically on the 'search' page with a depth of 2 nodes which ran to completion but it didn't sound any alarm then either.

For playit.ch the application found the vulnerability even with the shorter scan.

4.5.8 Result

Wapiti is a nice tool to test small scopes as it might be automatically integrated since it can give results in formats like XML and TXT files. It did not find the more intricate vulnerability which is disappointing. While it might be good for smaller scopes it is certainly not suitable if a user wants to quickly check bigger websites with many input fields as the number of requests per second is limited and the amount of different requests per input field discovered is even with only the XSS module quite big as seen by coop.ch.

4.6 W3af

4.6.1 Description

W3af is an open source Python Web Application Security Software. The main developer is Andres Riancho a security expert from Argentina. It's a general purpose scanner with a CLI as well as a graphical user interface [54].

4.6.2 Problem with the Installation

The program itself can be very easily cloned as a Git repository. It then should, when running either the GUI version or the console version, give a list of unmet dependencies which have to be downloaded. After trying to install those dependencies on Python 3.8.5 a dependency called 'ConfigParser' was needed which Python pip didn't seem to know. After a quick Google search it was clear that this module wasn't supported with Python 3 (`https://github.com/boltgolt/howdy/issues/458`). So we reinstalled Python 2. After running the w3af_gui code a list of unmet dependencies which had to be installed was shown. As Ubuntu 20.04 has dropped all support for Python 2, we had to manually install and download Python pip 2 with these instructions: `https://stackoverflow.com/questions/61981156/unable-to-locate-package-python-pip-ubuntu-20-04`. After a few hours of tweaking and installing additional add-ons for Python 2 all unmet dependencies could be installed. It then still got an error for the dependencies 'GTK' and 'PyGTK'. Even though both were installed over the APT it still got the same error. After realizing those two were only needed for the GUI version we only tested the console version of W3af which also didn't work as it didn't find the module 'netlib'. At this point no feasible solution could be produced as W3af didn't seem to work for the newer Ubuntu system without full support for Python 2.

4.7 Wfuzz

4.7.1 Description

Wfuzz is a framework written in Python. The major difference between Wfuzz and the other applications on this list, is that Wfuzz is merely a fuzzer and therefore does not crawl any pages but only injects payloads into one input and then analyses the results. Wfuzz is developed by Xavi Mendez, an IT Security Consultant [50].

4.7.2 Installation and Dependencies

We installed Wfuzz on WSL as it only had installation options for Ubuntu or Docker. The installation could smoothly be accomplished via Python pip. It automatically downloaded and installed all dependencies.

4.7.3 Complexity of the Interface and usage

As a CLI Wfuzz is pretty straightforward. The application needs a wordlist of the payloads it should check and the exact URL to inject it into. As it didn't have a spider no depth of the scan had to be configured but for a successful fuzzing the exact parameter to analyze had to be defined by putting the keyword FUZZ behind its value in the URL request.

4.7.4 Soundness of the Application

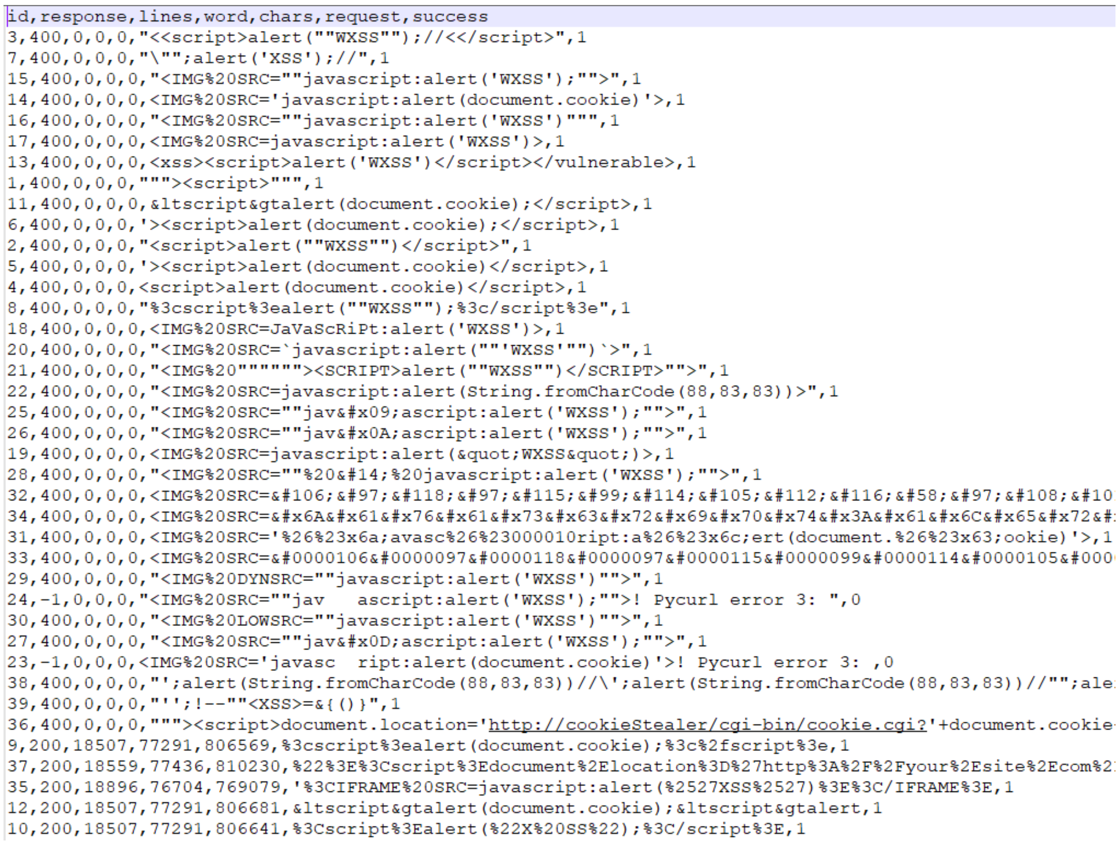
With the little fix the scanner itself was pretty stable. However when the result was specified as an HTML file it would first open a lot of alerts and then redirect the browser to a page called 'http://cookiestealer/cgi-bin/cookie.cgi?' as those were the attacks the Wfuzz tried on the webpage and so listed in an unsecure HTML tag. So the output file format had to be changed to another type as it otherwise would directly reroute to this fake page.

4.7.5 Duration of the Analysis

The speed of Wfuzz is not really comparable with the other scanners as Wfuzz doesn't have a crawl function and only tests a few input fields. As Wfuzz isn't a scanner but only a fuzzer so not all websites were checked. The scans all took less than ten seconds.

4.7.6 Complexity of interpreting the Results

We tried two different output formats after HTML. The first was a CSV file which was very confusing. The second was a JSON file, which, even after formatting, wasn't quite clear. Both showed all requests which were made with their Id, the response code they got, how long the resulting response was, the request which was made and a number if they were successful.



```
id,response,lines,word,chars,request,success
3,400,0,0,0,"<<script>alert(""WXSS"");//<</script>",1
7,400,0,0,0,"\"";alert('XSS');//",1
15,400,0,0,0,"<IMG%20SRC=""javascript:alert('WXSS');"">",1
14,400,0,0,0,<IMG%20SRC='javascript:alert(document.cookie)'>,1
16,400,0,0,0,"<IMG%20SRC=""javascript:alert('WXSS')""",1
17,400,0,0,0,<IMG%20SRC=javascript:alert('WXSS')>,1
13,400,0,0,0,<xss><script>alert('WXSS')</script></vulnerable>,1
1,400,0,0,0,"""><script>""",1
11,400,0,0,0,&ltscript&gtalert(document.cookie);</script>,1
6,400,0,0,0,'><script>alert(document.cookie);</script>,1
2,400,0,0,0,"<script>alert(""WXSS"")</script>",1
5,400,0,0,0,'><script>alert(document.cookie)</script>,1
4,400,0,0,0,<script>alert(document.cookie)</script>,1
8,400,0,0,0,"%3cscript%3ealert(""WXSS"");%3c/script%3e",1
18,400,0,0,0,<IMG%20SRC=JaVaScRiPt:alert('WXSS')>,1
20,400,0,0,0,"<IMG%20SRC=`javascript:alert(""'WXSS'"")`>",1
21,400,0,0,0,"<IMG%20""""""><SCRIPT>alert(""WXSS"")</SCRIPT>"">",1
22,400,0,0,0,"<IMG%20SRC=javascript:alert(String.fromCharCode(88,83,83))>",1
25,400,0,0,0,"<IMG%20SRC=""jav&#x09;ascript:alert('WXSS');"">",1
26,400,0,0,0,"<IMG%20SRC=""jav&#x0A;ascript:alert('WXSS');"">",1
19,400,0,0,0,<IMG%20SRC=javascript:alert(&quot;WXSS&quot;)>,1
28,400,0,0,0,"<IMG%20SRC=""%20&#14;%20javascript:alert('WXSS');"">",1
32,400,0,0,0,<IMG%20SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#10
34,400,0,0,0,<IMG%20SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C&#x65&#x72&#
31,400,0,0,0,<IMG%20SRC='%26%23x6a;avasc%26%23000010ript:a%26%23x6c;ert(document.%26%23x63;ookie)'>,1
33,400,0,0,0,<IMG%20SRC=&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#000
29,400,0,0,0,"<IMG%20DYNSRC=""javascript:alert('WXSS')"">",1
24,-1,0,0,0,"<IMG%20SRC=""jav   ascript:alert('WXSS');"">! Pycurl error 3: ",0
30,400,0,0,0,"<IMG%20LOWSRC=""javascript:alert('WXSS')"">",1
27,400,0,0,0,"<IMG%20SRC=""jav&#x0D;ascript:alert('WXSS');"">",1
23,-1,0,0,0,<IMG%20SRC='javasc  ript:alert(document.cookie)'>! Pycurl error 3: ,0
38,400,0,0,0,"';alert(String.fromCharCode(88,83,83))//\';alert(String.fromCharCode(88,83,83))//"";ale
39,400,0,0,0,"'';!--""<XSS>=&{()}",1
36,400,0,0,0,"""><script>document.location='http://cookieStealer/cgi-bin/cookie.cgi?'+document.cookie
9,200,18507,77291,806569,%3cscript%3ealert(document.cookie);%3c%2fscript%3e,1
37,200,18559,77436,810230,%22%3E%3Cscript%3Edocument%2Elocation%3D%27http%3A%2F%2Fyour%2Esite%2Ecom%2
35,200,18896,76704,769079,'%3CIFRAME%20SRC=javascript:alert(%2527XSS%2527)%3E%3C/IFRAME%3E,1
12,200,18507,77291,806681,&ltscript&gtalert(document.cookie);&ltscript&gtalert,1
10,200,18507,77291,806641,%3Cscript%3Ealert(%22X%20SS%22);%3C/script%3E,1
```

Figure 4.9: Results of a scan presented as a CSV file

4.7.7 Completeness of the Findings

Even though Wfuzz did find flaws in both etoa.ch and playit.ch it also found 39 vulnerabilities in coop.ch, but after checking them manually none of them could be verified. It might be, that the result file shows all requests the fuzzer made but then none of the flaws were found as they all had the same number indicating if the request was successful.

4.7.8 Result

We only ran the application on the four vulnerabilities that were already known, as the fuzzer had to have a specific input. Even though the application runs smoothly and fairly quickly, the results are most confusing and we couldn't understand them at all. It also doesn't help, that no documentation on how to interpret the results was available.

4.8 Grendel-Scan

4.8.1 Description

Grendel Scan is another Java based scanner. The developer David Byrne does not provide any documentation outside of the source code.

4.8.2 Problem with the Installation

First we tried to install the file provided on sourceforge (`https://sourceforge.net/projects/grendel/files/`) but when unpacking it, the ZIP folder only contained a bunch of JAR libraries. Even when directly trying to convert it to a EXE file with the online tool Ezyzip (`https://www.ezyzip.com/convert-tar-gz-to-exe.html`) no executable program could be obtained. So the source code was cloned and opened it with the IDE, Intellij IDEA 2018.3.5, but even then there were several issues. Not all dependencies could be satisfied as some packages like 'com.grendelscan.fuzzing' were missing but marked as imports in other files. In the issues list on sourceforge (`https://sourceforge.net/p/grendel/bugs/`) both problems, that the file couldn't be executed and that the project wouldn't build, were marked as open issues. So Grendel-scan seems to be unable to perform any scans.

4.9 Arachni

4.9.1 Description

Arachni is another general web application security scanner written in Ruby. It's developed by Tasos Lasok, a web security expert form Bulgaria. Arachni has three different ways to use it. It can either be used via CLI, or over a Web user interface. It can also be imported as a Ruby framework for projects to be directly used for integration testing [47].

4.9.2 Installation and Dependencies

The helpful download page gives a user several options, depending on the operating system. The website itself suggests using the Linux version for best experience, so we downloaded this version and unpacked it via WSL. As Ruby version 2.7 was already installed no further add-ons had to be installed. With the download the Web UI as well as the CLI were both included.

4.9.3 Complexity of the Interface and usage

Trying the Web UI first as it seemed promising. It had an authentication wall and even though user credentials for two default users were given somehow the login wouldn't work with either of them. So we checked the 'seeds.rb' file if those two were actually initialized and then a check if a database was present, which it was under the file 'production.sqlite3'. As there didn't seem to be an easy solution and the command line option with mostly the same functionalities was still available we did not further investigate into what the exact problem was. So we used the CLI version of Arachni on the WSL. After a successful scan the results have to be further processed through Arachnis reporter, which then creates a file in a defined format. Again the HTML format was used.

4.9.4 Soundness of the Application

The CLI application runs smoothly and appears to not have any major problems. The Web UI version seems to have some issues with authentication. Also it somehow wouldn't scan the website of Coop. Other than that it appears to be a pretty stable scanner.

4.9.5 Duration of the Analysis

The speed of the analysis depended widely on the website. Especially since Arachni made a lot more requests and scanned less pages then other scanners. For the most part it is a rather slow scanner.

4.9.6 Complexity of interpreting the Results

The reports made by the reporter are very beautiful. They have several pages. The first is a summary with all the flaws found in form of different graphs, where a bar graph shows how many of which issues where found and three separate pie graphs show their severity, what type of HTML-element it was and if it was a trusted or untrusted element. It then has on a second page all the different vulnerabilities. For each it has first a paragraph in which the general issue is explained. Underneath is the list of all found vulnerabilities. They are listed with the HTML-tag type, the name of the input parameter, what kind of HTML method it was and the general page. For each flaw it has a button where further information such as the injected seed, the proof from the response body and the exact request URL are shown. It also has a text for a possible solution, for XSS it is about input sanitization but also includes encoded inputs like '&lt' for '$<$'.

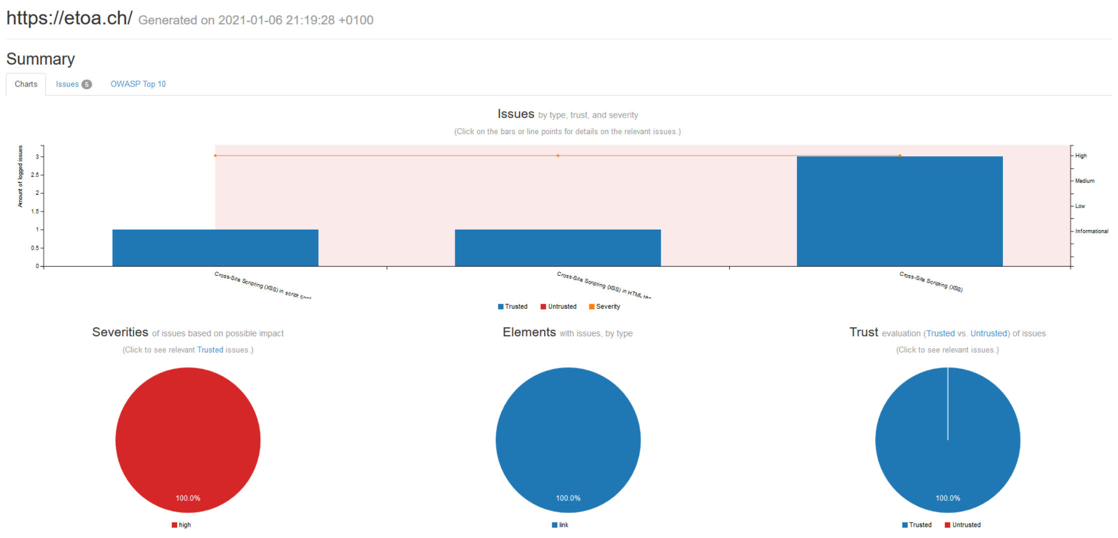
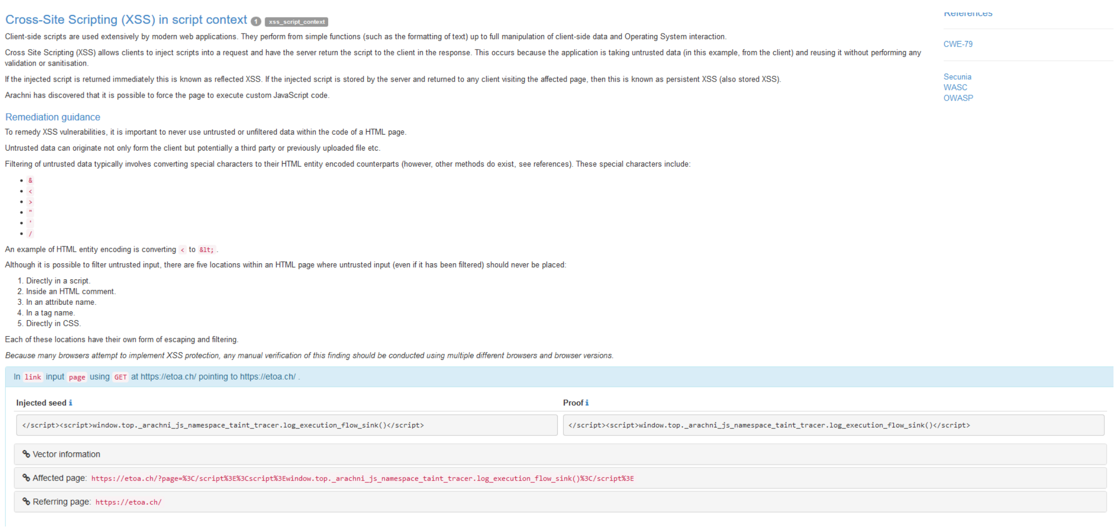


Figure 4.10: Summary of all vulnerabilities found by Arachni in a scan


Cross-Site Scripting (XSS) in script context 1 xss_script_context

Client-side scripts are used extensively by modern web applications. They perform from simple functions (such as the formatting of text) up to full manipulation of client-side data and Operating System interaction.

Cross Site Scripting (XSS) allows clients to inject scripts into a request and have the server return the script to the client in the response. This occurs because the application is taking untrusted data (in this example, from the client) and reusing it without performing any validation or sanitisation.

If the injected script is returned immediately this is known as reflected XSS. If the injected script is stored by the server and returned to any client visiting the affected page, then this is known as persistent XSS (also stored XSS).

Arachni has discovered that it is possible to force the page to execute custom JavaScript code.

Remediation guidance

To remedy XSS vulnerabilities, it is important to never use untrusted or unfiltered data within the code of a HTML page.

Untrusted data can originate not only form the client but potentially a third party or previously uploaded file etc.

Filtering of untrusted data typically involves converting special characters to their HTML entity encoded counterparts (however, other methods do exist, see references). These special characters include:

- `&`
- `<`
- `>`
- `"`
- `'`
- `/`

An example of HTML entity encoding is converting `<` to `&lt;`.

Although it is possible to filter untrusted input, there are five locations within an HTML page where untrusted input (even if it has been filtered) should never be placed:

1. Directly in a script.
2. Inside an HTML comment.
3. In an attribute name.
4. In a tag name.
5. Directly in CSS.

Each of these locations have their own form of escaping and filtering.

Because many browsers attempt to implement XSS protection, any manual verification of this finding should be conducted using multiple different browsers and browser versions.

In `link` input `page` using `GET` at https://etoa.ch/ pointing to https://etoa.ch/ .

Injected seed

```
</script><script>window.top._arachni_js_namespace_taint_tracer.log_execution_flow_sink()</script>
```

Proof

```
</script><script>window.top._arachni_js_namespace_taint_tracer.log_execution_flow_sink()</script>
```

Vector information

Affected page: `https://etoa.ch/?page=%3C/script%3E%3Cscript%3Ewindow.top._arachni_js_namespace_taint_tracer.log_execution_flow_sink()%3C/script%3E`

Referring page: `https://etoa.ch/`

References

CWE-79

Secunia
WASC
OWASP

Figure 4.11: Detailed description of a single found vulnerability

4.9.7 Completeness of the Findings

For etoa.ch it found both flaws on the input parameters ‘page’ and ‘faq’.

Also for playit.ch it did find the flaw for the input field ‘suchbegriff’ as a reflected, as well as a DOM based issue.

For coop.ch it didn’t find anything, even when directly scanning the search page.

Arachni did find a problem with Manor’s full text search input field ’q’, but after trying the seed Arachni provided as well as others from Chapter 2 no proof could be found that the attack input would slip out of a safe field.

4.9.8 Result

Arachni is an easy to use and reliable security scanner. It did not find the flaw on coop.ch, but as the result page of the report explicitly states this form of flaw, i.e. the encoding of angle brackets, the problem was more likely with the crawl rather than the vector injection. From all scanners this one has the most detailed results. It does a lot of requests per input field which is very time consuming, so it might not be well-suited for broader searches, but on the other hand it’s very rigorous. The developer stopped maintaining the application in 2020, so it might be out of date pretty quick [48].

4.10 GoLismero

4.10.1 Description

GoLismero is also a Python based scanner by the security tool creator Daniel Garcia. Like most Python scanners it works through a CLI. The interesting part about GoLismero is it aims to unify different smaller web application security tools into one. Those tools can be enabled or disabled for each scan as the user likes [38].

4.10.2 Installation and Dependencies

The installation was quite easy. For the WSL there is a complete description in the documentation on the Github repository, which after following it, should have taken care of all required dependencies. It even explains how to install Python.

4.10.3 Complexity of the Interface and usage

The documentation of GoLismero is quite brief and so in the beginning it isn’t immediately clear how to run a scan, but after a bit of trying the scan could be configured to only include the crawl of a website and then the XSS attacks through a tool called XSSer.

4.10.4 Problems with running

We tried the code on several different websites but would always get the same error. As soon as the application started running the XSSer, it had a problem with loading different Python modules, even though they were previously installed on the machine and

worked with different applications. Apparently the problem as described here (`https://github.com/golismero/golismero/issues/54`) is that, similar to W3af, GoLismero
work with Python 3 so there is a split in the dependencies which produces this error, so until GoLismero is updated it cannot be used.

4.11 Nikto/Wikto

4.11.1 Description

Nikto is a multipurpose web server assessment tool written in Perl and developed by Chris Sullo. Its main goal is, according to their website, to find server and software misconfigurations, insecure files and outdated servers and programs. So it is not specifically programmed to find XSS vulnerabilities [58]. Wikto is as the authors explain Nikto for Windows with some extra features. It's written in C"# and aims to simplify the Nikto installation for Windows users [63].

4.11.2 Installation and Dependencies

The first thing we tried was to install Wikto. As it proclaims to be an extension of Nikto, it should have all of the same functionalities, but after trying to install it there were some problems which couldn't be fixed and so Wikto couldn't be installed. So only Nikto was installed and tested. This was fairly easy over the WSL as Perl was already installed on the system.

4.11.3 Complexity of the Interface and usage

Nikto isn't primarily used to test websites remotely. Its main goal is to test servers locally. And therefore the documentation isn't structured in a way to instantiate such a remote injection scan, but after carefully reading the documentation, the starting arguments could be configured to only test for injection attack forms. What Nikto doesn't offer is a way to crawl through the whole website, so it is similar to Wfuzz in this regard, but instead of having to define one specific injection point it would test the whole page.

4.11.4 Soundness of the Application

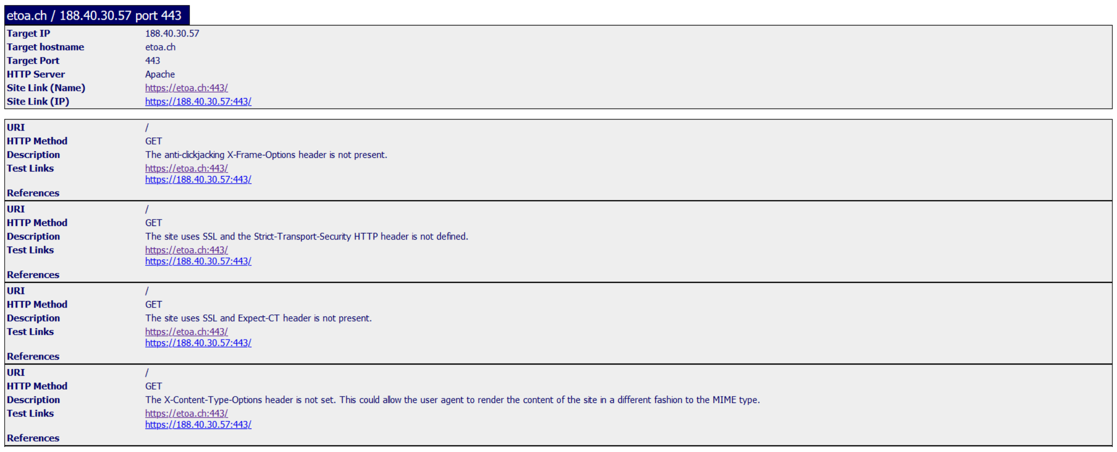
Nikto ran smoothly and did not produce any unexpected error messages.

4.11.5 Duration of the Analysis

The pages were scanned pretty quickly. Usually a scan took less then 2 minutes, but as it only tested single pages it's not really comparable to the other scanners.

4.11.6 Complexity of interpreting the Results

The findings are listed in a HTML file. The file contains a brief summary with how many requests were made, the time it took and what CLI input was given. Then it lists all vulnerabilities found with the URL, the HTTP method and a quick description of the flaw. To find the results of a certain type the user has to read through all the descriptions as the findings aren't categorized or in any form ordered.



etoa.ch / 188.40.30.57 port 443

Target IP	188.40.30.57
Target hostname	etoa.ch
Target Port	443
HTTP Server	Apache
Site Link (Name)	https://etoa.ch:443/
Site Link (IP)	https://188.40.30.57:443/

URI	/
HTTP Method	GET
Description	The anti-clickjacking X-Frame-Options header is not present.
Test Links	https://etoa.ch:443/ https://188.40.30.57:443/
References	
URI	/
HTTP Method	GET
Description	The site uses SSL and the Strict-Transport-Security HTTP header is not defined.
Test Links	https://etoa.ch:443/ https://188.40.30.57:443/
References	
URI	/
HTTP Method	GET
Description	The site uses SSL and Expect-CT header is not present.
Test Links	https://etoa.ch:443/ https://188.40.30.57:443/
References	
URI	/
HTTP Method	GET
Description	The X-Content-Type-Options header is not set. This could allow the user agent to render the content of the site in a different fashion to the MIME type.
Test Links	https://etoa.ch:443/ https://188.40.30.57:443/
References	

Figure 4.12: Results of a scan made by Nikto

4.11.7 Completeness of the Findings

The scans did find lots of misconfigurations on the websites but not a single XSS vulnerability was found. Even when checking the exact pages of the known flaws Nikto wouldn't report them.

4.11.8 Result

Nikto didn't find any of the flaws. Since it isn't the main focus of the application it might not be that surprising but as it clearly states, that XSS vulnerabilities can be found it at least should find the basic plain text ones on etoa.ch and playit.ch.

4.12 Scanners out of scope

Only the scanners above were tested. The following are also open-source scanners from OWASP and Pavitar Shankdhars lists but were all excluded for different reasons as provided:

- **SQLMap** since it is specifically for SQL Injection vulnerabilities and therefore cannot test for XSS.
- **Watcher** because it is an extension of an application called Fiddler which has different versions.
- **X5S** is, similar to Watcher, a plugin for Fiddler.
- **WebScarab** is an older project, also developed by OWASP and some sources say ZAP is its official successor.
- **purpleteam** is also a project developed by OWASP. It is used as an API to continuously check a locally stored web application and not to test the site through a browser.
- **Sec-helpers** is a bundle of very basic security validators which can be used by developers to test if a domains protocol standards meet current recommendations. It has no functionality which would do the same for XSS attacks [17].
- **Ride (REST JSON Payload fuzzer)** is an extension to Adobes Ride, a Java REST API automation framework. It is not really an application but a framework that can be integrated into a project.

4.13 Comparison

Finally we want to compare the different scanners with the criteria as we defined them in section 3.2 to see which of the scanners are better suited to test the XSS security of websites.

4.13.1 Installation and Usability

First we compare the scanners on how easily they can be installed and on their usability. As for the installation part we take a look at how many manual steps a user has to make until the application is installed. A step is defined as either one line of code in a command line or a single click with the mouse. We also try to measure the time it took us to install the software. Here we also take into account the time needed to troubleshoot when a problem appeared and the time needed to download data.

For the usability we look at how many steps a user has to take until the scan starts with the needed configurations or, if we have a CLI, how many arguments we have to put so the scan works with as we intend to. We also look at the documentation, first how detailed it is. For this we give one of the following four marks: ++, +, -, –. A higher mark means the documentation describes the software's function in more detail. The same marks are given for the structure of the documentation. Here the highest mark means the documentation is well-structured and can be easily searched.

Comparison Installation and Usability					
Scanner	Installation		Usability	Documentation	
GUI	Steps	Time	Steps to scan	Detailed	Structured
ZAP	7	12min	3(11)*	++	++
Vega	6	4min	4	+	+
CLI	Steps	Time	Arguments	Detailed	Structured
Skipfish	36	1.5h	5	-	+
Arachni	8	7min	5	++	-
Wapiti	50+	4.5h**	7	–	–
Wfuzz	1	2min	7	+	++
Nikto	1	5min	4	-	–

*For the first scan with those configurations
**Including upgrading WSL

4.13.2 Duration and Soundness

To compare the duration we tried to measure the time each of the ten scans took and then averaged it.

For the robustness we look at how often a scanner freezes or crashes during the scan of Coop's website as a reference. And for the second criteria we count how often we have to restart or cancel a scan because it had a problem, didn't scan a site properly or seemed to be stuck. Here we count several failures of the same scanner on the same website as one.

Comparison Duration and Soundness			
Scanner	Duration	Soundness	
GUI	Average scan	Crashes	Failed scans
ZAP	5h 12min	1	6
Vega	12h 30min	5	0
CLI			
Skipfish	3h 39min	0	4
Arachni	11h 57min	0	4
Wapiti	9h	0	6
Wfuzz*	1min	0	0
Nikto*	1min	0	0

*Only scan a very small amount of a website

4.13.3 Findings and Results

To judge the completeness of the results try to find out which of the vulnerabilities each scanner finds. Most scanners cannot detect the encoded vulnerabilities like the one found in Coop's website but only the plain texted ones.

For the results the first criteria is if the found flaws are grouped in a sensible way, which if they are grouped in categories that make sense, e.g. all XSS flaws together. The other criteria are if we can find out on which webpage the flaw is located, if we see the seed, i.e. the request made by the scanner to detect the flaw, and finally if the result shows the proof it found in the response body.

Comparison Findings and Results					
Scanner	Findings	Results			
GUI	Completeness	Grouped	Page	Request	Proof
ZAP	All flaws	Yes	Yes	Yes	Yes
Vega	Plain text	Yes	Yes	Yes	No
CLI					
Skipfish	Plain text	Yes	Yes	Yes	No
Arachni	Plain text	Yes	Yes	Yes	Yes
Wapiti	Plain text	Yes	Yes	Yes	Yes
Wfuzz	Plain text	No	Yes	Yes	No
Nikto	Plain text	No	Yes	No	No

4.13.4 Conclusion

While not all Scanners are as easily comparable and might not have been made with the same intended use, some might have the goal to test whole websites while others are programmed to work best for integration testing where new pages can be checked automatically, there definitely are some superior to others. For example if someone prefers a Graphical User Interface or just would like to be able to persist and reuse certain test specifications, ZAP is definitely better suited than Vega even though they both follow the same concept. If a user wants to quickly check the whole website for several vulnerabilities Skipfish might be preferred but if only a small section should be tested for a certain flaw then Arachni or Wapiti might be better suited.

Overview Scanners					
Scanner	Findings	Installation	Usability	Robustness	Duration
GUI					
ZAP	complete	easy	very good	medium	fast
Vega	only unencoded	easy	good	bad	medium
Grendel Scan	*Couldn't get it to run*				
CLI					
Skipfish	only unencoded	easy	medium	good	very fast
Arachni	only unencoded	easy	good	medium	slow
Wapiti	only unencoded	hard	medium	medium	slow
Wfuzz	unclear	easy	good	good	incomparable
Nikto	none	easy	good	good	incomparable
W3af	*Couldn't get it to run*				
Grabber	*Couldn't get it to run*				
GoLismero	*Couldn't get it to run*				

5 Analysis and Mitigation of found Flaws

After a Web Crawler reports Cross Site Scripting vulnerabilities on a website it is important to first analyze the supposed flaw and then, if it is deemed a security relevant vulnerability, mitigate it.

5.1 Analyzing the results of a Web Crawler

Each Web Crawler has a different way to present its findings but all of them show at least the request and usually the response they got from the webserver.

5.1.1 Analyzing the Request

First the request must be checked. HTTP-requests contain some general data like browser used, cookies, and other more technical details. Important for analyzing it are only the method used and the URL path.

The method is important because it shows us if an attack might be persistent if the method is for example ‘PUT’ or ‘POST’. If the attack is reflected or DOM based, then it’s usually a ‘GET’ request.

In the URL path the page of the website which is impacted can be found. It also shows if the problem stems from a specific input-field and if so which code broke it.

As an example here is the request from an alarm reported by skipfish:

GET /help/content/wiki?page=faq&faq=270-->”>’>’”<sfi000921v703569>

First it is a GET HTTP-Request so it's definitely not persistent. A quick look into the page's HTML-code shows that the vulnerability is reflected because it is not dynamically generated in any form.

The problematic field here is a field called 'faq' that broke with the input "270-->">'>'"<sfi000921v703569>". Here the method used by skipfish can be observed. It first put escape sequence characters in an attempt to get out of the safe zone and then set a new tag, here called 'sfi000921v703569'. Afterwards it will check the response body if it finds this tag somewhere.

5.1.2 Analyzing the Response

In the response can be seen where the supposed flaw is returned. This helps to check if this really is a vulnerability or if it's a false positive, as it might be, when the tag is still in a secure zone where code wouldn't be parsed. It could also be that other checks prevent a malicious attack. The easiest way is to actually just replace the fake tag '<sfi000921v703569>' with the code used in manual testing <script>alert('Rafi')</script>. If then a pop-up dialog opens it's certain that the page is flawed. If the dialog doesn't pop-up the code can be tweaked as explained in Chapter 2, or the HTML can be analyzed to see if different inputs in form of HTML tags create unexpected behavior of the webpage.

For suspected DOM based XSS it is important to find out if the value of an input field is used in a XSS function which uses this input to form the page. Some of these functions are for example:

- innerHTML()
- outerHTML()
- document.write()
- document.writeln()

The problem with those functions is they all return their input in an unsafe way, which means if a user can write HTML tags in those functions and the input is reflected back to the page those tags are parsed and interpreted as such [24].

Another bad function is the eval() function. This function takes a string and interprets it as a JavaScript function [30]. If a user can put his own code in there it might have a terrible outcome.

5.2 Mitigation tactics

There are several different kinds of mitigation tactics. The vast majority of them can be categorized into one of three categories:

- Input sanitization
- Output Encoding
- Mitigating through CSP

It usually isn't enough to just use methods of one of these categories, but if correctly used together they can be a powerful defense against most, if not all XSS attacks [25].

5.2.1 Input Validation and Sanitization

Input validation is very important. Before using input of any user, it is validated to see if it is in an expected form and does not contain any code or data which could compromise the system. There are two possible ways to validate the value of an input. With whitelisting, so only certain values for the input are allowed and an error is returned if the input isn't one of the expected ones. This works especially well for fields with only a limited number of possible values. For example if a field is looking for a city, a check can be performed where first the input has to be a safe protocol like HTTP or HTTP and then test if it corresponds with a city from an existing database. If the protocol has an unexpected type like 'javascript' or 'data' or the searched city can't be found the input isn't returned.

The second but much less secure possibility is to blacklist certain protocols or keywords. This list must be very extensive and updated frequently. Otherwise with new technologies the safety of blacklisting dwindles [21].

5.2.2 Output Encoding

Before returning user input back to the site it is important to properly encode or filter special characters or keywords in a way that a browser doesn't execute any unintended code. This works really well if all special characters like <, >, \, /, &, ", ', (,), #, %, ; , +, - can be filtered out. If some of these special characters cannot possibly be filtered because they are indispensable it gets a lot more complex and insecure since different encodings have to be taken into account. For example &lt or \u003c for <, and keywords like 'script', javascript' and many more. Even if only a few of those are missing the security of the system can be compromised [46].

There are some open source libraries which do those kind of sanitization automatically, for example OWASP does maintain a Java and JavaScript library called Enterprise

Security API (ESAPI) which has an Encoder interface that can encode a String for HTML or for JavaScript which makes the returned value safe to use. However, ESAPI's maintenance seems to be slowing, which means the safety it provides might decline in the future [61].

5.2.3 Mitigating through CSP

Content Security Policy (CSP) is a protocol that serves as a safety net, which has to be implemented in HTTP headers. With CSP a developer can define which domains are trusted. Only scripts from those domains will be executed. Others, including inline scripts can be disallowed. For ultimate protection a developer can even opt out of all client side script execution. CSP is defined by a website provider but the execution of the policies lies in the browsers. Therefore, not all browsers might be able to check the full extent of a CSP. CSP is fully backwards compatible which means no matter the browser the website is still accessible. If a browser detects a violation of the CSP it blocks the HTTP-Request and, if defined, reports it.

To configure the Content-Security-Policy in a header there are several different arguments which can be defined. Especially important for XSS security is the keyword script-src <source>. If this argument is given only scripts from the defined <source> are executed. Note that the source can either be a 'self' or a URL like 'http://url.ch' or 'http://*.url.ch', here the * stands for a wildcard, which means it matches all subdomains of the website. The source can also be protocols like HTTP (http:) or HTTP (https:). It is also possible, but not recommended, to define data types Binary Large Object (Blob) or mediastreams. With the argument 'none' no scripts are allowed. It is also possible to weaken the security if necessary, for example the argument 'unsafe-eval' allows the eval() function to be executed, 'unsafe-inline' allows inline elements like the <script> tag. These should only be used if absolutely necessary [8].

If script-src is not defined CSP falls back onto the default-src keyword. Each undefined argument will always fall back onto this one, so it is vital to carefully choose the sites which here should be allowed [22].

Google did develop a CSP Evaluator which helps developers to evaluate the strength of their CSP against XSS attacks [23].

With a well defined CSP a website can get an additional layer of security but it cannot be the only one as some browsers like Internet Explorer only marginally support CSP, other users might not be up to date with their current browser and therefore don't support all keywords [8].

6 Conclusion and Future Work

Several Web Application Security Scanners were tested for their efficacy in detecting XSS flaws. During these tests we gained further knowledge into the inner workings of these kinds of scanners. As those scanners were only tested for their ability to detect Cross-Site Scripting vulnerabilities there is still a lot to be tested. Most other attack vectors which these Scanners could detect were left out, as well as more advanced commercially available scanners like those detailed on OWASPs website (`https://owasp.org/www-community/Vulnerability_Scanning_Tools`). Mitigation tactics weren't discussed in great detail and neither were any of the libraries mentioned checked for efficacy nor were the different tactics demonstrated on the websites tested.

7 Anleitung zu wissenschaftlichen Arbeiten

In this chapter we will explain in greater detail how those Web Application Security Scanners which successfully ran on the Windows Subsystem for Linux were installed. The exact command with which the scans were run with will also be provided.

Skipfish

Download and Installation

From the download page (`https://code.google.com/archive/p/skipfish/downloads`) we downloaded the most recent version (skipfish 2-10b). It is a TAR archive compressed to a TGZ file. After extracting the files the README was extensive and very helpful. It instructed us to install the libdin library which we did but then got another problem with a dependency called 'openssl/ssl.h'. After some trial and error we found a solution on the tecadmin website (`https://tecadmin.net/install-openssl-on-windows/`) which after a little bit of tweaking worked. It was definitely more time consuming to install than Vega but not impossible. However after trying to run it with the given example code in the documentation, we recognized that it was made for Linux systems. The C-file 'skipfish' was attempted to be run with Windows commands but to no avail. In the end we decided to switch to the WSL. The extracted packages could be installed after installing two dependencies called libpcre3-dev and libidn11-dev over APT. Then the application could be installed via the Makefile and ran without any problems.

Code used to run the application

For the test the following starting parameters were used in each test:

```
skipfish -o directory -S minimal.wl -W- -L -d 8 https://website.ch
```

-o directory: specifies 'directory' as the new path which it should write its findings to -S minimal.wl: The minimal dictionary already provided by skipfish was used. -W-: specifies to not write newly learned paths anywhere -L: forbids the application to auto-learn new keywords d 8: tells the application to not crawl further than 8 links deep

Wapiti

Download and Installation

The source code was downloaded from here `https://sourceforge.net/projects/wapiti/files/wapiti/wapiti-3.0.3/`. To run it the Python version had to be upgraded to 3.5. Then it was tried to run the setup file as described which should download and install all dependencies, but apparently it had a missing module named 'setuptools' in its imports which had to be installed first. After that the setup ran. It did prompt this warning:

'WARNING: The C extension could not be compiled, speedups are not enabled.

Plain-Python build succeeded.' After finding out this comes from a different missing module called 'python-dev' the setup ran into another problem: with a missing dependency called 'MarkupSafe' and finally after installing that Wapiti was able to run. Running it threw another error: 'SyntaxError: invalid syntax'. After some research a possible solution was found here `https://github.com/flairNLP/flair/issues/1469`. The author had solved a similar problem by upgrading their Ubuntu system. The same was done for the WSL, but as an upgrade to version 18.04 wasn't successful WSL had to be reinstalled with version 20.04.

Code used to run the application

For our purposes the following starting argument were used:

```
wapiti -u http://website --scope folder -m xss -f html -v 2 -d 8
```

-u http://website: defines the URL that should be scanned

--scope folder: defines the scope. A user can define the scope as big as they want. They could also choose to attack all linked websites as well. Folder was chosen, which means it scans all URLs beneath the one defined as the start URL, as the other scanners usually would scan all URLs underneath the base too.

-m xss: defines the modules used, this option only scans for XSS

-f html: gives the output format. The HTML output was deemed to be the best, it does also have options for XML and txt files but both seem less readable than the HTML.

-v 2: is the verbosity during the scan. It has 3 levels (0-2) where 0 gives no information, 1 gives an update if an error occurred and 2 shows all HTTP-Requests which are made. 2 was chosen since otherwise it would be unknown if the scanner was still running.

-d 8: restricts the depth of the search to 8 nodes, otherwise it would default to 40.

Wfuzz

Download and Installation

The installation via Python pip was only one command:

pip install wfuzz

Code used to run the application

The following arguments were used:

wfuzz -o html -w wordlist/Injections/XSS.txt -u https://website/.../parameter=FUZZ -v

-o html: defines the results file type

-w wordlist/. . . : defines the list of the payloads which should be used.

-u https://...: defines the URL to the webpage. The FUZZ at the end will be replaced by the different payloads during the scan.

-v: defines that the scan should be verbose

As the attack vectors are saved as a TXT file, it's easy to check and modify them.

None of the analysis could finish as they all stopped with the error "Fatal exception: Pycurl error 3:". Thankfully on the issues list of wfuzzes Github the developer gave a solution (`https://github.com/xmendez/wfuzz/issues/138`) by just amending '-Z' to the arguments.

Arachni

Download and Installation

From the download page (`https://www.arachni-scanner.com/download/`) the Linux x86 64bit version was downloaded as a tar.gz file. it was unpacked via WSLs inbuilt tar extractor and then the following two commands were run, as stated on the installation wiki (`https://github.com/Arachni/arachni/wiki/Installation`) to get all dependencies and install arachni:

```
sudo apt-get install build-essential curl libcurl3 libcurl4-openssl-dev ruby ruby-dev
gem install arachni
```

Code used to run the application

The Github wiki page under:
`github.com/Arachni/arachni/wiki/Command-line-user-interface`
has an extensive list of all usable starting arguments. Through the documentation the following command input was decided on:

```
arachni https://website --output-verbose
--scope-directory-depth-limit 8 --checks xss* --report-save-path website
```

As the arguments are written out they are pretty much self-explanatory. Depth was restricted to 8 nodes as in previous scan. The '--checks:xss*' argument means the scanner only checks for Cross-Site Scripting vulnerabilities. The save path saves the report as an AFR file. Which can then be converted to a html file with the 'arachni_reporter' through the command:

```
arachni_reporter website.afr –reporter=html:outfile=website.html.zip
```

GoLismero

Download and Installation

With the following code the Github repository was cloned and the applications dependencies installed:

```
sudo git clone https://github.com/golismero/golismero.git
cd golismero
sudo pip install -r requirements.txt
sudo pip install -r requirements_unix.txt
```

Code used to run the application

After installation the following code was used to run GoLismero:

```
golismero scan https://website.ch/ -o - -o website.html -e dns* -e spider* -e xsser*
```

Scan: specifies the URL of the site -o - -o website.html: defines the output format of the report the first empty one means the report will be printed on the Command Line Interface, the second will produce an HTML report. -e: enables the specified tool, here dns* and spider* are tools needed to crawl the website. Then XSSer is a XSS fuzzer which would check different XSS specific payloads.

Nikto/Wikto

Download and Installation

Trying to install Wikto

The first thing that was done was downloading the) ZIP folder from the official release page (`https://github.com/sensepost/wikto/releases/tag/2.1.0.0`) then extract it and install the file. The installation wizard gave an error where it suggests to install a version of Microsofts .NET as it is needed, but even after installing this version and linking it in the PATH it still wouldn't recognize it. The problem most likely stems from this requirement, as the Wikto release is from 2008 and therefore might not be compatible with newer Windows operating systems or the current version of .NET. Instead only Nikto was installed and tested.

Installation of Nikto

To install Nikto only the github repository had to be cloned:

```
git clone https://github.com/sullo/nikto
```

As Perl was already installed Nikto could run without any issues.

Code used to run the application

To run Nikto the following arguments were used:

```
perl nikto.pl -h https://website -T 4 -output website.html
```

-h: defines the website or server that should be tested -T 4: The T stands for tuning, it defines a certain group of attack forms which should be performed. Here 4 is the group Injection. It does not only include XSS but also Script and HTML injections. -output: defines the name under which the report should be saved and with the ending also the format.

Performance of the different Scanners

The raw data on how each scanner performed on the particular websites. Different methodologies for measuring the performance of each scanner were tried. Not all of these could be determined for each scanner.

- **Duration of Scan** The time the application had for the whole scan, which includes the crawl as well as the active scan for vulnerabilities.
- **URLs discovered** How many URLs the scanner discovered during the crawl.
- **Attack Requests** How many different HTTP-Requests were made during the process
- **Remarks** When the scanners settings deviated from the ones mentioned in chapter 3.4 or if a scan was stopped manually after running it for at least 15 hours it would be remarked here.

Vega

As Vega doesn't provide any information about how long a scan was, how many URLs were discovered or requests were made, the length of the scan could only be approximated by comparing the start time of a scan with the time when it was finished. However as the computer couldn't be attended the whole day some results might be off a bit.

Scan Performance				
Website	Duration of Scan	URLs discovered	Attack Requests	Remarks
Etoa	∼3h	?	?	-
Playit	∼22h	?	?	-
Coop	∼18h	?	?	-
Orellfüssli	∼18h	?	?	-
Migros	∼7h	?	?	-
Interdiscount	∼9h	?	?	-
Sportxx	∼13h	?	?	-
Ricardo	∼16h	?	?	-
Staemme	∼1h	?	?	-
Manor	∼18h	?	?	-

Skipfish

Skipfish shows how long a analysis took and how many requests were made. It doesn't specify how many pages were scanned with these requests. As Skipfish is the only application where the kind of attack vectors it should use can't be specified, the requests are not all XSS attacks but rather all kinds of different attack forms.

Some of the websites seemed to have a protection against that many requests in such a short amount of time and would block Skipfish's requests after a few seconds, which indicated to Skipfish, that it has finished the scan and wouldn't run any more.

Scan Performance				
Website	Duration of Scan	URLs discovered	Attack Requests	Remarks
Etoa	1h 20min	? URLs	1'792'078 Requests	-
Playit	25h	? URLs	4'914'425 Requests	cancelled manually
Coop	5h 26min	? URLs	2'566'272 Requests	-
Orellfüssli	3min	? URLs	13'063 Requests	-
Migros	5s	? URLs	68 Requests	-
Interdiscount	0.33min	? URLs	510 Requests	-
Sportxx	0.66min	? URLs	71 Requests	-
Ricardo	1s	? URLs	51 Requests	-
Staemme	0.66min	? URLs	2'989 Requests	-
Manor	4h 40min	? URLs	476'986 Requests	-

Zed Attack Proxy

The DOM-based XSS scans in ZAP had to be shut off as they seemed to go on for an unreasonably long time. Some crawls also had to be stopped manually as ZAP froze with higher amounts of URLs. For some websites during the active scan it froze where the application sometimes wouldn't recover and had to be shut down. For Coop, as the active scan would freeze during the scan of the whole page, only the search page `www.coop.ch/de/search` was scanned.

Scan Performance				
Website	Duration of Scan	URLs discovered	Attack Requests	Remarks
Etoa	38.5min	3101 URLs	1513 Requests	-
Playit	1h 44min	5725 URLs	13'179 Requests	-
Coop	4h 51min	16'142 URLs	410 Requests	only on search
Orellfüssli	3h 5min	23'991 URLs	74'495 Requests	no DOM
Migros	24h 55min	16'908 URLs	23'750 Requests	no DOM, scan froze
Interdiscount	15h 32min	86'604 URLs	44'191 Requests	no DOM
Sportxx	35min	21'932 URLs	327 Requests	no DOM
Ricardo	5 min	20'023 URLs	2363 Requests	no DOM, stopped crawl
Staemme	0.5 min	442 URLs	215 Requests	no DOM
Manor	36.5 min	131'146 URLs	12'097 Requests	no DOM

Wapiti

Even though Wapiti doesn't provide a timer or timestamps for the requests as others do, the time was measured by simply subtracting the time the scan was started from the timestamp when the report was created.

It did not specify how many HTTP-Requests were made and the number could not be retrieved from the WSL terminal for most of the scans as it didn't show more than 1000 lines. When a crawl wasn't finished after 15 hours or the active scan after 10 hours they were stopped manually. The scans were restricted to 4 nodes as most scans weren't finished with the discovery phase after 15 hours.

Scan Performance				
Website	Duration of Scan	URLs discovered	Attack Requests	Remarks
Etoa	35min	?	?	-
Playit	10min	1112 URLs	?	depth 4 nodes
Coop	26h	?	?	stopped crawl and scan
Orellfüssli	15h 15min	?	?	stopped crawl
Migros	2min	5 URLs	30 Requests	didn't run properly
Interdiscount	5h 54min	?	?	depth 4 nodes
Sportxx	5h 2min	?	?	depth 4 nodes
Ricardo	13h 2min	?	?	depth 4 nodes
Staemme	0.5min	19 URLs	19 Requests	depth 4 nodes
Manor	24h 2min	?	?	depth 4 nodes, stopped crawl

Arachni

At the end of Arachnis scan it shows a statistic with how many websites it scanned, how many requests were made and so on. These numbers were taken there. The number of URLs isn't those discovered but the ones scanned, which had to be discovered as well. Many scans had to be stopped manually. Luckily it is possible to still generate a report when a scan has to be cancelled.

When a scan of coop was attempted it only scanned one or two URLs. It wouldn't go deeper no matter the starting arguments, even without a depth restriction and with the argument '—scope-include-subdomains', or the starting URL, ''/', '/de', '/fr', '/de/search'. It is unsure why it wouldn't go deeper as it always responded as if the scan was successful.

Scan Performance				
Website	Duration of Scan	URLs scanned	Attack Requests	Remarks
Etoa	11h 20min	6127 URLs	2’460’365 Requests	-
Playit	2h 46min	483 URLs	357’880 Requests	-
Coop	0.5 min	2 URLs	0 Requests	Didn’t scan properly
Orellfüssli	22h 7min	12’223 URLs	4’633’838 Requests	stopped manually
Migros	20h 54min	78 URLs	252’517 Requests	stopped manually
Interdiscount	4h 51min	586 URLs	91’631 Requests	-
Sportxx	6h 52min	508 URLs	310’155 Requests	-
Ricardo	9h 36min	1304 URLs	458’652 Requests	-
Staemme	16h 50min	1905 URLs	167’387 Requests	-
Manor	24h 17min	856 URLs	889’649 Requests	stopped manually

Nikto

The report of each scan shows the time elapsed and the number of requests that were made. However since Nikto doesn’t use a spider to discover further URLs it only scanned one URL, the main page. For the websites with a vulnerability where it isn’t on the main page a second scan on the flawed page was made as well.

Scan Performance				
Website	Duration of Scan	URLs scanned	Attack Requests	Remarks
Etoa	121s	2 URLs	2033 Requests	scans on main and help page
Playit	57s	1 URLs	1002 Requests	-
Coop	49s	1 URLs	879 Requests	-
Orellfüssli	47s	1 URLs	879 Requests	-
Migros	119s	78 URLs	879 Requests	-
Interdiscount	40s	1 URLs	879 Requests	-
Sportxx	62s	1 URLs	880 Requests	-
Ricardo	8s	1 URLs	151 Requests	-
Staemme	61s	1 URLs	879 Requests	-
Manor	31s	1 URLs	879 Requests	-

Glossary

Account Hijacking Attack form in which a user's account credentials are stolen and the affected account is accessed [41]. 2

Ad Blocker Plug in for a browser used to suppress advertisement on websites. 6

add-on Software that provides additional functionality to an appliaction. 22, 31, 34

API Application Programming Interface, interface where other applications can access its functionality [18]. 47

APT Advanced Package Tool, a package management system used by UNIX operating systems like Linux and Ubuntu [3]. 31, 49

attack vector Method used to penetrate or exploit a vulnerability on an application [41]. 14, 15, 18, 20, 21, 23, 27, 48, 51

Beta Beta phase is the evaluation phase of a software development process where the general public can access a software product and test its usability. 24, 25

Binary Large Object (Blob) A collection of binary data which can store any type of file or media [7]. 47

Black Box Testing Testing an application without looking at its source code but only on how it responds to certain inputs. 8

browser Application which allows a user to access the web. 1–4, 6–8, 13, 22, 23, 32, 44, 46, 47

CLI Command Line Interface in which a user can provide arguments to execute different commands and control applications. 17, 31, 34, 37, 41, 61

cookie Website specific data stored in a user's browser which is sent with a request to provide information about the user to the website's server [15]. 3, 14, 17, 44

CPU (Central Processing Unit) Integral part of the hardware in a computer responsible for executing instructions [34]. 10, 16, 17, 28

crawl Recursively Accessing different URLs through examining response bodies of webpages. 9, 17, 21–23, 26, 28, 32, 37, 50

CRUD Refers to the four main functions of a data storage system Create, Read, Update and Delete. 16

cURL Tool to automatically transfer data and protocols to a web server [57]. 28

CVE Common Vulnerabilities and Exposure, a public list of known security vulnerabilities [4]. 4

Docker Platform to isolate application environments into containers [5]. 32

DOM Document Object Model, API in which a document is represented as a tree structure [26]. 24–26, 36, 44, 45

eval() JavaScript function which evaluates the String Argument as a JavaScript function [30]. 2, 26, 45, 47

EXE file extension of executable Windows applications. 13, 21–23, 27, 34

Ezyzip Online tool which can compress and extract different files [35]. 34

Github Cloud based Website in which developers can store and share source code. 37, 51, 52

GUI Graphical User Interface, Interface of an application where a user can point and click on icons or text to modify its state. 31

HTML HyperText Markup Language, the basic language in which web pages are written. 1, 2, 6, 7, 9, 18, 25, 26, 28, 32, 35, 45, 47, 51

HTTP Hypertext Transfer Protocol, protocol in which internet communication is generally sent [6]. 15, 17, 28, 44, 46, 47

HTTP-Request Request sent, usually from a browser, to a web server [6]. 45, 47, 51, 54, 56

HTTP-Response Response to a request consisting of a header with general information and a body which contains the HTML code for the browser to interpret [6]. 24

IDE Integrated Development Environment, Application which supports developers through different functions like code completion, highlighting and debugging [32]. 34, 60

innerHTML() JavaScript function, takes a string as an argument which then is put into the document that called the function [11]. 26, 45

integration testing Type of testing where newly developed parts of a system are checked to see if they meet the systems standards in aspects like functionality or security. 34

Intellij IDEA Java IDE developed by JetBrains. 34

Internet Explorer Browser developed by Microsoft, maintenance was discontinued in 2020 [62]. 47

JAR Java archive consisting of Java classes, which can be used to import into other Java projects. 34

Java Object oriented programming language developed by Sun Microsystems. 13, 22, 34, 46

JRE Java Runtime Environment, the application which is used to run the JVM. 13, 23

JVM Java Virtual Machine is the virtual machine used for Java code. 13

Linux Openly available Unix based Operation System [19]. 10, 11, 34, 49, 51

log-file Automatically generated file of an application which stores all defined activities of the application. 5

malware artifical word shortened from malicious software, can be any kind of harmful program. [37]. 1, 3, 4

mediastream interface representing media content such as video or audio [13]. 47

NIST National Institute of Standards and Technology is part of the U.S. Department of Commerce responsible for innovation and industrial competitiveness [51]. 4

NVD National Vulnerability Database, provides a list of common vulnerabilities and tries to further security and compliance in web technology [14]. 4

operating system A computer's underlying software, responsible for managing the hardware and its resources [19]. 10, 13, 34

outerHTML() JavaScript function, gets the serialized HTML fragment of a defined element [12]. 45

OWASP The Open Web Application Security Project, is a non-profit organization with the goal to further web application security [52]. 4, 6–8, 22, 40, 46, 48

page-factory A design pattern in which the general web page stays as is, only a few defined elements are replaced. 13, 17

proxy A server set up between a user-webserver communication which can modify requests and responses. 22

Python General purpose programming language, developed as a scripting language by Guido van Rossum [53]. 21, 27, 31, 37, 38, 50

Python pip Pythons package management tool to install and modify Python libraries [55]. 21, 31, 32, 51

Ruby Object Oriented Programming Language developed by Yukihiro Matsumoto [1]. 34

Samy Worm Persistent and self replicating XSS attack developed by Samy Kamkar which shut down the Internet Platform MySpace [40]. 1

server Computer Hardware which shares its resources through the web. 2, 3, 5, 13, 23, 27, 44

spider Tool for automatically discovering new URL paths from a page.. 9, 23, 27, 32

terminal See CLI. 56

Ubuntu Operating System based on the Linux core [20]. 11, 27, 31, 32, 50

UI (User Interface), interface of a software which its user interacts with. 16, 34, 35

URL Uniform Resource Locator, is an address in the web containing a unique domain name and a payload [2]. 2, 3, 9, 17, 18, 22, 25–27, 32, 35, 44, 47, 50, 51, 56

UTF-8 Unicode Transformation Format 8 is a widely used character encoding scheme [16]. 7

UX Stands for the User Experience of a certain application. 15

virtual machine Software interpreting code into an executable program. 60

webmaster The person responsible for a websites features, performance and stability. 5

wizard An installation wizard is a software providing a GUI for guidance through a software installation. 13, 14, 23

write() JavaScript function, writing a string into the document that called it [9]. 45

writeln() JavaScript function, similar to write() but it adds a new line command at the end [10]. 45

WSL Windows Subsystem for Linux, optional addon of Windows which allows a user to run Linux applications on a Windows system [44]. 10, 11, 17, 21, 27, 32, 34, 35, 37, 49–51, 56

XML Extensible Markup Language, defines the format of certain objects [42]. 7, 18, 21, 22, 30, 51

Bibliography

[1] About Ruby. URL: https://www.ruby-lang.org/en/about/.

[2] The anatomy of a full path URL. URL: https://zvelo.com/anatomy-of-full-path-url-hostname-protocol-path-more/.

[3] APT. URL: https://wiki.ubuntuusers.de/APT/.

[4] CVE: Mainpage. URL: https://cve.mitre.org/.

[5] Docker overview. URL: https://docs.docker.com/get-started/overview/.

[6] HTTP request methods – what are HTTP requests? URL: https://rapidapi.com/blog/api-glossary/http-request-methods/.

[7] Mozilla: Blob. URL: https://developer.mozilla.org/de/docs/Web/API/Blob.

[8] Mozilla: Content security policy (CSP). URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP.

[9] Mozilla: Document.write. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document/write.

[10] Mozilla: Document.writeln. URL: https://developer.mozilla.org/en-US/docs/Web/API/Element/writeln.

[11] Mozilla: Element.innerHTML. URL: https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML.

[12] Mozilla: Element.outerHTML. URL: https://developer.mozilla.org/en-US/docs/Web/API/Element/outerHTML.

[13] Mozilla: Mediastream. URL: https://developer.mozilla.org/de/docs/Web/API/Mediastream.

[14] NVD: General information. URL: https://nvd.nist.gov/general.

[15] PcMag: cookie. URL: https://www.pcmag.com/encyclopedia/term/cookie.

[16] PcMag: UTF-8. URL: https://www.pcmag.com/encyclopedia/term/utf-8.

[17] sec-helpers 0.3.2, download page. URL: https://pypi.org/project/sec-helpers/.

[18] What is an API? URL: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces.

[19] What is Linux? URL: https://www.redhat.com/en/topics/linux/what-is-linux.

[20] What is Ubuntu? URL: https://help.ubuntu.com/lts/installation-guide/s390x/ch01s01.html.

[21] Cross site scripting prevention cheat sheet, November 2020. URL: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

[22] CSP: default-src, December 2020. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/default-src.

[23] CSP evaluator, January 2020. URL: https://github.com/google/csp-evaluator.

[24] DOM based prevention cheat sheet, 2020. URL: https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html.

[25] How to prevent XSS, 2020. URL: https://portswigger.net/web-security/cross-site-scripting/preventing.

[26] PortSwigger, DOM-based XSS, October 2020. URL: https://portswigger.net/web-security/cross-site-scripting/dom-based.

[27] W3, the HTML coded character set, 2020. URL: https://www.w3.org/MarkUp/html-spec/html-spec_13.html.

[28] WAPITI, 2020. URL: http://www.ict-romulus.eu/web/wapiti/home.

[29] Zap proxy mainpage, 2020. URL: `https://www.zaproxy.org/`.

[30] eval() documentation, January 2021. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval`.

[31] David Mirza Ahmad, Bruce Leidl, and David McKinney. About Vega, 2014. URL: `https://subgraph.com/vega/documentation/about-vega/index.en.html`.

[32] Kenneth Leroy Busbee. Integrated development environment. URL: `https://press.rebus.community/programmingfundamentals/chapter/integrated-development-environment/`.

[33] Steve Champeon. JavaScript: How did we get here?, July 2016. URL: `https://web.archive.org/web/20160719020828/http://archive.oreilly.com/pub/a/javascript/2001/04/06/js_history.html`.

[34] Donald Rosato Dominick Rosato. Central processing unit. URL: `https://www.sciencedirect.com/topics/engineering/central-processing-unit`.

[35] Andrew Dyster. ezyzip: Mainpage. URL: `https://www.ezyzip.com/`.

[36] Ahmed Mohamed Elhady. Complete cross-site scripting walkthrough, September 2020. URL: `www.infosec4all.tk`.

[37] Josh Fruhlinger. Malware explained: How to prevent, detect and recover from it. URL: `https://www.csoonline.com/article/3295877/what-is-malware-viruses-worms-trojans-and-beyond.html`.

[38] Daniel Garcia. GoLismero, Github page. URL: `https://github.com/golismero/golismero`.

[39] Romain Gaucher. Grabber, 2006. URL: `http://rgaucher.info/beta/grabber/`.

[40] Jeremiah Grossman, Robert Hansen, Petko D. Petkov, and Anton Rager. *XSS Attacks; Cross Site Scripting Exploits and Defense*. Amorette Pedersen, 2007.

[41] B. B. Gupta and Pooja Chaudhary. *CROSS-SITE SCRIPTING ATTACKS; Classification, Attack and Countermeasuments*. CRC Press, 2020.

[42] David Hemmendinger. Britannica: XML. URL: https://www.britannica.com/technology/XML.

[43] Álvaro Díaz Hernández. DDoS attacks through XSS, March 2015. URL: https://www.incibe-cert.es/en/blog/ddos-attacks-through-xss.

[44] Computer Hope. WSL. URL: https://www.computerhope.com/jargon/w/wsl.htm.

[45] Yiftach Keshet. Cynet, browser exploits – legitimate web surfing turned death trap, January 2020. URL: https://www.cynet.com/blog/browser-exploits-legitimate-web-surfing-turned-death-trap/.

[46] Tasos Laskos. Result page of Arachni scan, June 2014.

[47] Tasos Laskos. Arachni, readme, 2017. URL: https://rubydoc.info/github/Arachni/arachni#contributing.

[48] Tasos Laskos. Arachni is no longer maintained, January 2020. URL: https://www.arachni-scanner.com/blog/arachni-is-no-longer-maintained/.

[49] Jim Manico and Robert RSnake Hansen. OWASP, XSS filter evasion cheatsheet, September 2020. URL: https://owasp.org/www-community/xss-filter-evasion-cheatsheet.

[50] Xavi Mendez. Github Wfuzz mainpage. URL: https://github.com/xmendez/wfuzz.

[51] CWE over Time NIST. National Vulnerability Database, 2020. URL: https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time.

[52] Project Top Ten OWASP. OWASP, 2017. URL: https://owasp.org/www-project-top-ten/.

[53] Sohom Pramanick. History of Python. URL: https://www.geeksforgeeks.org/history-of-python/.

[54] Andres Riancho. W3af main page, 2013. URL: http://w3af.org/.

[55] Isaac Rodriguez. What is Pip? a guide for new pythonistas. URL: https://realpython.com/what-is-pip/.

[56] Pavitra Shankdhar. infosecinstitute, 14 popular web application vulnerability scanners, July 2020. URL: `https://resources.infosecinstitute.com/topic/14-popular-web-application-vulnerability-scanners`.

[57] Yogesh Singh. curl command in Linux with examples. URL: `https://www.geeksforgeeks.org/curl-command-in-linux-with-examples/`.

[58] Chris Sullo. Nikto introduction. URL: `https://cirt.net/nikto2-docs/introduction.html`.

[59] Kanishk Tagade. Breaking down web application scanning: Know-how and know-why, July 2020. URL: `https://blog.eccouncil.org/breaking-down-web-application-scanning-how-why/`.

[60] F. Tenzer. Marktanteile der führenden Betriebssysteme weltweit im Oktober 2020, November 2020. URL: `https://de.statista.com/statistik/daten/studie/828610/umfrage/marktanteile-der-fuehrenden-betriebssystemversionen-weltweit/`.

[61] Kevin W. Wall. OWASP Enterprise Security API (ESAPI), July 2020. URL: `https://owasp.org/www-project-enterprise-security-api/`.

[62] Tom Warren. Microsoft will bid farewell to Internet Explorer and legacy Edge in 2021. URL: `https://www.theverge.com/2020/8/17/21372487/microsoft-internet-explorer-11-support-end-365-legacy-edge`.

[63] Dominic Whits. Wikto Github page. URL: `https://github.com/sensepost/wikto`.

[64] Michal Zalewski, Niels Heinen, and Sebastian Roschke. Google Code Archive, SkipfishDoc, 2012. URL: `https://code.google.com/archive/p/skipfish/wikis/SkipfishDoc.wiki`.