Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Security and Testing Domain Specific Language for Natural History Museum

Author:
Han Wang

Supervisor:
Madasar Shah

Submitted in partial fulfillment of the requirements for the MSc degree in MSc Computing Specialism of Imperial College London

September 2019

Abstract

Web applications gained huge popularity during the last two decades. And with the machine learning become more and more prevalent in almost every industry, the security of personal data/ organizational data is no longer only security experts' concern. This paper demonstrates a Domain-specific language that I developed for users to define testing procedures with straightforward code so that the previous tedious procedure to do penetration tests is simplified and the barrier is minimized between the world of security testing and general developer/researcher.

Keywords: Domain Specific Language, Web Security, Penetration Testing, Web Security Scanner

Acknowledgments

Throughout the writing of this dissertation, I have received a tremendous amount of support and assistance. I would first like to thank my supervisor Dr. Madasar Shah, who helped me in formulating the topic, building a solid foundation to start my research, and encouraged me when I'm disheartened. Without him, this project wouldn't be possible.

I would like to thank Mr. Roger Fleuty and Mr. Chris Sleep from Natural History Museum for this opportunity to complete a project that will work in real-life production and for their collaboration and support.

I would also like to thank my colleague ukasz Niepolski and Fangyi Zhou, who have helped me in developing the website, pointing me to the right direction of building the web interface.

Ethical and Professional Considerations

This project has some slight potential for malevolent/criminal/terrorist abuse, as it is related to Penetration testing which, when performed, the scope and regulation need to be discussed and authorized. However, this project is merely implementing the tools that were already out there, so it's not adding more risk than the misuse of already existing tools. Also, since the web scanners are mostly used to find the most well-known vulnerabilities, it is not going to be a huge advancement to the "black-hats" out there even when misused.

Contents

Chapter 1

Introduction

The goal of this project is to build a DSL(Domain Specific Language) that focus on security and testing, and more specifically, is used to perform all kinds of tests (including but not limited to OWASP - Open Web Application Security Project and etc.) for web applications – the DSL will generate different artifacts according to the test that the users specified, perform the test against the target application and eventually generate a test report providing information and suggestions.

The DSLs target users are development teams at The Natural History Museum, but it can also be used by all the web application developers and quality assurance staff around the world. The DSL is to be used during the Continuous Integration stage of the development – teams and developers write tests using our DSL to write testing programs to detect problems early for each check-in. The finished project is expected to include an online web-based IDE, documentation for the language. Another high-reaching goal is that when the users of our DSL are building test cases for web application, we will also use these test cases to test against our own web application.

Currently, the testing process is tedious and intricate, and there are tons of tools out there. Without testing the effectiveness of each tool, it is very hard to tell which tool is best for the testing procedure at NHM. And setting up each tool on each machine takes a huge amount of time. The project we develop aims to simplify this process, making a single virtual machine an ”expert of security and testing” on which all the testing takes place. The project also aims to be a high-level abstraction of the testing tools so that a developer/ user without prior experience with the tools we implemented can use the language and define testing procedures in ease.

With the web-based editor and the server hosting it, Natural History Museum can deploy only one machine on which multiple testing tools are installed and have that machine do all the testings for the web applications that are on the staging server. So that when websites actually go live, it is more secure with fewer vulnerabilities.

The language is intended to be extensible and highly customizable, implementing

different kinds of tools. Therefore, when another cooperation or security experts found other tools that have not been implemented, they can simply add those tools to the language, as long as the tool is scriptable and repeatable, i.e., having a REST API.

Chapter 2

Background

2.1 Literature review

Chang Liu and Debra J. Richardson (1) presented to us why the security of systems and software is becoming a more and more important topic in the field of computer science – more and more computers are connected to the internet behind which there are unprofessional inexperienced administrator or even no administrator at all. And exploited systems does not only put their own data at risk but also threatens other large web servers as theyd become Broiler of the attacker.

Liu and Richardson (1) then proposed a framework for security checking and patching, where the securibot discovers vulnerabilities, develops patches, announces vulnerability checks security and deploy the patch in an automated way. Liu and Richardsons work gives us a brief idea of why automated testing needs our attention.

Jason Bau, Elie Bursztein, Divij Gupta, John Mitchell from Stanford University (2) has done a researching testing the effectiveness and shortage of eight existing automated testing tools. They came with the conclusion that the scanners' tests are most extensively Cross Site Scripting, SQL injections, and Cross Channel Scripting, and they are great at finding known vulnerabilities as we should expect. However, most of them are not good at following links through active content technology like Flash and Java applets, and they are also not good at finding stored vulnerability.

In order to find vulnerability more effectively to protect the servers and systems which had become valuable assets of companies and individuals, Rohan Vibhandik and Arijit Kuma Bose proposed a new way of assessing vulnerabilities (3), that is to use a combination of security testing tools. And in their paper, they presented their researching regarding top 10 OWASP based threat modelling of web Applications. They performed different test using different parameters such as open-source or freeware, TSL/SSL, HTTP proxy and etc. using a combination of three tools, Nikto, PenTest Tool and W3AF and came with the conclusion that the combination of such

tools can provide most comprehensive and stable coverage of possible vulnerability. A DSL specialized in security and testing will come in handy to make this process easier in practice.

Another aspect, Test Driven Development(TDD), is software development practice that completes short repetitive circles - first developer/team writes automated test cases that define an improvement or addition of a method, then developer/team try to make such improvement/ add such function to pass the test in order to complete the task. TDD has gained large popularity over the decades since it was first introduced in the 1990s because of the effectiveness and code stability it provided.

Selenium as a well-known testing library is a great tool for TDD. It is cross-platform tool that brings lots of benefits, according to Xinchun Wang and Peijie Xu (4), first, it can be used from the perspective of end users to test applications through the Selenium testing script; second, its easier to locate a browsers incompatibilities by running tests in a different browser. However, there are shortages as well, for example, the Selenium scripts are usually hard to read and maintain. And this is where domain specific language (DSL) comes in handy, it can significantly reduce the number of lines of code testers/developers have to write and it improves readability and maintainability of the code. Introduced in their paper Build an Auto Testing Framework Based on Selenium and FitNesse, in their framework, Xinchun Wang and Peijie Xu has actually built a DSL to use FitNesse fixtures more effectively. The DSL is divided into two types in their framework, one is a single-line table and the other is a multi-line table where DSL comes in handy – that is when testers/developers would want to perform the same test repetitively using different inputs.

2.2 Choice of language workbench

To develop a DSL, we need a language workbench that are tools/frameworks we use to define the grammar, code generation, validations and so on. There are three most popular language workbench – Xtext, Spoofax, and Jetbrains MPS.

Out of the three, Xtext is the most popular choice and therefore most stable and documented, it is parser-based and uses ANTLR internally which is a parser generator that uses LL(*) for parsing. Spoofax is the least popular amongst the three and its also parser-based, however, it uses its own parser system which makes combination of languages easier. MPS is projection-based therefore different than the other two, however, it gives most freedom to language designers and combination of language is also easy.

Considering that the popularity and rich documentation and knowing that combination of languages is not needed in our project, we decided to choose Xtext as our language workbench.

Xtext is an open-source Eclipse framework for implementing Domain Specific Language together with the integration with Eclipse IDE itself. Xtext allows its users to develop a DSL quickly by covering aspects like parser, generator, IDE integration with Eclipse, web IDE and etc. Each component of xtext is highly custimizable, therefore the users of xtext can make the decision of either adopt the default settings, or implementing more advanced concepts as needed.

The community of xtext is also pretty friendly. Based on observation, most of posts on xtext's forum is responded within two days, despite the fact it is a relatively minor area within computer science field compared to machine learning and data science.

2.3 OWASP Zed Attack Proxy

OWASP ZAP is a cross-platform free and open-source penetration test tool for web applications that the users have permission for and its an OWASP flagship project. ZAP is an intercepting Proxy, which means users should configure their browser to proxy through ZAP for ZAP to see all the requests and responses, and ZAP can intercept and change them. ZAP also include passive scanners that are safe to use on all sites as it does not perform any sort of attacks and active scanners that are used on applications that user have permission to test. Spider is another feature of ZAP that allows users to crawl the web application to find pages that users have missed or those that have been hidden. Other features of ZAP includes Report Generation that gives users suggestions and information regarding their test results and Brute Force tools that can be used to find files with no links to them and etc.

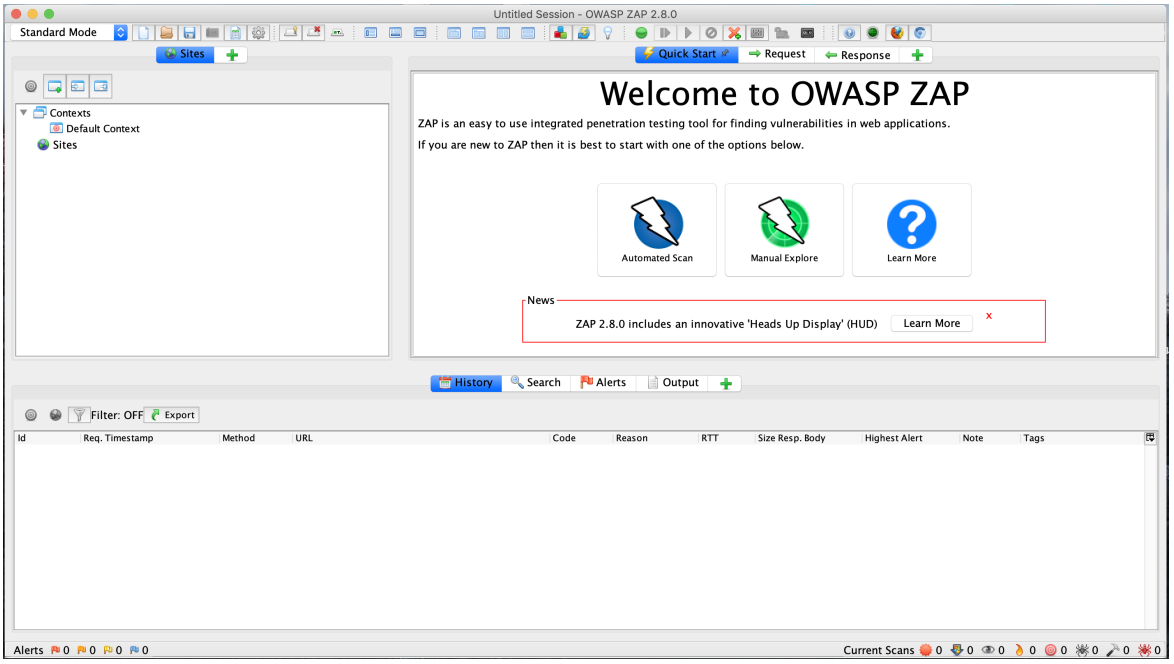


Figure 2.1: Zed Attack Proxy GUI

Figure 5.15 is the GUI of ZAP.

There are a couple of reasons why I chose to implement ZAP in my language. The most important reason is that Zap is an OWASP flagship project, and it is well-maintained.

When I was doing research on tools for application security testing, I found that lots of them are no longer maintained, or are completely abandoned by the author, like a famous tools BOON (https://people.eecs.berkeley.edu/ daw/boon/), was abandoned by its author, rendered the tool unsupported and unusable. Interestingly, the

other tool we have implemented in the language, called w3af, has some components that were less used, therefore undermaintained, and during my development of the project, I have contributed to a bug fix of the w3af project which I will discuss further in the session where w3af was introduced.

Another reason why we chose ZAP is that it provides a REST API (Representational State Transfer Application programming interface) which makes the function programmatically accessible. Official python/java clients were also developed to make the testing process using ZAP highly scriptable, which is exactly what we needed for the language we are trying to develop. ZAP also provides a web-based GUI which is exactly same as GUI of the system-dependent software, making Orchestration testing possible which I have considered doing but later decided to change to API client manipulation.

2.4 Web App Attack and Audit Framework

W3AF, which is short for Web application attack and audit framework, is a popular, flexible and powerful tool aiming to find and exploit web application vulnerabilities. W3af was started by Andres Riancho in 2007 and after several years of developing and receiving contributions from the community, it was sponsored by and partnered with Rapid7 which is the company developed one of the most famous vulnerability testing tool for systems – Metasploit Project.

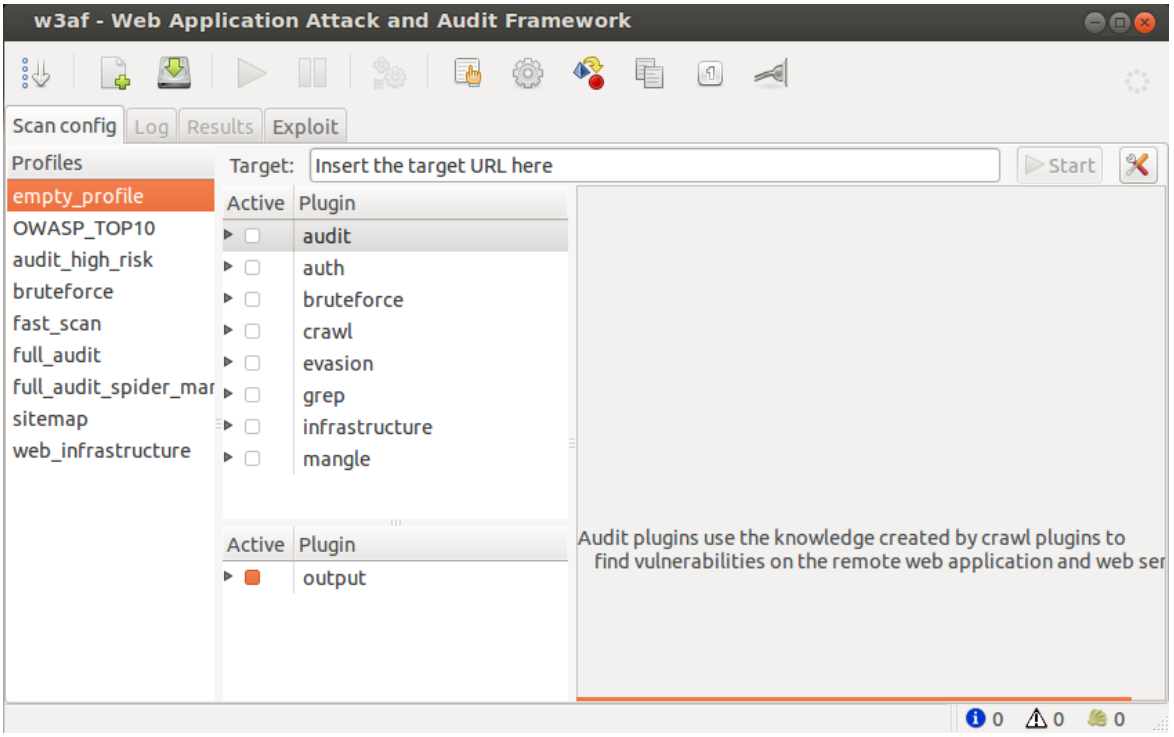


Figure 2.2: w3af GUI

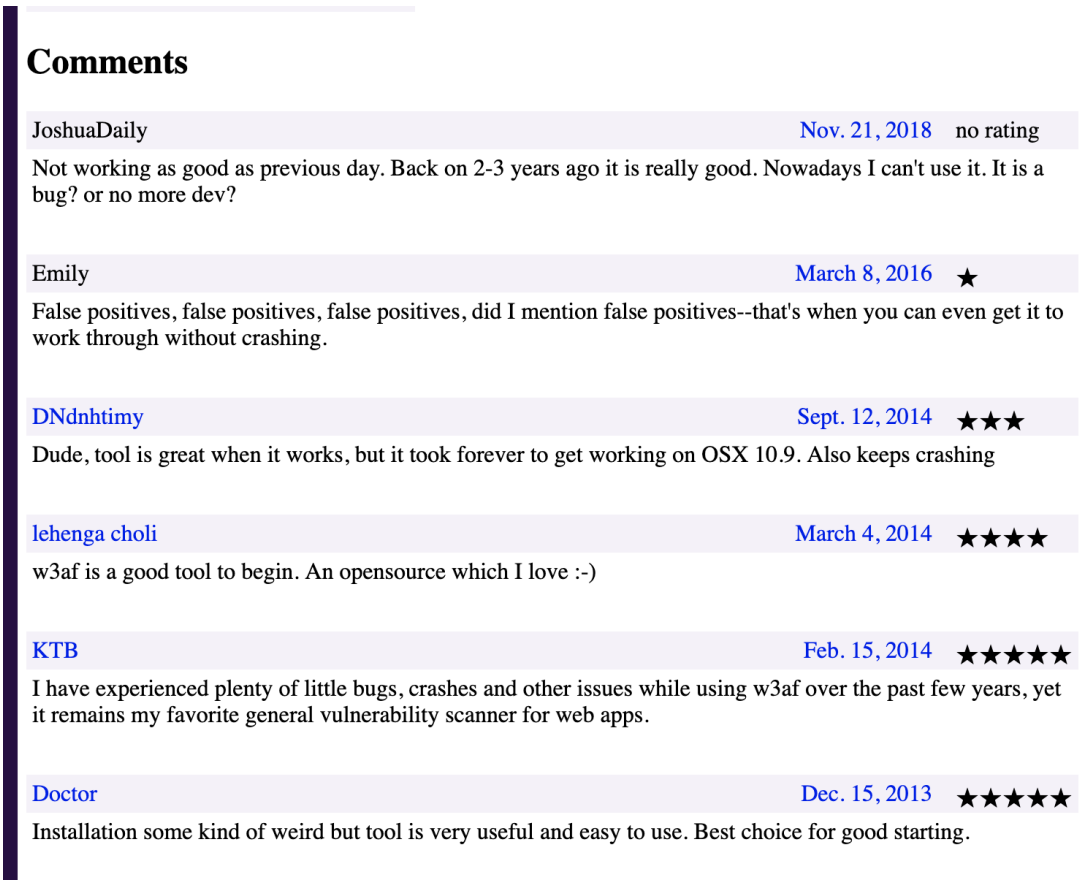
W3af is divided into two parts - core and plugins, both of those are written in python (2.x).

The core coordinates the process and provides features that are consumed by the plug-ins, which find the vulnerabilities and exploit them. The plug-ins are connected and share information with each other using a knowledge base.[wiki]

And the plugins are divided into eight categories, Discovery, Audit, Grep, Attack, Output, Mangle, Evasion or Bruteforce, and the three main plugins are crawl, audit and attack. The crawl plugins are the starting point of them all, it crawls the target URL, finding new URL, forms and other injection points [w3af docs]. The audit plugins then carefully go through the lists found by crawl plugins, send purposefully generated data into those injections to identify vulnerabilities. And the attack

plugins will actually try to exploit the vulnerabilities that were found by the audit plugins, trying to return a shell on the remote server, or dump a remote table if its trying to exploit a SQL injection.

Despite being constantly rated as one of the best/most popular tools for security testing tools, w3af is pretty controversial. The cons that are most criticised are its compatible issues, and its false positive rate. Personally, when I was trying to setup w3af on my laptop, I first wanted to use the API w3af provided, so I tried to pip install w3af-api-client, however, it wasnt successful because of a syntax error! I have created a pull request for this issue and it was fixed after the author approved my pull request, merging it into the master branch. Other compatibility issues are more platform/ environment-dependent, for example, quoting a comment by DNdnhtimy, on sectools.org, "tool is great when it works, but it took forever to get working on OSX 10.9. Also keeps crashing". In reality, when I was setting up the tools locally and remotely on the virtual machine that Natural History Museum provided me, I have encountered tons of error since the tool was written in python 2.x and some of the grammar was no longer valid in python 3.x, and other dependency problems as the tool requires some library written in C but cannot handle that using pip install since it was, again, written in Python.



Comments

JoshuaDaily Nov. 21, 2018 no rating

Not working as good as previous day. Back on 2-3 years ago it is really good. Nowadays I can't use it. It is a bug? or no more dev?

Emily March 8, 2016 ★

False positives, false positives, false positives, did I mention false positives--that's when you can even get it to work through without crashing.

DNdnhtimy Sept. 12, 2014 ★★★

Dude, tool is great when it works, but it took forever to get working on OSX 10.9. Also keeps crashing

lehenga choli March 4, 2014 ★★★★

w3af is a good tool to begin. An opensource which I love :-)

KTB Feb. 15, 2014 ★★★★★

I have experienced plenty of little bugs, crashes and other issues while using w3af over the past few years, yet it remains my favorite general vulnerability scanner for web apps.

Doctor Dec. 15, 2013 ★★★★★

Installation some kind of weird but tool is very useful and easy to use. Best choice for good starting.

Figure 2.3: Comments on w3af from sectools.org

Chapter 3

Development of the Language

I have referred to Lorenzo Bettini's book (6) Implementing Domain-Specific Language with Xtext and Xtend throughout the development stage and also referred to Martini Fowler's book (5) Domain Specific Languages for several components of the language.

For the non-language related part of the project, Dr. Patrick Engebretson's book (7) had been a great introductory material for me. All of the three books have great content in terms of scope and depth and are friendly to readers that are new to these fields. They will be referred back to as further development are conducted.

3.1 Syntax

The first thing I modified after the project was created is "ITL.xtext." which is the main file that defines the model and grammar of the Domain specific language.

*ITL.xtext ITLGenerator.xt ServerLauncher. ItlServiceDispa »9

```
1  grammar ic.ac.uk.itl.ITL with org.eclipse.xtext.common.Terminals
2
3  generate iTL "http://www.ac.ic/uk/itl/ITL"
4
5  Model:
6      testing_tools += testing_tool*;
7
8  testing_tool:
9      Spider | W3af
10 ;
11
```

Figure 3.1: Main model of the language

The first line here declares the name of the language, which is also the extension name of the file written in this language, ITL, which is an abbreviation for Imperial Testing Language. And this line also indicates that it inherit grammar from Terminals which defines the basic grammar rules, i.e. single/double quotes for strings.

The second line declares the generation rules for EMF(Eclipse Modeling Framework), which will be used when generating xtext artifacts.

Then it comes to the main model of our language. The first rule tells the parser where to begin and the type of the root element of the model in my language, the AST(abstract syntax tree). In my language, I declared that an ITL program is a combination of different "testing_tool" elements. And it is stored in a Model type object, more specifically, a Model type object that is named testing_tools. The "+=" operator means that it is a combination and the * operates indicates that there can be an arbitrary number of elements (greater or equal to 0), which means a void file that contains nothing is also a valid ITL program.

There are two types of testing_tool I choose to implement, ZAP and W3AF which is also indicated on the 9th line here.

```
12 ZAP:
13     'ZAP' name=ID '{'
14     zap_target= ZAP_TARGET ';'
15     zap_address=ZAP_ADDRESS ';'
16     zap_max_depth=ZAP_MAX_DEPTH ';'
17     zap_api_key=ZAP_API_KEY (';'?) '}';
18
19
20 ZAP_ADDRESS:
21     'ZAP_ADDRESS:' name=STRING;
22 ZAP_MAX_DEPTH:
23     'ZAP_MAX_DEPTH:' name=INT;
24 ZAP_API_KEY:
25     'ZAP_API_KEY:' name=STRING;
26 ZAP_TARGET:
27     'ZAP_TARGET:' name=STRING;
28
```

Figure 3.2: Zed Attack Proxy Model

ZAP and W3AF each has its own shape, expressed in the rule defining them. A ZAP testing_tool contains four attributes, zap_target, zap_address, zap_max _depth and zap_api_key. This ZAP tool we implemented is supposed to generate a python script that uses ZAPs API to do a crawling of the target website, first spidering the target URL, then performing passive scans followed by active scans. So the attribute zap_target defines the target on which the users want to perform the scanning; the attribute zap_address defines the location where the report will be generated; the attribute zap_max _depth sets the maximum depth that the spider can crawl, 0 means unlimited and finally zap_api_key defines which zap instance the script will be running on, as different zap instance will be assigned different API keys upon instantiation.

```
29 W3af:
30     'W3af' name =ID
31     '{' w3af_test_type=W3AF_TEST_TYPE';'
32      w3af_address=W3AF_ADDRESS';'
33      w3af_report_path=W3AF_REPORT_ADDRESS';'
34       w3af_target=W3AF_TARGET (';'?)'}';
35
36 W3AF_REPORT_ADDRESS:
37     'W3AF_REPORT_ADDRESS:' name=STRING;
38 W3AF_ADDRESS:
39     'W3AF_ADDRESS:' name=STRING;
40 W3AF_TEST_TYPE:
41     'W3AF_TEST_TYPE:' name=STRING;
42 W3AF_TARGET:
43     'W3AF_TARGET:' name=STRING;
44
```

Figure 3.3: w3af Model

Similarly, a W3af testing_tool contains four attributes as well, w3af_test_type, w3af_address, w3af_report_path and w3af_target. Note that the code snippet that begins with w3af generates a script with extension name .w3af which is the script that can be run by a w3af instance. We will talk more about those artifacts in the later session. The first attribute w3af_target defines the target on which the users want

to perform the scanning using w3af; the attribute w3af_address defines the location where w3af is located (note this is different from zap_address where it specifies the location of generated report; the attribute w3af_test_type sets the type of tests that will be performed(crawl, audit and attack, which is the three main modules of the w3af plugins) and finally w3af_report_path defines where the report will be generated.

Note that if the user of ITL language only wants to generate the python/ w3af script and run them locally on their machine (that is they have already installed either or both of the tools locally), it is not necessary for them to specify w3af_address, since it is only used to locate w3af instance on the virtual machine that NHM provided me (or if the users want to set up their own virtual environment in which the w3af instances are running).

3.2 Generator

After the grammar of the language is determined, the next mission will be implementing the generator for the language. Xtext has provided nice IDE for us to write ITL programs now after the grammar is complete and it already comes with syntax highlighting and instant error feedback, but the code doesnt do anything yet – they are just beautiful code blocks. To make the language actually useful, we have to implement the generator of the language.

The generator is implemented in the file called ITLGenerator.xtend shown below.

The doGenerator function first divides the content of user code into two parts, ZAP and w3af. When doing generations for ”ZAP” code snippet, all the code snippet started with ”w3af” will be filtered out, and similarly, code snippet started with ”ZAP” will be filtered out when ”w3af” codes are concerned.

3.2.1 ZAP Generator

For a ZAP code snippet, the generation process is pretty straight forward. For the python script that we want to generate, first of all, the target and api key was specified which is the same string that came after ZAP_TARGET: and ZAP_API_KEY: in the ITL program.

The code in the generator works like this, it first imports the class that was generated by EMF, taken each of its fields, and if the field is an object we defined, it will be imported as well. Then, when generating the desired scripts based on users code, it is done by calling the method with signature compile(Object obj) which is an overloaded function that can take different object as input, and outputs a string paragraph (the generated script). Note that inside the compile function, only the code inside will be interpreted, everything else will be treated as strings and put into the final string paragraph as output.

ITL.xtext | ITLGenerator.xtend | ITLServlet.xtend | GenerateITL.mwe2 | ic.ac.uk.itl.ide | ic.ac.uk.itl.ui | style.css | ItlServiceDispatcher.xtend

```
def compile(ZAP spider) {
    '''
    #!/usr/bin/env python
    # A basic ZAP Python API example which spiders and scans a target URL

    import time
    from pprint import pprint
    from zapv2 import ZAPv2
    # use pip install python-owasp-zap-v2.4 to install python api client in order to run this script

    target = "«spider.zap_target.name»"
    apikey = "«spider.zap_api_key.name»" # Change to match the API key set in ZAP, or use None if the API key is disabled
    #
    # By default ZAP API client will connect to port 8080
    zap = ZAPv2(apikey=apikey)
    # Use the line below if ZAP is not listening on port 8080, for example, if listening on port 8090
    #zap = ZAPv2(apikey=apikey, proxies={'http': target+':'+str(8080), 'https': target+':'+str(8080)})

    # Proxy a request to the target so that ZAP has something to deal with
    print('Accessing target {}'.format(target))
    zap.urlopen(target)
    # Give the sites tree a chance to get updated
    time.sleep(2)

    print('Spidering target {}'.format(target))
    zap.spider.set_option_max_depth(«spider.zap_max_depth.name»)
    scanid = zap.spider.scan(target)

    # Give the Spider a chance to start
    time.sleep(2)
    while (int(zap.spider.status(scanid)) < 100):
        # Loop until the spider has finished
        print('Spider progress %: {}'.format(zap.spider.status(scanid)))
        time.sleep(2)

    print ('Spider completed')

    while (int(zap.pscan.records_to_scan) > 0):
          print ('Records to passive scan : {}'.format(zap.pscan.records_to_scan))
          time.sleep(2)

    print ('Passive Scan completed')

    print ('Active Scanning target {}'.format(target))
    scanid = zap.ascan.scan(target)
    while (int(zap.ascan.status(scanid)) < 100):
        # Loop until the scanner has finished
        print ('Scan progress %: {}'.format(zap.ascan.status(scanid)))
        time.sleep(5)
```

Figure 3.4: ZAP generator

After the target and api key are set, a zap instance will be instantiated with the api key, then the target url will be set using zaps method urlopen. Once the target is set, the spider is ready to crawl after specifying the max depth allowed, which is the parameter taken from user code after ZAP_MAX_DEPTH. The passive scan will start automatically after the spider is done crawling, inspecting all records that the spider collects.

The passive scan will be followed by an active one, which is started by calling the method zap.ascan.scan(target), and this will usually take some time since it actually tries to exploit the vulnerabilities that were found by passive scan previously.

After crawling and both scans are done, the report will be generated and returned to the console and that completes the script we generated for each Zap code snippet.

3.2.2 w3af Generator

The generation process of w3af is much more straight forward as the .w3af file contains exactly the command you would type in when running w3af's console. The attribute test _type will define which plugins of w3af will be enabled. For instance, if we specify the test_type to be "crawl, audit", then in the script, after the line "plugins", a couple of lines of code shown below will be added.

```
12
13  crawl web_spider
14  crawl config web_spider
15  set only_forward true
16  back
17  crawl
18
19  audit all
20  audit
```

Figure 3.5: w3af Generator

"crawl web_spider" is to enable a plugin called "web spider" within crawl category. I have explicitly chosen this specific plugin as this is the plugin that was most stable, and efficient one during my experiment, the other crawl plugins either took too long to run or have unstable behaviour.

"crawl config web_spider" is to configure the parameters of this plugin. And for the staging server of Natural History Museum, we are setting the spider to only crawl forward, so that it doesn't crawl the whole sites when we want it to only crawl a single web application.

Similarly, "audit all" here means all the audit plugins are enabled for this scan.

3.3 Artifact

Two different types of scripts will be generated based on user's code, python scripts and w3af scripts. As we mentioned earlier, the code snippet begins with ZAP will generate python scripts and that begins with "w3af" will generate w3af scripts. Alongside those scripts that will be running to do the testing, with each python/w3af scripts generated, a bash script containing the command that will be used to run each script will also be generated and the bash script will be associated with the run button on the web editor.

3.4 Web Editor

The web editor is one of the most important components in this project, as it provides direct user interactions and it presents the language's components in a straightforward way.

The basic syntax highlighting, error feed-backs functions are already implemented in the .js file when the xtext artifact was generated (after we defined the grammar). The generate button, download button and run buttons are needed so that user can see the generated scripts or run the tests directly on the virtual machine.

To do so, I first implemented a service dispatcher to handle such requests. For different type of requests, it will call different services. For example, if 'generate' was requested, getCompileService will be called, inside which the injected generatorService will be called and return a document type as a result.

ITL.xtext | ITLGenerator.xt | ITLServlet.xten | GenerateITL.mwe | ic.ac.uk.itl.ui | ItlServiceDispa

```
1  package ic.ac.uk.itl.web
2
3  import com.google.inject.Inject
12
13 class ItlServiceDispatcher extends XtextServiceDispatcher {
14
15     @Inject IResourceBaseProvider resourceBaseProvider
16
17     @Inject
18     GeneratorService generatorService;
19
20     override protected createServiceDescriptor(String serviceType, IServiceContext context) {
21         switch serviceType {
22
23             case 'generate':
24                 getCompileService(context)
25
26             default:
27                 super.createServiceDescriptor(serviceType, context)
28         }
29     }
30
```

Figure 3.6: Service Dispatcher Class

After the service dispatcher is in position, I implemented the generate button on the frontend. I modified the html file under WebRoot folder, added a div where the result of generation will be shown and added a generate button.

For the frontend, a download function is created, which takes two parameters, filename and content. The download function will create a blob and provide a reference to this blob, then click on it to start downloading.

Then I bound an inline function to generate button's onclick event using JQuery.bind(). First, we call fetch function to get the result document returned by generator service, which is a json contains multiple artifacts objects (python/w3af scripts). Each object has two fields – name which is the path of the artifact followed by the name of it

and its content. I sliced their name to only keep the file name and stored them in an array and store their content in another array.

For each artifact, a download button will be dynamically generated by calling document.createElement(”BUTTON”), and the button will be assigned innerHTML value corresponding to the artifact’s name and an event listener which calls the download() function with parameter artifact’s name and its content.

index.html — ~/eclipse-xtext-itl/ic.ac.uk.itl.parent/ic.ac.

index.html

```
32    var editor = xtext.createEditor();
33      jQuery('#generate-button').bind("click", function(e){
34      fetch('http://' + location.host + '/xtext-service/generate?resource='+
35      editor.xtextServices.options.resourceId).
36      then(function(response){
37        response.json().then(function(data){
38          var numOfScript = Object.keys(data.artifacts).length;
39          console.log(numOfScript);
40          var lastIndex;
41          for (i = 0; i < numOfScript; i++) {
42
43            lastIndex = data.artifacts[i].name.lastIndexOf('/');
44            artifactName.push(data.artifacts[i].name.slice(lastIndex+1));
45            artifactContent.push(data.artifacts[i].content);
46            btn.push(document.createElement("BUTTON"));
47            btn[i].innerHTML = "Download " + artifactName[i];
48            btn[i].addEventListener('click', download.bind(this, artifactContent[i], artif
49
50            run_btn.push(document.createElement("BUTTON"));
51            run_btn[i].innerHTML = "Run " + artifactName[i];
52
53            console.log("artifactContent[i]",artifactContent[i]);
54            document.getElementById("buttons-div").appendChild(btn[i]);
55            document.getElementById("buttons-div").appendChild(run_btn[i]);
56            lastIndex=0;
57          }
58
59        })
60      }).catch(function(e){
61        console.log('error: ' + e.toString());
62      })
63      jQuery('#generator-result').html('<iframe src="http://' +
```

Figure 3.7: index.html

```
92  <script>
93    function download(text, filename){
94      console.log(text);
95      console.log(filename);
96      var blob = new Blob([text], {type: "text/plain"});
97      var url = window.URL.createObjectURL(blob);
98      var a = document.createElement("a");
99      a.href = url;
100     a.download = filename;
101     a.click();
102   }
103
104
105 </script>
```

Figure 3.8: index.html

Chapter 4

Architecture

4.1 Virtual Machine Setup

To let the Natural History Museums developers and staff test their web applications more efficiently, I have designed the architecture with help from the Natural History Museums Data service Manager Roger Fleuty. A virtual machine named web-zap-1 that comes with CentOS operating system within Natural History Museums network was provided to me.

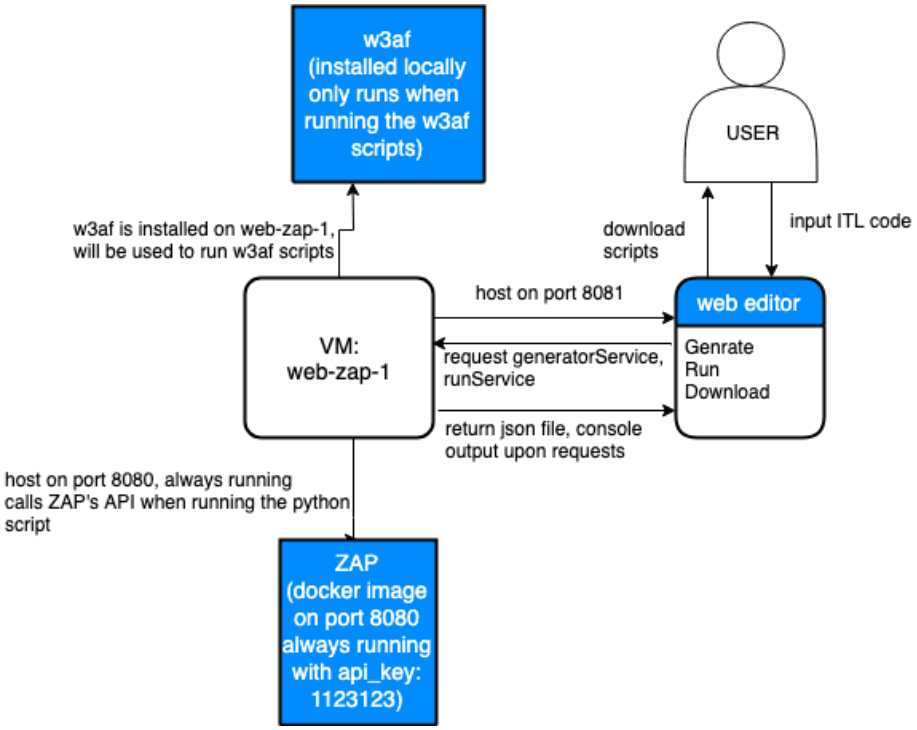


Figure 4.1: architecture

I have set up a tomcat server on the virtual machine (using homebrew) to host the main web-based IDE for the language, where user can write an ITL program and this is hosted on port 8081. On the virtual machine, there is also a ZAP instance running inside a docker container, which listens on port 8080, with api-key 1123123.

```
[hanw2@web-zap-1 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
  STATUS              PORTS               NAMES
[hanw2@web-zap-1 ~]$ docker run -u zap -p 8080:8080 -i owasp/zap2docker-stable
 zap.sh -daemon -host 0.0.0.0 -port 8080 -config api.addrs.addr.name=.* -confi
g api.addrs.addr.regex=true -config api.key=1123123
Found Java version 1.8.0_212
Available memory: 3789 MB
Using JVM args: -Xmx947m
```

Figure 4.2: the command line used to run ZAP

After the ITL program is completed, users can click on the generation button on the right, and the whole generation process takes place on the backend, i.e., the virtual machine named web-zap-1. The generator on the back end would generate the artifacts (python and w3af scripts) corresponding to users code, on the location that users specify and then a json will be returned back to the front end containing the artifacts name(including path), and content.

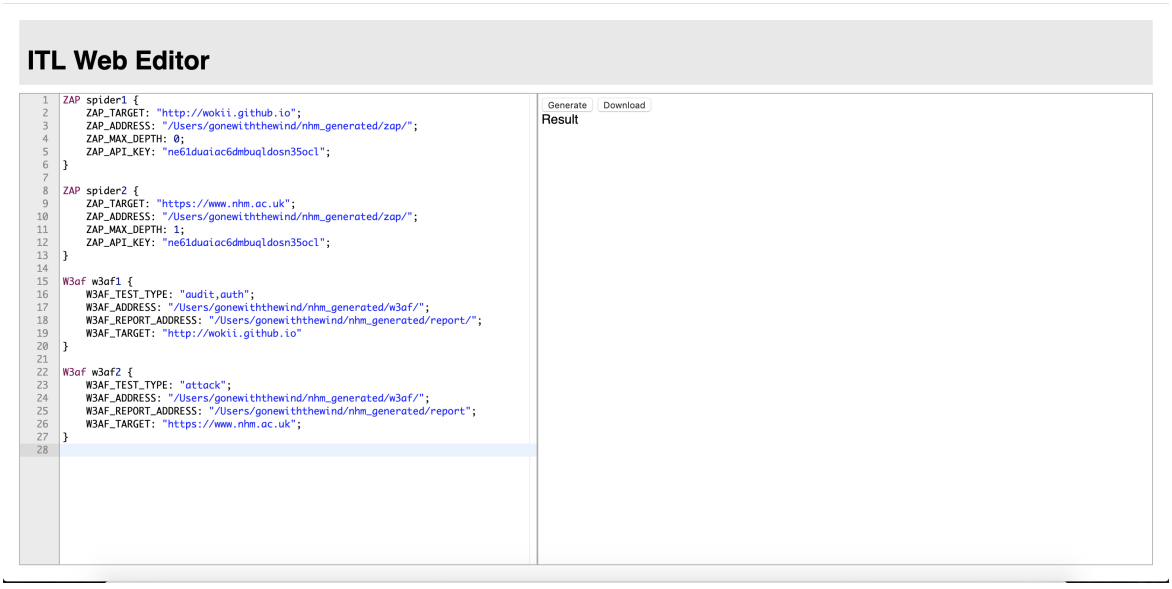


Figure 4.3: web-based editor

The frontend will fetch the json response and parse it, save the parsed result into an array of objects. Then an array of blobs named after each script and containing the content of each script will be created. Each blob will dynamically create a download button, with event listener associated with them, calling a download function with

parameter scripts name and scripts content. By clicking on them, users can download corresponding python or w3af scripts.

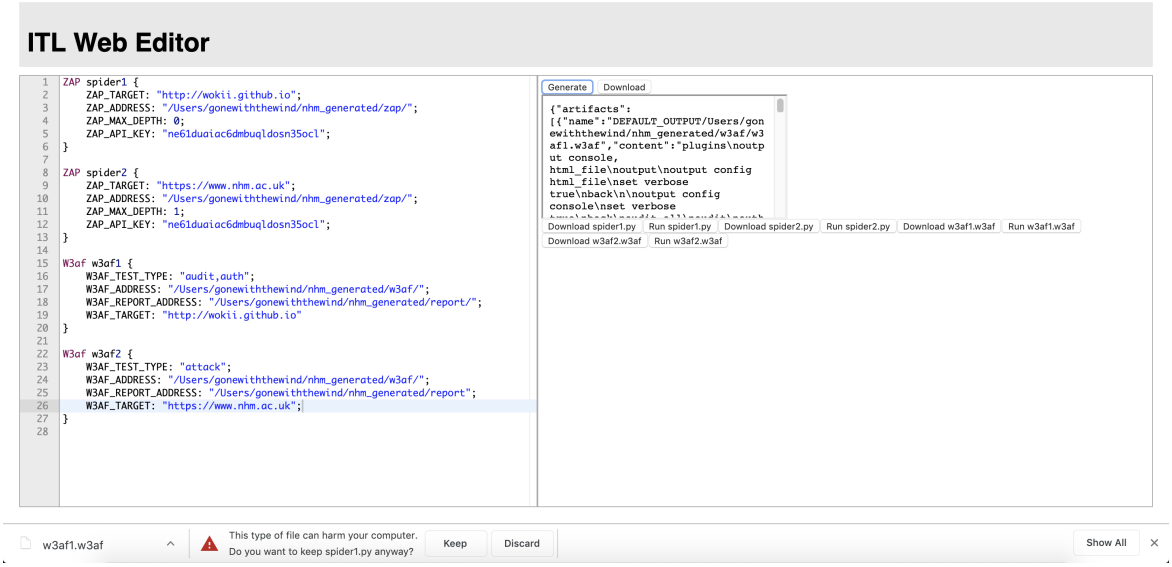


Figure 4.4: web editor after clicking generate button

Run buttons will also be dynamically generated when fetching the json response from backend server. Each run button will run the correspondent script. When running the python script, the command line being used are “python3 /path/to/script” and when running the w3af script, the command line will run “/path/to/w3af/w3af_console -s /path/to/w3afscript”.

4.2 Continuous Integration Pipeline

wokii Update Jenkinsfile 40f8e42 on Jul 23

1 contributor

45 lines (43 sloc) 1.07 KB Raw Blame History

```
pipeline {
    agent {
        docker {
            image 'maven:3'
        }
    }
    stages {
        stage('Build') {
            steps {
                sh 'ls'
                dir("org.xtext.nhmproject.parent") {
                    sh 'ls'
                    sh 'java -version'
                    sh 'mvn -v'
                    sh 'mvn clean verify'
                }
            }
        }
        stage('Test') {
            steps {
                dir("org.xtext.nhmproject.parent") {
                    sh 'mvn test'
                }
            }
            post {
                always {
                    dir("org.xtext.nhmproject.parent") {
                        sh 'ls'

                        dir("org.xtext.nhmproject.tests") {
                            sh 'ls'
                            junit 'target/surefire-reports/*.xml'
                        }
                    }
                }
            }
        }
        stage('Deliver') {
            steps {
                echo 'dummy delivery'
            }
        }
    }
}
```

Figure 4.5: JenkinsFile in github repo

When developing the project, I deployed a Jenkins site on a virtual machine that I deployed on digitalOcean.

The Jenkins server is connected to my GitHub repository, so that every time a commit is pushed, the server will start building automatically. In side a maven3 docker container, ”mvn clean verify” was run to generate all the .jar and .war files and run all the JUnit tests. The .war can theoretically be copied automatically to the remote server to deploy the web-based editor, using ”-scp” command. However, the virtual machine provided by NHM is on the intranet and can only be ssh’ed thru a bastion server, during which a google authenticator code has to be obtained manually from my phone. I was not able to find a way to bypass this, thus I only save the .war file locally on the Jenkins server.

Chapter 5

Evaluation

For the evaluation of the project, I have setup two experiments.

1. Write ITL program to test against http://www.insecurelabs.org/, which is intentionally made vulnerable for cross scripting attacks
2. Test against the endpoints on NHM's staging server that Mr Fleuty provided.

5.1 InsecureLabs.org

If we inspect the website manually, we will notice that under the agenda tab, there is a search box. A search box that is implemented without cautious is usually vulnerable for Cross Site Scripting and if we type "<script >alert('XSS!')</script >" into the search box, after pressing "Enter", we can immediately see an alert popping saying "XSS".

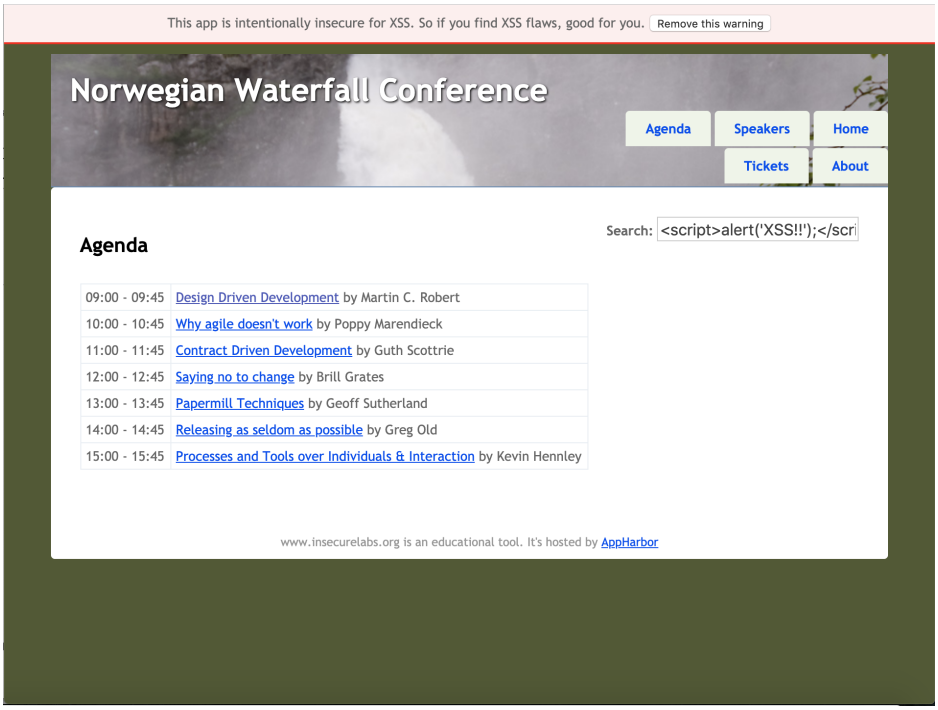


Figure 5.1: manual inspection 1

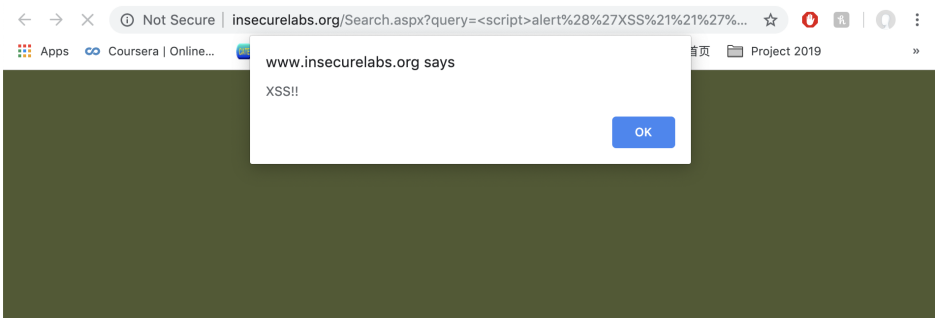


Figure 5.2: manual inspection 2

First, we use ssh tunnelling to editor website hosted on the virtual machine onto our localhost:8081 (we need to use ssh tunnelling because the virtual machine is on the intranet of Natural History Museum when we are outside of the museum I have to ssh thru their bastion server). Then I wrote the below ITL program.

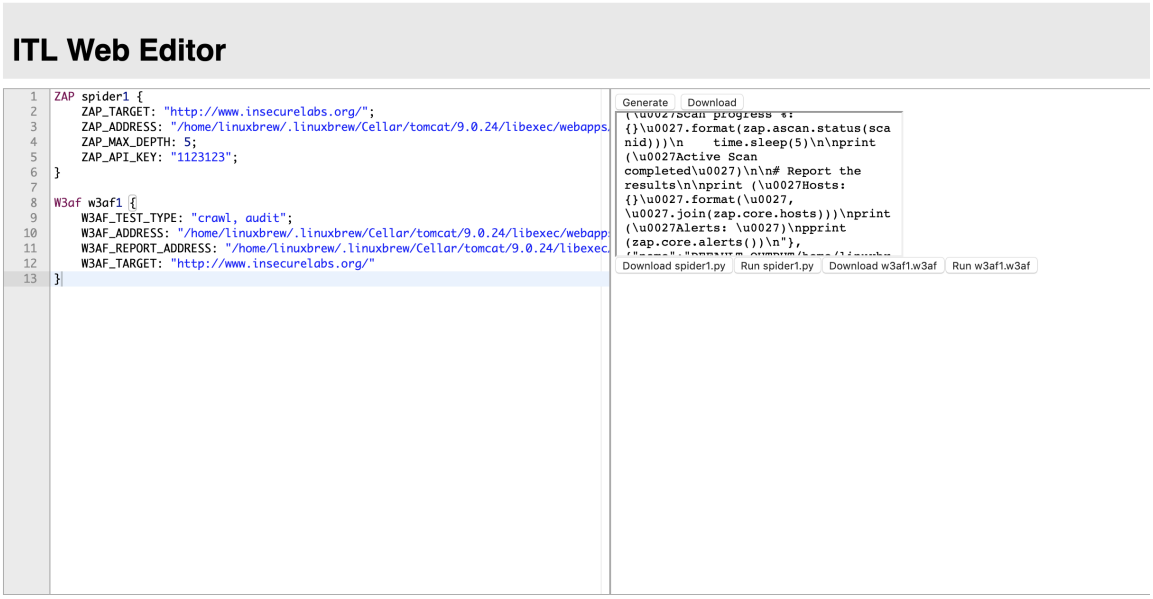


Figure 5.3: ITL program for InsecureLabs.org

We can download the scripts and see the contents of each.

5.1.1 ZAP Performance

The python script that is used to call the API of Zed Attack Proxy to perform crawling, passive scan and active scan are attached below.

```
#!/usr/bin/env python
# A basic ZAP Python API example which spiders and scans a target URL

import time
from pprint import pprint
from zapv2 import ZAPv2
# use pip install python-owasp-zap-v2.4 to install python api client in
#order to run this script

target = "http://www.insecurelabs.org/"
apikey = "1123123"
# Change to match the API key set in ZAP, or use
# None if the API key is disabled
#
# By default ZAP API client will connect to port 8080
zap = ZAPv2(apikey=apikey)
# Use the line below if ZAP is not listening on port 8080,
# for example, if listening on port 8090
#zap = ZAPv2(apikey=apikey, proxies=
# {'http': target+':'+str(8080), 'https': target+':'+str(8080)})

# Proxy a request to the target so that ZAP has something to deal with
print('Accessing target {}'.format(target))
zap.urlopen(target)
# Give the sites tree a chance to get updated
time.sleep(2)

print('Spidering target {}'.format(target))
zap.spider.set_option_max_depth(5)
scanid = zap.spider.scan(target)

# Give the Spider a chance to start
time.sleep(2)
while (int(zap.spider.status(scanid)) < 100):
    # Loop until the spider has finished
    print('Spider progress %:
    {}'.format(zap.spider.status(scanid)))
    time.sleep(2)

print ('Spider completed')

while (int(zap.pscan.records_to_scan) > 0):
      print ('Records to passive scan : {}'.format(zap.pscan.records_to_
      time.sleep(2)
```

```
print ('Passive Scan completed')

print ('Active Scanning target {}'.format(target))
scanid = zap.ascan.scan(target)
while (int(zap.ascan.status(scanid)) < 100):
    # Loop until the scanner has finished
    print ('Scan progress %: {}'.format(zap.ascan.status(scanid)))
    time.sleep(5)

print ('Active Scan completed')

# Report the results

print ('Hosts: {}'.format(', '.join(zap.core.hosts)))
print ('Alerts: ')
pprint (zap.core.alerts())
```

We can either download this file and run it locally or click the run button to run it automatically on the server. Here we download it and add a couple of more lines of code so that it generates a nice report with json format. The code we add at the end of the file is shown below.

```
import json
with open('zap_report.json', 'w') as f:
    json.dump(zap.core.alerts(), f)
```

Then the script was run by using the command

```
$ python3 ~/Downloads/spider1.py
```

After the scans are completed, feedback is output to the console using ”pprint”, in the format of list of dictionaries. It successfully detected the cross site scripting vulnerability as expected.

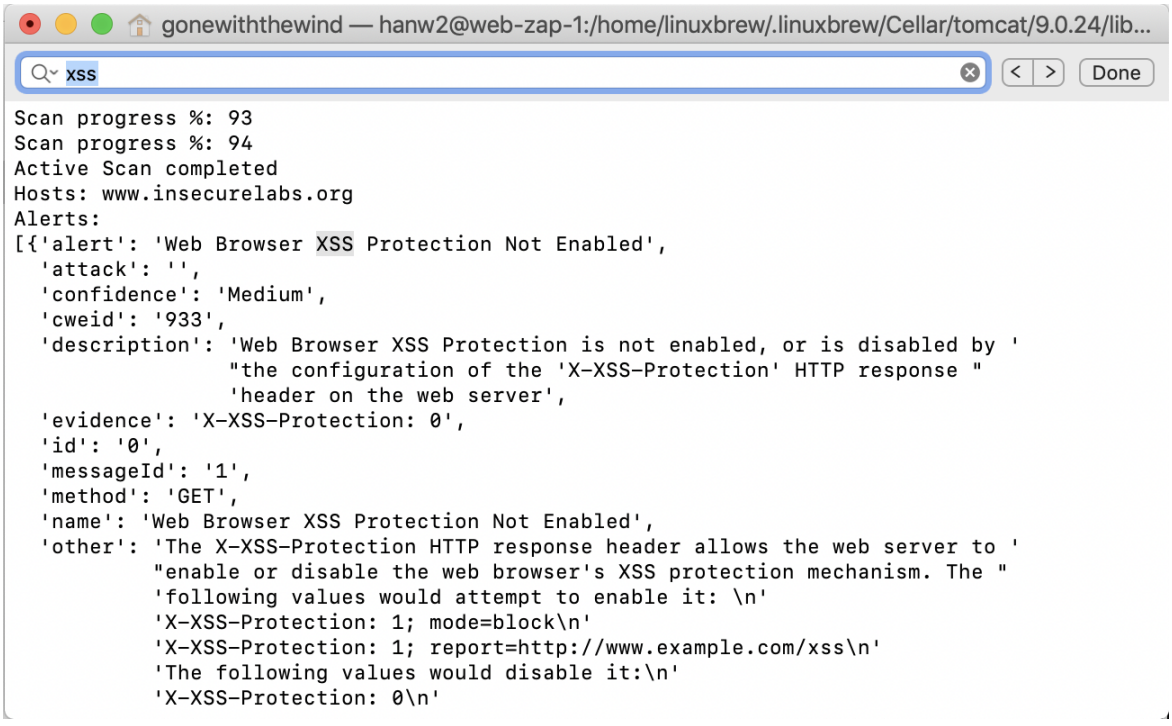


Figure 5.4: ITL program for InsecureLabs.org

The whole scan process took 343.732s to complete and raised 21 alerts, according to the logs inside ZAP’s docker container.

5.1.2 w3af Performance

On the other hand, w3af script looks much more concise, because it is exactly what you would type in a console in which w3af is running. The w3af script is attached below

```
plugins
output console, html_file
output
output config html_file
set verbose true
back

output config console
set verbose true
back
crawl web_spider
config web_spider
set only_forward true
back

crawl
audit all
audit


back
target
set target http://www.insecurelabs.org/

back
start

exit
```

After downloading the script, we use command line to run this script using the w3af tool.

```
$ cd path/to/w3af
$ ./w3af_console -s ~/Downloads/w3af1.w3af
```

After w3af is done crawling and auditing, the result is output to the console, showing that multiple requests that w3af used during the auditing process have detected Cross Site Scripting vulnerability of the target website.

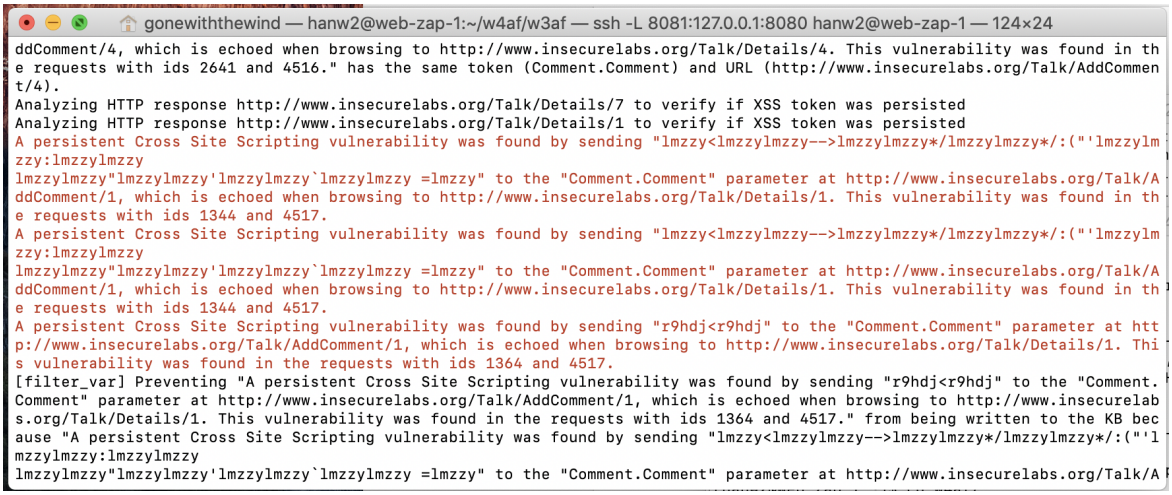


Figure 5.5: w3af result (1)

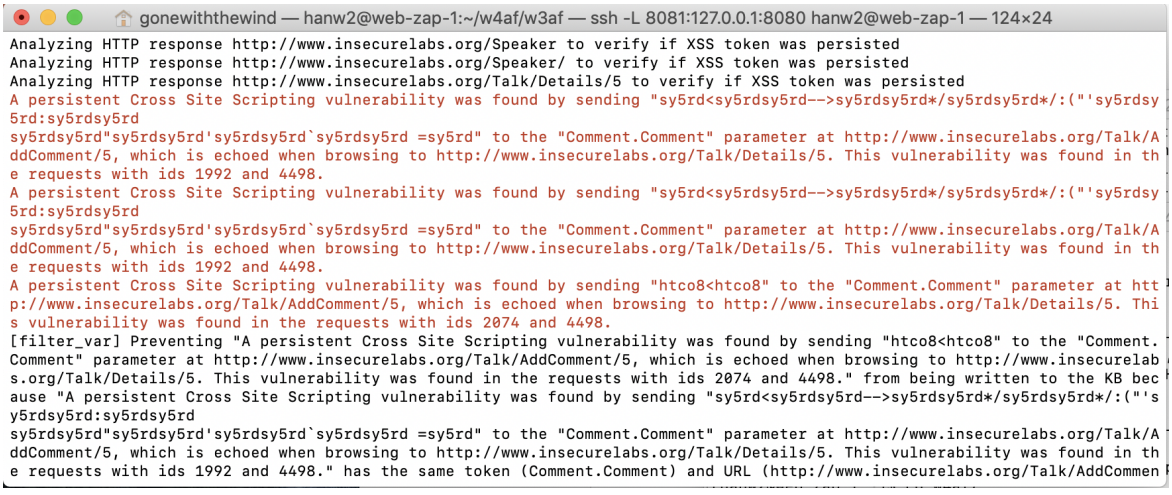


Figure 5.6: w3af result (2)

According to the console output, w3af took 15min 56s to complete the job.

5.2 Endpoints on NHM's staging server

NHM's staging server is the previous stage before websites come out online, where web applications are inspected and tested. All the endpoints here are real applications that have already deployed to the actual server or are to be deployed. So testing against these endpoints can demonstrate if the scripts we generated are actually useful.

Application	URL
Bird types	https://staging.nhm.ac.uk/research-curation/scientific-resources/collections/zoological-collections/type-specimens-of-birds/index.html
Linnean Butterfly Types	https://staging.nhm.ac.uk/research-curation/research/projects/linntypes/
Moss types	https://staging.nhm.ac.uk/research-curation/research/projects/moss-types/
Duxbury	https://staging.nhm.ac.uk/research-curation/scientific-resources/collections/palaeontological-collections/duxbury/search
First Fleet	https://staging.nhm.ac.uk/nature-online/art-nature-imaging/collections/first-fleet/art-collection/
Butterflies and moths	https://staging.nhm.ac.uk/our-science/data/butmoth/
Chalcidoids	https://staging.nhm.ac.uk/our-science/data/chalcidoids/database
Cockayne	https://staging.nhm.ac.uk/our-science/data/british-butterflies-moths/database
Endeavour	https://staging.nhm.ac.uk/our-science/departments-and-staff/library-and-archives/collections/cook-voyages-collection/endeavour-botanical-illustrations
Host plants	https://staging.nhm.ac.uk/our-science/data/hostplants/search
linean plant typification	https://staging.nhm.ac.uk/our-science/data/linnaean-typification/search
macgillivray	https://staging.nhm.ac.uk/our-science/departments-and-staff/library-and-archives/collections/macgillivray
metcat	https://staging.nhm.ac.uk/our-science/data/metcat/search
spruce	https://staging.nhm.ac.uk/our-science/data/spruce
tropical snails	https://staging.nhm.ac.uk/our-science/data/tropical-land-snails
Lichen ID guide	https://staging.nhm.ac.uk/take-part/identify-nature/lichen-id-guide

Figure 5.7: endpoints on staging server

There weren't much finding when I inspected the websites manually, besides receiving a couple of 404 Error as some of the endpoints are no longer valid.

Note that, since the endpoints deployed here are relatively shallow compared to InsecureLabs.org, the time it took for w3af to scan is pretty short – around 15 seconds. And to avoid ZAP crawling the whole staging server (which will happen if the max depth allowed is set to unlimited), we have purposely set max depth allowed to be

3.

5.2.1 ZAP Performance

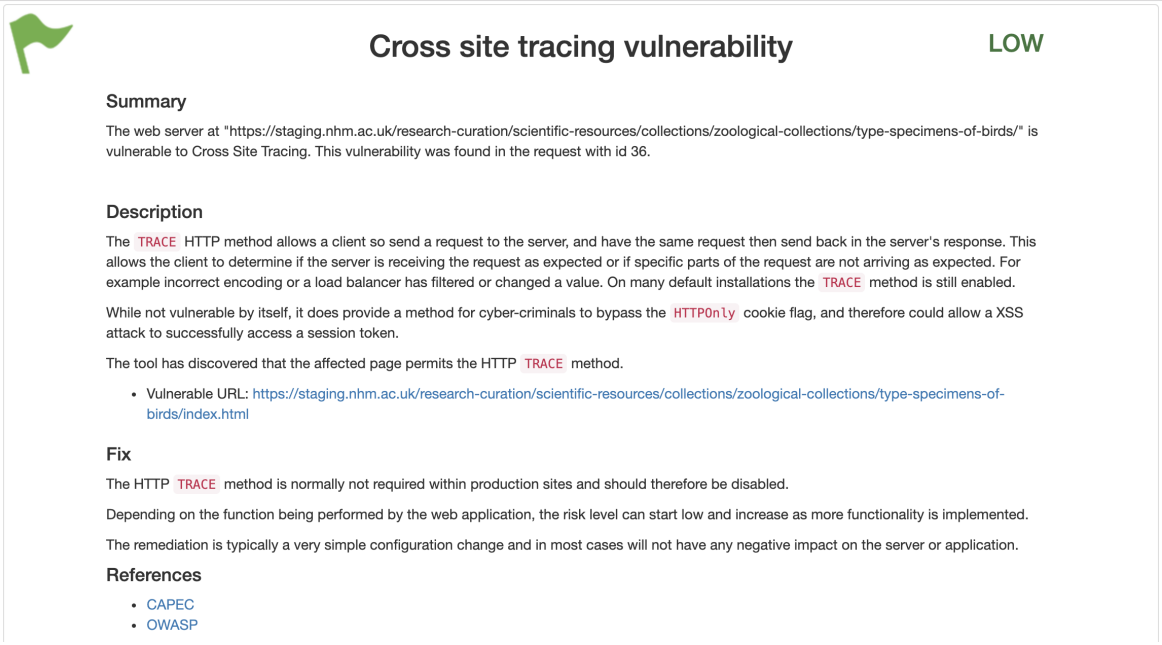
Similar to how we wrote the program for InsecureLabs.org, we wrote ITL programs for the first endpoint, but instead of downloading the scripts and run them locally, we run the generated scripts on the virtual machine this time, since the targets now are all on the intranet of NHM. The result of ZAP tool shows that there are a couple of minor issues, all of which are of low risk, including "Absence of Anti-CSRF Tokens", "Incomplete or No Cache-control and Pragma HTTP Header Set", "Cross-Domain JavaScript Source File Inclusion", "Secure Pages Include Mixed Content" and etc.

For other endpoints (besides those that are no longer on staging server), all the results are similar to the first endpoint.

5.2.2 w3af Performance

For the first endpoints, after running the w3af scripts, the console shows that SSL certificate that is used for the website was found with request id 1, this is probably due to the fact that this is the staging server.

And w3af also shows that the website is vulnerable to Cross Site Tracing, which is a common vulnerability with low risk. If we download the html format report, we can see a more detailed information regarding this finding.



Cross site tracing vulnerability

LOW

Summary

The web server at "https://staging.nhm.ac.uk/research-curation/scientific-resources/collections/zoological-collections/type-specimens-of-birds/" is vulnerable to Cross Site Tracing. This vulnerability was found in the request with id 36.

Description

The `TRACE` HTTP method allows a client so send a request to the server, and have the same request then send back in the server's response. This allows the client to determine if the server is receiving the request as expected or if specific parts of the request are not arriving as expected. For example incorrect encoding or a load balancer has filtered or changed a value. On many default installations the `TRACE` method is still enabled.

While not vulnerable by itself, it does provide a method for cyber-criminals to bypass the `HTTPOnly` cookie flag, and therefore could allow a XSS attack to successfully access a session token.

The tool has discovered that the affected page permits the HTTP `TRACE` method.

- Vulnerable URL: https://staging.nhm.ac.uk/research-curation/scientific-resources/collections/zoological-collections/type-specimens-of-birds/index.html

Fix

The HTTP `TRACE` method is normally not required within production sites and should therefore be disabled.

Depending on the function being performed by the web application, the risk level can start low and increase as more functionality is implemented.

The remediation is typically a very simple configuration change and in most cases will not have any negative impact on the server or application.

References

- CAPEC
- OWASP

Figure 5.8: html format report

We tested against other endpoints and the main issues are that

1. Cross site tracing vulnerability

2. SSL certificates can be obtained with request id 1

3. fuzzable requests with method: GET

To address the first issue, we suggest disabling TRACE method on those websites as it is not required for production sites. And since for all of the endpoints, SSL can be obtained, we conclude that is due to the fact that we are testing on the staging server and NHM has special setup on staging server.

For the last issue, an actual attack will need to be conducted to show if the vulnerability is actually exploitable either manually or using other tools.

Application	URL	Vulnerabilities	Risk
Bird types	https://staging.nhm.ac.uk/research-curation/scientific-resources/collections/zoological-collections/type-specimens-of-birds/index.html	Cross Site Tracing; SSL Cert	Low
Linnean Butterfly Types	https://staging.nhm.ac.uk/research-curation/research/projects/linntypes/	Cross Site Tracing; SSL Cert	Low
Moss types	https://staging.nhm.ac.uk/research-curation/research/projects/moss-types/	Cross Site Tracing; SSL Cert	Low
Duxbury	https://staging.nhm.ac.uk/research-curation/scientific-resources/collections/palaeontological-collections/duxbury/search	404	
First Fleet	https://staging.nhm.ac.uk/nature-online/art-nature-imaging/collections/first-fleet/art-collection/	SSL Cert	Low
Butterflies and moths	https://staging.nhm.ac.uk/our-science/data/butmoth/	Fuzzable request, Cross Site Tracing	Low
Chalcidoids	https://staging.nhm.ac.uk/our-science/data/chalcidoids/database	404	
Cockayne	https://staging.nhm.ac.uk/our-science/data/british-butterflies-moths/database	404	
Endeavour	https://staging.nhm.ac.uk/our-science/departments-and-staff/library-and-archives/collections/cook-voyages-collection/endeavour-botanical-illustrations	404	
Host plants	https://staging.nhm.ac.uk/our-science/data/hostplants/search	404	
linean plant typification	https://staging.nhm.ac.uk/our-science/data/linnaean-typification/search	404	
macgillivray	https://staging.nhm.ac.uk/our-science/departments-and-staff/library-and-archives/collections/macgillivray	404	
metcat	https://staging.nhm.ac.uk/our-science/data/metcat/search	404	
spruce	https://staging.nhm.ac.uk/our-science/data/spruce	404	
tropical snails	https://staging.nhm.ac.uk/our-science/data/tropical-land-snails	404	
Lichen ID guide	https://staging.nhm.ac.uk/take-part/identify-nature/lichen-id-guide	404	

Figure 5.9: tests conclusion

5.3 User Feedback

Besides the tests I did on NHM's staging server, I started a tomcat server on a virtual machine powered by Digital Ocean to host the web editor alongside the tools. I invited several of my colleagues from Imperial College to try out the language I designed, asking them to write a program that runs w3af tests and compare it to installing and using w3af to do those tests.

Each of the invitees are then asked to answer a questionnaire that contains 6 questions asking about

1. the security testing tools that they know of

2. the amount of time that was required for them to write their first ITL program

3. if they have found any of vulnerabilities on their targets

4. if the testing results are helpful, the amount of time that was required for them to install w3af on their own PC or Mac

5. suggestions for the language

Six responses were collected and the following charts were produced based on those.

We can observe that Metasploit is the most known tool to the invitees, and besides one of the invitees that have never heard of any security testing tools listed, all of them have least heard of one tool.

Five out of six invitees were able to write their first program under 15 minutes just by reading the documentation of the language and start testing on their target sites, which are either their personal github.io sites or the websites they developed for their project. Half of them were able to identify some vulnerabilities of the websites, and those vulnerabilities all fall into the category of cross site scripting and cross site tracing. Two of the invitees who found vulnerabilities on their target sites found the testing results helpful and informative, while the other complained about the repetitiveness and tediousness of the results.

When they were asked to install w3af, three of the invitees spent more than thirty minutes debugging and setting up the environment and ceased making an effort; two of the invitees were able to setup w3af successfully but spent more than thirty minutes. One invitee was able to setup w3af without any problem in less than 15 minutes.

Some of the inspirational feedback are that

1. The grammar of the language can be further optimized, i.e., let users set up the hosting virtual machine use parameters to set up the paths once and for all so that they don't have to specify the path of scripts generation and report generation.

2. The testing results can be more condensed, so it is more useful and readable.

Those feedback will be taken into account for future development.

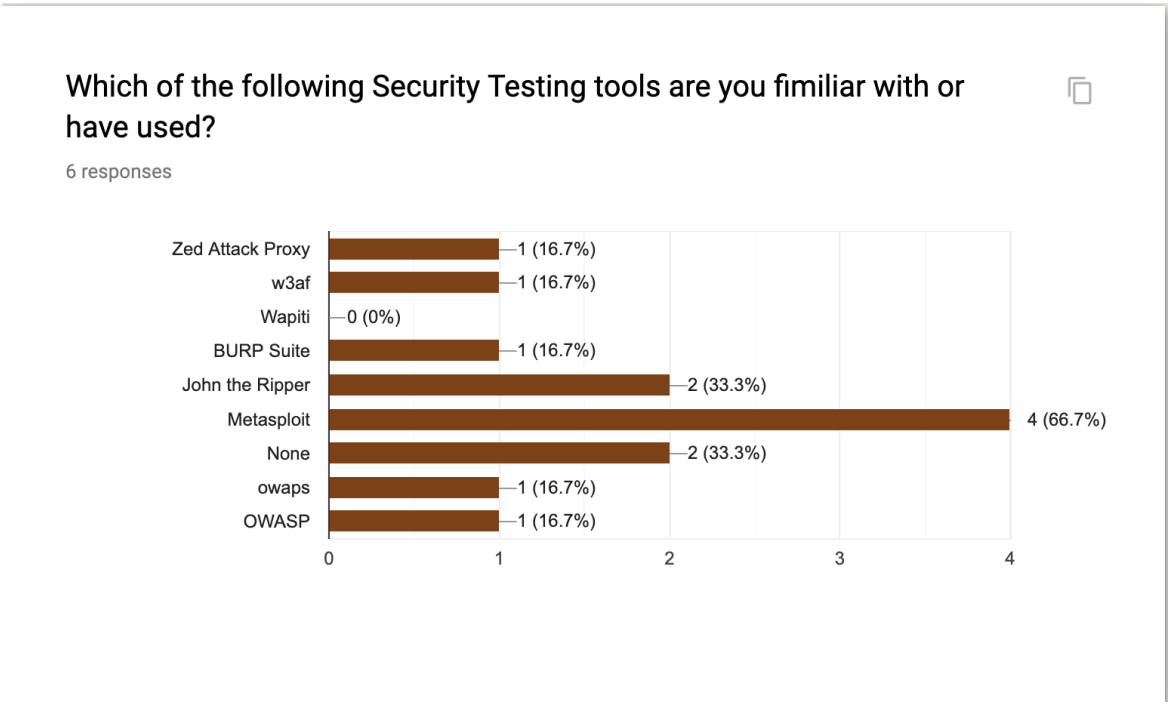


Figure 5.10: users feedback 1

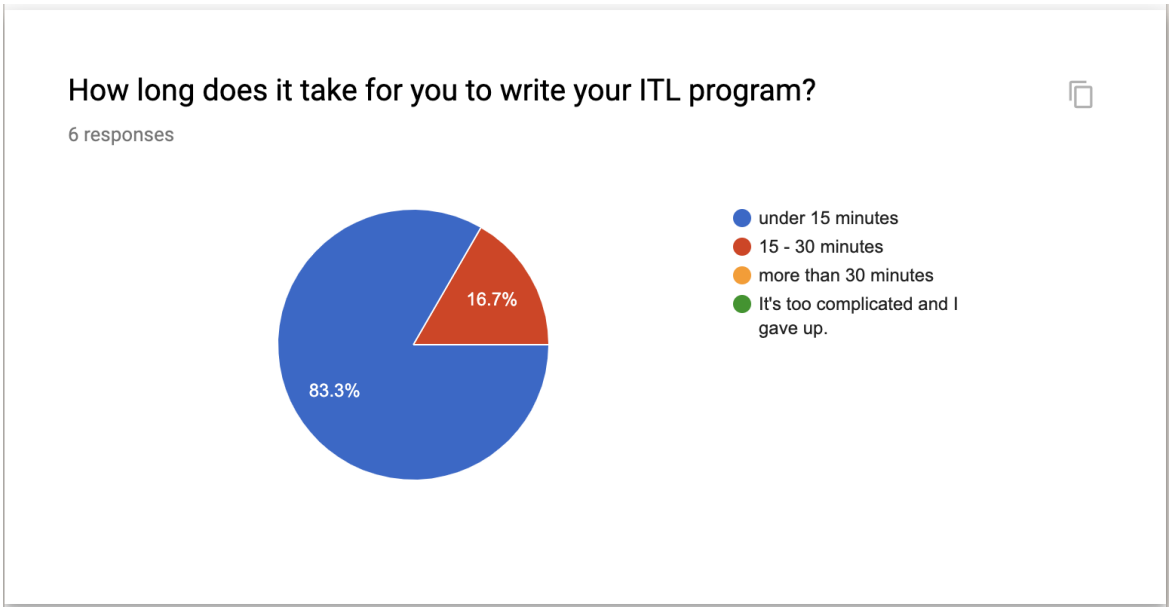


Figure 5.11: users feedback 2

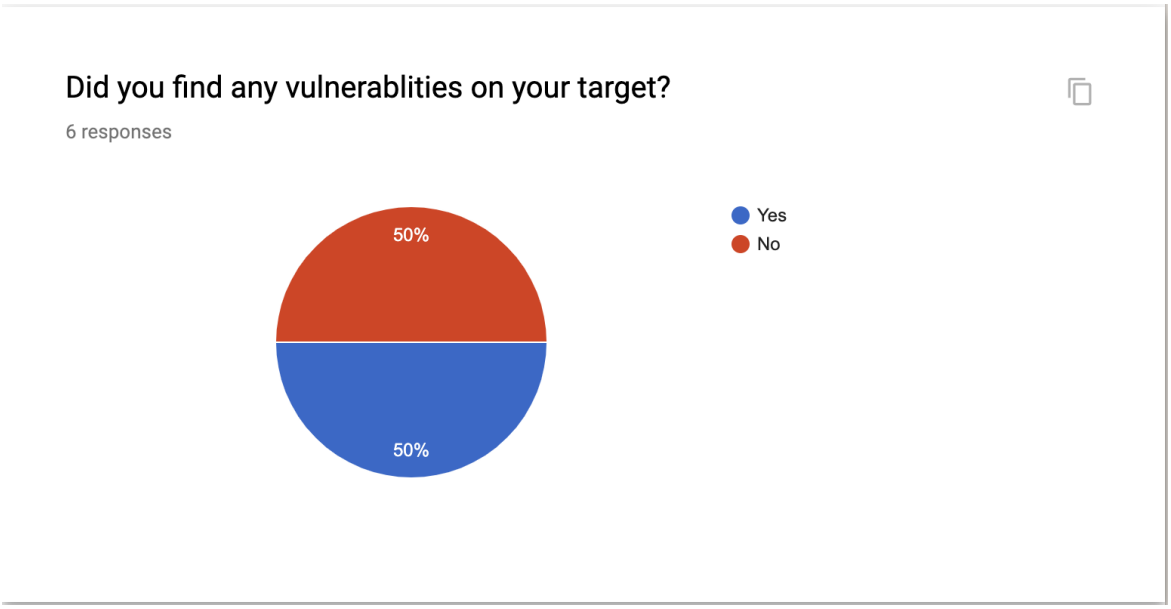


Figure 5.12: users feedback 3

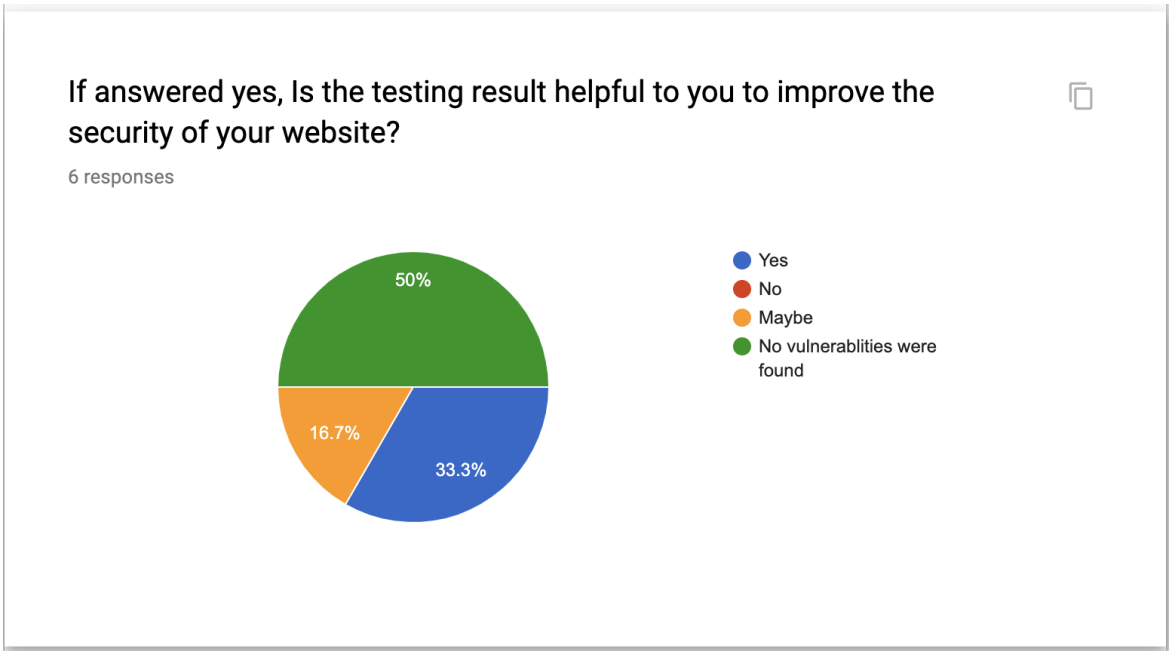


Figure 5.13: users feedback 4

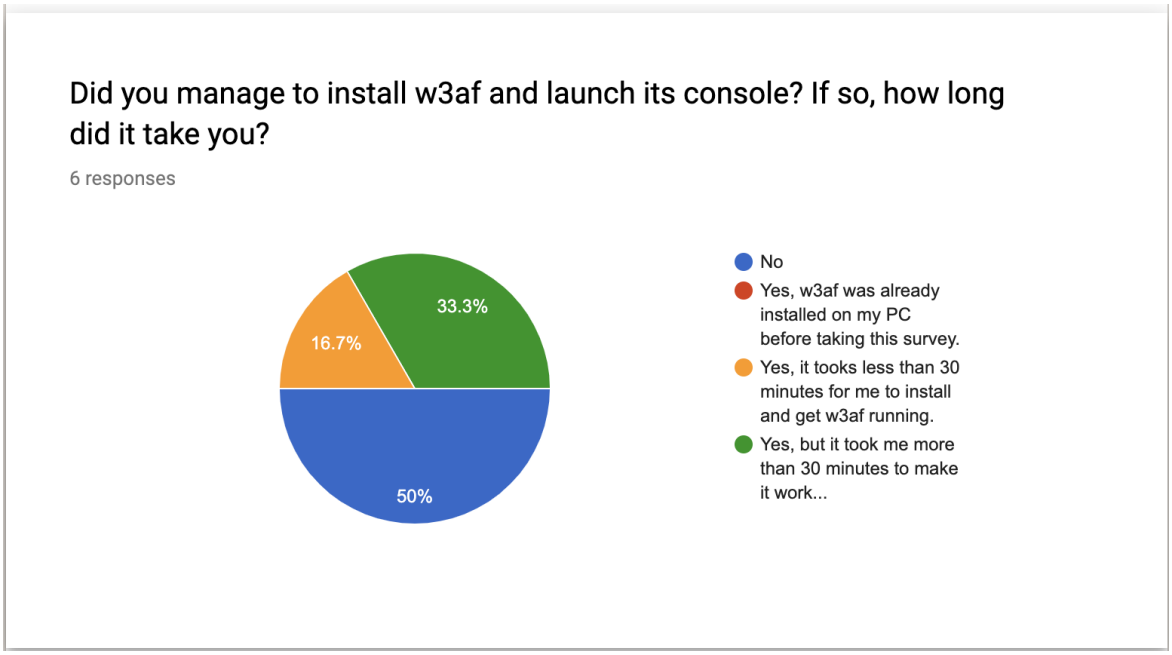


Figure 5.14: users feedback 5

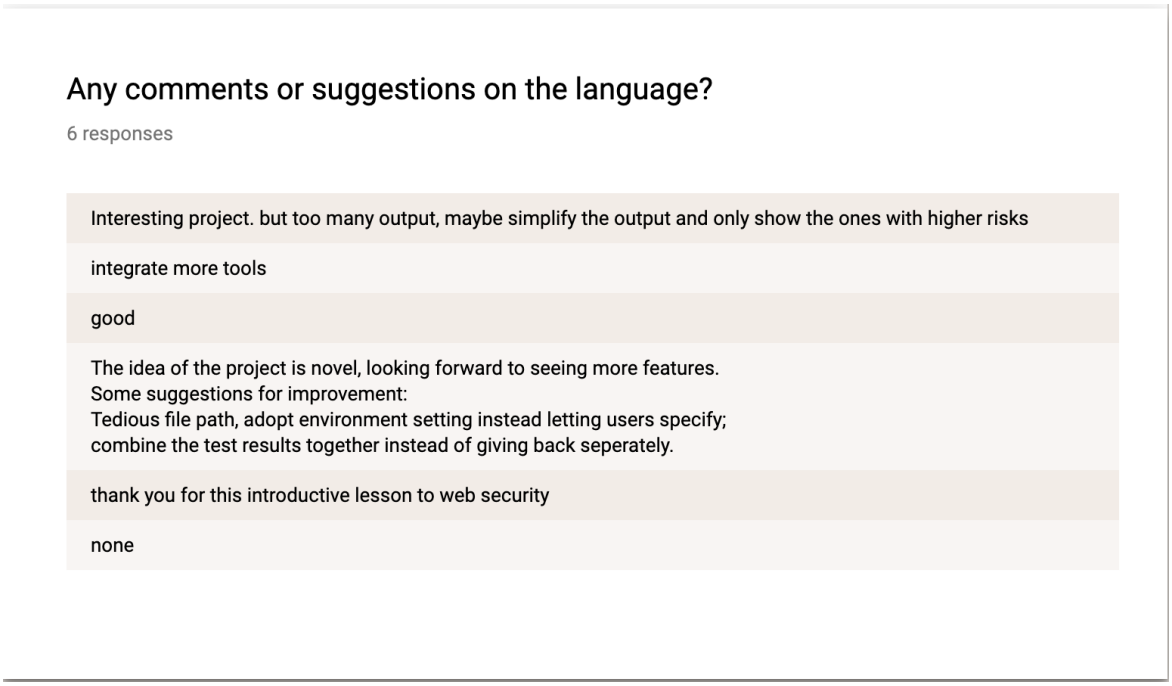


Figure 5.15: users feedback 6

Chapter 6

Conclusion

In this project, we developed a Domain Specific Language that users can use to write a special program to generate scripts that do security testings against their web application remotely or locally as they wish.

A web-based IDE is implemented for the convenience of the users, on which user can write their ITL program, download python/ w3af their code generates to run them locally if they have the tools installed and setup, or run remotely on the webserver that hosts this web editor.

The process of testing is simplified and the tedious and painful environment setup that was previously a mandatory task for each developer wish to test their application was reduced – only one machine needs to install the tools and setup environment, and all developers can just request from the website and wait for the result to be delivered to them as long as the server is not overloaded.

The language is also highly extensible and customizable, new tools can be implemented and added to the language by simply adding additional grammar rules in testing_tools inside the main "ITL.xtext" file and additional generator functions in "ITLGenerator.xtend". With this extensibility, organizations and security experts can add in new tools as they see fit.

This project is also instructive and heuristical to researchers and developers, especially the junior ones since lots of them have never heard of/ unaware of the vulnerabilities their websites may have according to the result of the questionnaire. It is a valuable experience for everyone studying/ working in the field of computer science to be exposed to the testing tools so that they are aware of those vulnerabilities that may cause a colossal amount of loss, and this project has done the job of minimizing the barrier between them and the world of vulnerabilities.

6.1 Limitation

Since this is a prototype project, limitations do exist.

As the first developer, my main focus was to decide the architecture of the project, and to experiment with the tools, identifying the best one to be implemented in the language. Because of the time constraints, only two tools are implemented with limited scanning option. However, thanks to the extensibility of the project, much more tools can be hopefully implemented in higher level of details in the future.

Also, some of the grammar was unnecessary, for example, the user shouldn't have to specify the path where the script will be generated every time. Instead, it should be anchored when the server was set up, once and for all.

Besides the limitation of the project itself, the limitation of all testing tools is the barrier that is nearly impossible to breakthrough. It is a fact that tools are unable to detect a combination of vulnerabilities which may result in a critical risk. I'm using this example explained by "Jeroen - IT Nerdbox" on StackExchange (https://security.stackexchange.com/questions/73744/w3af-and-automated-vulnerability-scanners-vs-manual-testing) –

1. Session cookie does not have the httpOnly attribute set
2. Application is vulnerable to CSRF attacks
3. Application is vulnerable to Persistent XSS attacks

No. 1 is considered a low risk here while No.2 and 3 are both considered high risk, but when these vulnerabilities all exist on the same website, a hacker can exploit these three all together to commit a session hijacking which is critical in terms of risk.

Therefore, we can not rely on the language to detect all the vulnerabilities – it is advised to use these tools to gather information, do an early assessment and then audit the websites manually.

6.2 Future Work

1. As mentioned previously, more tools can be implemented to extend the scope of the language.

2. A docker image can be created to contain the websites of the editor, zap and w3af that was required to run the generated scripts. So that the whole project is packed into a docker image and it can be deployed anywhere with a ”docker run” command, and the machine becomes our ’web-zap-1’ (name of the virtual machine hosting the web editor) at Natural History Museum.

Bibliography

[1] C. Liu and D. J. Richardson, ”Automated security checking and patching using TestTalk,” Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering, Grenoble, France, 2000, pp. 261-264. pages 3

[2] J. Bau, E. Bursztein, D. Gupta and J. Mitchell, ”State of the Art: Automated Black-Box Web Application Vulnerability Testing,” 2010 IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, 2010, pp. 332-345. pages 3

[3] R. Vibhandik and A. K. Bose, ”Vulnerability assessment of web applications - a testing approach,” 2015 Forth International Conference on e-Technologies and Networks for Development (ICeND), Lodz, 2015, pp. 1-6. pages 3

[4] X. Wang and P. Xu, ”Build an Auto Testing Framework Based on Selenium and FitNesse,” 2009 International Conference on Information Technology and Computer Science, Kiev, 2009, pp. 436-439. pages 4

[5] Fowler, M. and Parsons, R. (2011). Domain-specific languages. Boston, Mass: Addison-Wesley. pp. 121-128 pages 10

[6] Bettini, L. (2016). Implementing Domain-Specific Languages with Xtext and Xtend - Second Edition. Packt Publishing, pp. 1-312. pages 10

[7] Engebretson, P. (2013). The basics of hacking and penetration testing. Amsterdam: Syngress, an imprint of Elsevier. pages 10