M.Sc. Thesis
Master of Science in Engineering

Multi-step scanning in ZAP

Handling sequences in OWASP ZAP

Lars Kristensen (s072662)
Stefan Østergaard Pedersen (s072653)

Kongens Lyngby 2014

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

English

This report presents a solution for scanning sequences of HTTP requests in the open source penetration testing tool, *Zed Attack Proxy* or ZAP. The report documents the analysis, design and implementation phases of the project, as well as explain how the different test scenarios were set up and used for verification of the functionality developed in this project. The proposed solution will serve as a proof-of-concept, before being integrated with the publically available version of the application.

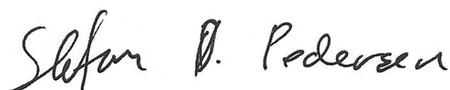
Dansk

Denne rapport præsenterer en løsning der gør det muligt at skanne HTTP forespørgsler i open source værktøjet til penetrationstest, *Zed Attack Proxy* eller ZAP. Rapporten dokumenterer faserne for analyse, design og implementering af løsningen, samt hvordan forskellige test scenarier blev opstillet og anvendt til at verificere funktionaliteten udviklet i dette projekt. Den foreslåede løsning vil fungere som et proof-of-concept, før det integreres med den offentligt tilgængelige version af applikationen.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfilment of the requirements for acquiring a M.Sc. degree in respectivly Computer Science and Engineering, and in Digital Media Engineering.

Kongens Lyngby, September 5. 2014



Lars Kristensen (s072662)
Stefan Østergaard Pedersen (s072653)

Acknowledgements

The authors would like to thank the supervisor of this project, Christian W. Probst, for guidance and feedback throughout the entire project.

The authors would also like to thank the ZAP Developer Community, especially Simon Bennetts, the project leader for ZAP, and Cosmin Stefan Dobrin, a core developer of ZAP, for technical guidance and for always being helpful when we had questions.

Special thanks goes to our families, for their patience and eternal support.

Contents

Summary i
English . . . i
Dansk . . . i

Preface iii

Acknowledgements v

Contents vii

1 Introduction 1
1.1 Motivation . . . 1
1.2 Vision . . . 1
1.3 Thesis Definition . . . 2
1.4 Scope . . . 3
1.5 Development Methodology . . . 3
1.6 Report Structure . . . 4

2 Background 5
2.1 The Internet . . . 5
2.2 The HyperText Transfer Protocol (HTTP) . . . 6
2.3 Web Security . . . 10
2.4 The Open Web Application Security Project . . . 16
2.5 The Zed Attack Proxy Project . . . 18

3 Analysis 23
3.1 Scenarios . . . 23
3.2 ZAP Workspace . . . 26
3.3 ZAP Functionality . . . 27
3.4 ZAP User Experience . . . 33
3.5 Target Audience . . . 35
3.6 Requirements . . . 38
3.7 Test Strategy . . . 40

4 Design 43
4.1 Extending ZAP . . . 43

4.2 Hooking into ZAP 44
4.3 Handling Sequences 46
4.4 User Experience 50

5 Implementation 55
5.1 Iteration 1: ScannerHook 55
5.2 Iteration 2: Proof Of Concept 57
5.3 Iteration 3: Optimization 61
5.4 Iteration 4: Injection 63

6 Test 65

7 Conclusion 71
7.1 Project Conclusion 72
7.2 Future Work 73

A User Manual 75

Bibliography 81

CHAPTER 1

Introduction

This project covers the development and implementation of an extension of functionality in the OWASP ZAP project. This chapter will give a general introduction to the project together with thesis background and motivation. Beyond this, the chapter will also outline how the project was developed and provide an overview of the rapport structure.

1.1 Motivation

As most developers know, security in web applications is an important feature, but the last couple of years has shown many examples of compromised security in web applications. Though many of the techniques and vulnarbilities used by hackers to corrupt or gain access to web-applications have been known for years, their are still widely used today.

To help developers and web-masters create a safe environment for their web applications, several tools exist to perform various scans and penetration-testing in order to find any known vulnerabilities. One such tool is the Zed Attack Proxy (ZAP)[1], which is used for penetration test. ZAP is an open source project developed within the Open Web Application Security Project (OWASP)[2].

As two masters degree students in respectively computer science and digital media we have both had different courses concerning computer security with various focus. At the same time we have both worked with web application development and know how much of a challenge it can be to test and verify that a desired level of security is obtained. Because of this, we feel that it is very relevant for us to become engaged in the development of a tool such as ZAP. It would that we can contribute to the development of a large and very wide-spread tool, and at the same time learn some of the techniques used when testing web applications.

1.2 Vision

The purpose of our thesis is to make it possible to invoke sequences of HTTP requests when performing a scan of web applications through ZAP. Based on this, the overall vision is to extend ZAP, and contribute to a large scale open source project. The

[1] `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`
[2] `https://www.owasp.org/`

extension should be adapted and adjusted to fit into ZAP's development strategy, both regarding the interface, and structure of code. The development is done with the future deployment of mentioned functionality in mind, but for the duration of this project, the goal is to provide a proof-of-concept extension for the ZAP workspace.

1.3 Thesis Definition

ZAP is a tool that can perform various tests and scans by making it possible for users to group web-application sites together based on HTTP requests. The goal of this thesis is to create a mechanism for ZAP to handle sequences of HTTP requests. To achieve this, we have identified a number of questions. The questions serve as sub-goals for this thesis. How these goals are going to be achieved will be elaborated throughout the rest of the chapters in this report.

Problem Definition

- Is it possible to make ZAP implement sequences in such a way it would find vulnerabilities that would otherwise not be discovered?
  - Why are sequences important while performing an active scan using stateless HTTP?
  - Which vulnerabilities are possible to expose using sequences?
  - How is it possible to extend ZAP to support sequences?

Thesis goals

By extending the problem definition, the following goals can be created:

1. Make ZAP able to handle sequences of HTTP requests during an active scan.
2. Make it possible for users to create, modify and persist sequences in ZAP.
3. Extend functionality in ZAP in a optimal way, so low coupling and high cohesion is achieved as much as possible.
4. New sequence functionality must be integrated into ZAP in a way that feels natural and follows the application standards.
5. Development and communication is done in a way that matches that of a Open Source Community, which should feel that they are able to impact any choices needed for this expansion.

1.4 Scope

The primary focus of the development is to create a proof-of-concept of a solution concerning the mentioned thesis goals. Ideally a complete extension will be part of a released version of ZAP, but it is not a goal for this project. Furthermore, the development will not attempt to fix or change any application wide problems or malfunctions. Nor is the goal to change the overall structure of the software since this is part of a much larger procedure. The developed software should be integrated as much as possible into the existing structure of ZAP.

1.5 Development Methodology

The development of this project is done in collaboration between the participants (the two authors) and the ZAP development team. It is important to make sure that there are matching expectations between all contributors, and that there is a clear working plan of how the project ideally should proceed.

The approach taken for this project has been that of a partially agile development strategy. It has been important to work closely together so that key decisions would be taken jointly. The development itself has been split in to a series of iterations, but since this solution primarily is a proof-of-concept, each iteration is mostly for determining that the development is moving in the desired direction. However, the iterations will continuously keep adding functionality and complexity to the final product. Unlike business style agile development, the project is not a continuously developed product with many developers associated. This means that any correction or change to the solution is rare and is primarily made by the authors independently and in cooperation with the ZAP development team at regular intervals. Each day starts with a meeting where the daily goals are declared and discussed. This makes sure that both members of the group are aware of what should be done and what the current challenges are.

Another aspect is to work for and together with an open source community such as the ZAP community. Regular meetings with the project lead, Simon Bennetts[3], will make sure that development is on the right track. While this naturally is very helpful, it is also necessary to get feedback and comments other developers in the community. This will be done respectively by using the development community forum[4] and by posting to a blog that has been created for this project[5].

[3] https://www.owasp.org/index.php/User:Simon_Bennetts
[4] https://groups.google.com/forum/#!forum/zaproxy-develop
[5] http://zapmultistep.wordpress.com/

1.6 Report Structure

In this section, a brief description of the following chapters of this report will be given.

After this introduction, a chapter describing the background of this project will be presented. This is to further elaborate on some of the elements described in this introduction, and give an overview of which concepts will be explored throughout this project. The chapter concludes with a brief introduction to ZAP, which is the Open Source software project that will be expanded upon in this project.

The background chapter is followed by an analysis chapter. Here, the architecture of ZAP will be presented, including an overview of the graphical user interface, the codebase of ZAP, and how scripting is used in ZAP. The chapter also includes an analysis of what the target audience of this project is. The analysis chapter concludes with a set of requirements to the code that will be produced, and a description of the test strategy that will be used in this project.

The following chapter will describe some of the design choices that have been made, based on the findings of the analysis chapter. A description of how some of the core parts of the solution will be presented.

The implementation chapter will focus on giving in-depth descriptions of how specific elements were developed to achieve the goal of the project.

The following chapter will describe testing of the developed functionality. This includes following up on the test strategy described earlier. A selection of examples used throughout the development process will also be given.

The final chapter will conclude on all the previous processes of the project. A follow-up on the questions that were posed in the Thesis definition will be presented. A brief description of future work on this project will also be presented.

CHAPTER 2

Background

In the previous chapter, the problem definition has been explained, along with our motivation for doing this project. This chapter serves to elaborate some of the relevant concepts, and give an in-depth explanation of some of the tools and techniques that will be used in this project.

The chapter begins with a brief description of the evolution of the Internet, followed by a look at the HyperText Transfer Protocol. Afterwards, a description of some concepts regarding web security are described, along with related examples of threats and attacks. The next section gives an introduction to OWASP, which is an organization that aims at improving security in Web Applications. The chapter concludes with a description of ZAP, which is a free Open-Source tool for penetration testing and vulnerability assessment in Web Applications.

2.1 The Internet

In the beginning of the 1960s the first plans for a network, what was then called ARPANET was developed and almost 10 years later a network consisting of initial four nodes were created. More nodes were continuously added, and by 1970 a basic communications protocol was created, called Network Control Program (NCP). However as the amount of users began to increase the NCP was revealed to have problems. For instance, it did not support data reliability and if packages would be lost it could be difficult to detect. At the same time the NCP protocol was not really flexible enough to handle larger interconnected systems and this meant that a new protocol would have to be developed. By 1983 the Transmission Control Protocol(TCP) was released, which, in combination with IP technology, is still the standard today. TCP (and UDP), was a major step forward for ARPANET which would grow into the modern Internet. [LCC+09]

The early experiences with ARPANET gave birth to the modern structure of the Internet. A common way of explaining the various structural parts of the Internet is using a layered model also referred to as the OSI model, which consists of 7 layers.

1. The Physical layer. This layer, as the name suggest, describes the physical elements, such as wires etc., and is primarily focused on the transfer of binary data.

2. The Data-link layer is responsible for transferring data between modules in the network. This is done by using hardware specific addresses such as MAC addresses.

3. The Network layer can be defined as packages. Packages contains an IP address (either IPv4 or IPv6). IP addresses are unique so will function like an global address, and will make it possible to transfer data to a specific node on the network.

4. The Transport layer describes how data is to be transferred. This layer contains protocols such as TCP or UDP. TCP for instance enforces reliable delivery of packets where UDP is more focused on speed of delivery which can be desirable, for instance when streaming data.

5. The Session layer handles connections between computers, and make sure to maintain or shut down connections appropriately.

6. Presentation layer. Handles presentation of data for applications. This includes MIME encoding or encryption/decryption of data.

7. The Application layer. Finally the application layer is a process on a computer. Usually addressed by a port on a computer. Applications is responsible for displaying the received data to the user.

Every time a communication is made through the internet, each of the mentioned layers will be utilized for transporting data back and forth between the user and the target. A common analogy is to compare the structure with that of sending a physical package with a postal service. The name of the person represents the end-receiver, in this case a process receiving a message through the internet. The house address is comparable with a IP address and so on. The point of this is to explain that each layer has a specific task when sending a message through the Internet, and every layer is required when doing so.[1]

2.2 The HyperText Transfer Protocol (HTTP)

Today most people associates the Internet with the World Wide Web (WWW, or web), but the two are not entirely the same. While the Internet had evolved since the early days of ARPANET, the web was not developed until the beginning of the 90s, where the HyperText Transfer Protocol(HTTP) was created. The HTTP is part of the top application layer of the OSI model. Designed to be simple, the first version did nothing more than making it possible for a client to request data by using an ASCII-string. The resulting data would also just be returned as text and were

[1] http://en.wikipedia.org/wiki/OSI_model

initially restricted to HyperText Markup Language(HTML) files, which had been created together with the HTTP.[Gri13]

Since then the popularity of both the Internet and the web has exploded. Where there in 1990 existed one website on the web, there are now billions[2]. HTTP and HTML are important parts of the structure, that makes it possible for most people to use the Internet. Initially web content was mostly static information making it easier for people to find certain data. Today, social and user generated aspects has become an increasingly popular feature. The term *Web 2.0* is often used to classify this transition, even though it does not refer to any actual upgrade of technology. While there currently is a HTTP 2.0 version in development, the current standard of HTTP is 1.1, which was released in 1997 (updated 1999). Compared to the first version many elements has been added or adjusted.

The HTTP protocol is by design stateless. This means that the protocol inheritable will not handle states or modes. Every communication consists of a request and a corresponding response, and does not by default have any notion of what the user or requester has done previously (except for an optional referrer header).

```
1 POST http://localhost:8181/zap-test-webapp/sequence/step2.jsp HTTP/1.1 2
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:31.0) Gecko/
20100101 Firefox/31.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: http://localhost:8181/zap-test-webapp/sequence/step1.jsp
3 Cookie: JSESSIONID=EC674BF59A698568062D6AE693278A93
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
4 Content-Length: 25
Host: localhost:8181

5 title=qwert&message=qwert
```

Figure 2.1: An Example HTTP request message

Figure 2.1 shows an example of a HTTP request. Every request contains a request method (1), in this case the request uses a POST method. HTTP is created with the idea that users wants to use or manipulate server resources, and the request method indicates what the user attempts to do with a specified resource. There are four request methods, which are generally considered as the primary methods: GET, PUT, POST and DELETE. GET very simply asks the server to return the specified resource. PUT is designed to replace an existing resource, or create it if it does not already exists. Simultaneously PUT is designed to be idempotent, meaning that calling this method multiple times in a row should result in the same outcome as calling

[2] http://googleblog.blogspot.ca/2008/07/we-knew-web-was-big.html

it a single time. POST can be used for various tasks, but usually means create a object. Unlike PUT, the POST request method is not considered idempotent. Finally DELETE removes a resource. There are many other request methods that have different definitions, but the previous mentioned ones are the most common. It is important to mention that requests methods are only a definition, and relies on the individual servers implementation, so often the guidelines are not entirely ensured.[3]

Related to the request methods is a reference to the resource that wants to be addressed, which is determined by a uniform resource identifier (URI). This is a string of characters, and is usually a uniform resource locater (URL), more commonly known as a web address. A web-address consists of a protocol, followed by the host-name which is sometimes followed by a port if the default port is not used (default is 80). After the host-name comes a path the to resource on the server, which is usually a web-page file of one type or another, but could also be other file types such as images etc. The path is based on a file path on the servers directory, but some server types makes it possible to manipulate it, so the requested path is not the actual location of the resource on the server. In the example the URL ends with the path, but in many cases it will also contain a list of parameters. These variables are often set from a referring link, and will inform the server of certain information before generating a response page. Finally, the request displays an HTTP protocol standard used(2).

The remaining part of the request message is mostly comprised of optional metadata. In the HTTP 1.1 standard only the host field is a requirement, but other data can be necessary for the specific requested server in order to return a valid response. In this case the request also contains a HTTP cookie field(3). A cookie is a way for the server to store a value on a requesters computer through the browser, and can only be a name together with an associated value. Because of this it is not possible for a cookie to contain malware or viruses, but can however be used for tracking user behaviour on the Internet. A cookie is created from a HTTP response message, where the response header contains a message (Set-Cookie) that sets the name and value of the cookie. Furthermore there are different attributes that can simultaneously be set, this includes a domain and path for where the set cookie is relevant, together with an expiration date, though these are not required.

The final part of the HTTP request message contains the request body. This means that instead of sending data as parameters, it is possible to send it with a request body as well(5). Usually, this is only done when using a POST request method, but it is not strictly enforced. The Content-length(4) header field indicates how many characters the body consists of.

Figure 2.2 shows an example how a HTTP response can look like. Besides the protocol, the first element delivered is a return code(1), which informs of how the re-

[3] `http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html`

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=EC674BF59A698568062D6AE693278A93; Path=/zap-test-webapp/; HttpOnly
Content-Type: text/html;charset=UTF-8
Content-Length: 1443
Date: Thu, 21 Aug 2014 13:25:36 GMT
```

Figure 2.2: An Example HTTP response message

quest has been handled. The return code of 200 means that the request has returned successfully. Generally, all return codes in the 200s means that the call has been successful. A short overview of the return codes can be seen below.

- 100: Return codes in the 100s are not as clearly defined as the following return codes, are rarely used. However they are used for indicating that only part of the communication has been made, and the specific number indicates how the client should proceed.
- 200: Successful calls are responded by a 200 return code.
- 300: Means a redirection, and usually the browser will do this automatically.
- 400: Return codes in the 400s, means a client error. These return codes informs the requester that they have made some kind of error, for instance syntax errors in the resource that they have requested.
- 500: This means a server error. Returned when a error has occurred on the server.

Figure 2.2 only shows an example of a response header. In addition to the header, the response contains a body, which is the actual content that will be delivered to the user. This could be a HTML page or similar. The size of the body is displayed by the content-length field.

HTTP Sequences

HTTP is designed to be a stateless protocol. This means that each request should be understood by the recipient, no matter what previous requests have been made. However, most web applications want to maintain a state for the users of their site. This can be achieved in several different manners, while still using the HTTP protocol.

The web application could instruct the user (or rather, the browser) to resend any information with each request, but that is not very practical. Instead, cookies can be used to track the users data. Cookies are created in the response of a HTTP request,

and subsequently included in each following request (within the expiration period of the cookie).

Cookies can store user-related values, but are usually meant for small pieces of information, such as authentication status or temporary tokens. For larger collections of data, the server can maintain a Session for the current user. The user then simply needs to identify which session ID is used (which is usually stored in a cookie), and the session data will be available.

Web application implement these states for a lot of different reasons. Usually authentication is the main reason, but once a user is authenticated, the web application could present the user with different possible actions. Some actions may require several linked requests, like a *wizard*[4]. This is often seen in online forums, where a user can review a post before actually posting it. Online shopping websites also use this approach, where the payment process is usually handled in several different requests. Some websites also implement mechanisms to ensure that requests are performed in a specific order, by checking the **referer**[5] header field, in the HTTP request. A page may not be accessible before the previous requests have been made.

In this project, a **sequence** is defined as simply being a series of HTTP requests, in a specific order. Any information related to the users state in the application may be included in the requests, but ultimately the context in which the Sequences are created, are irrelevant.

Once a Sequence has been defined, it can be iterated to perform each request and observe the resulting response. If any additional information needs to be set in the HTTP requests for the execution to be successful (e.g. Authentication cookies), it should be handled by the client that performs the execution of the sequence.

2.3 Web Security

Back in the early days of the Internet, security was not a major concern since the amount of users was rather limited, and the primary concept was to share data between scientists. Somewhere along the process of becoming a global network, it became obvious that not every user on the Internet had good intentions. When the connected nodes were limited to universities, it was most likely easier to find whoever was accountable for aggressive behaviour, or simply had just made a mistake. When dealing with an international network consisting of billions of computers and network components, it is not as easy a task.

[4] `http://en.wikipedia.org/wiki/Wizard_(software)`
[5] `http://www.w3.org/Protocols/HTTP/HTRQ_Headers.html#z14`

During the 1980s, inexpensive computers started to become accessible, but this also meant that hackers and crime related to the Internet also began to appear. Occurrences of organized hacking became publicly known, such as Captain Zap (Ian Murphy), who successfully hacked the American telephone corperation AT&T, and inverted current phone rate system making it cheaper to call during prime times, and vice versa. Another example is Robert Morris who created the *Morris worm*, which spread it self to thousands of UNIX servers [Day13].The 1990s saw the emergence of the web, which made it possible for destructive viruses and malware to become widespread. However, the nature of the Internet essentially meant that enforcing security had become a much more complicated task. In recent events where web security has been compromised, stakeholders are often concerned about loss of information from various sources. In 2014 a bug was found in the Open-SSL cryptographic library, called the Heart-bleed bug. The bug made it possible to read data from a servers memory, basically enabling outsiders to gain access to assumed secure information[6].

Today, computer security is a comprehensive topic, and can be divided into smaller subjects. Data security is primarily focused on securing the data while storing or transmitting it. This is often achieved by cryptography, which is the concept of making data as difficult as possible to decrypt, without having the correct key. This ensures that the data is secure, even if it is intercepted. When using the HTTP, cryptography is often applied by using a secure variant called HyperText Transfer Protocol Secure (HTTPS)[7]. Network security is often defined as the act of securing the actual network that is being used, primarily based on the lower levels of the OSI-model. This can be achieved by using firewalls, and in general making sure that accessing system resources is authorized.

In the process of defining security in computing, there are some keywords that are often used to describe the primary goals. These are also often called the *CIA properties*, and consist of confidentiality, integrity and availability.

Confidentiality

Confidentiality describes how a secure network ensures that it is not possible for unauthorized users to view information, that require authentication and authorization, making sure that perceived secrets are secret. This could be content such a credit card information or passwords. Usually confidentiality can be achieved by usage of cryptography.

Integrity

Integrity in computer security means that it should not be possible to modify information when unauthorized. This means that users can be sure that data is what

[6] `http://heartbleed.com/`
[7] `http://en.wikipedia.org/wiki/HTTP_Secure`

they expect it to be, and has not been modified. This concept is also known as data integrity. Another aspect is integrity of origin, where the source of the information is also as it is expected to be. Integrity can be ensured by having functionality that prevents data to be modified without authentication.

Availability

As the name suggests, availability means that the system is available when users need it to be. Attacks against availability is often connected to Denial of Service (DoS) attacks, which usually makes sure that users will not be able to use a given system while the attack is being conducted. Availability is very difficult to guarantee, since DoS attacks are tough to identify.

Other

Besides the properties above, there are some additional ones which are often used together with the *CIA properties*. Authentication implies that users are who they claim to be. Non-repudiation makes sure that actions are traceable, and it is possible to locate who is accountable for any problems.

Threats and vulnerabilities

In this section, some of the threats to be aware of when dealing with security, will be presented. Threats can be divided into four main categories. Disclosure, deception, disruption and usurpation[8].

- Disclosure is unauthorized access of information.
- Deception describes cases where the system accepts false data believing it is valid.
- Disruption is, as the name indicates, a disruption of a service or operation.
- Usurpation means that unauthorized entities gain control of the system.

There are many subcategories of the above, but the four points listed are the ones that should primarily be addressed when designing a secure system. Threats are only a problem as long as there is a vulnerability to be exploited. Vulnerabilities are often defined as a weakness in the security architecture, but can also be a weakness in organization, or just poor security awareness. Naturally, it is desirable to prevent attacks, but this might not always be possible. Alternatively, systems can deter attacks by making it as difficult as possible, and simultaneously make other targets more interesting. Common network attacks can be broken into several categories. Active attacks include modification, deletion or fabrication of a communication or

[8] http://en.wikipedia.org/wiki/Threat_(computer)#Threat_model

data (often the ones most relevant when looking at web-security). Passive attacks are often eavesdropping or similar, where external sources can gain access to information, otherwise hidden or secret, without alarming the attacked system. [Day13]

The Attacker

The stereotype hacker, known from numerous movies, is based on some of the persons that began hacking in the 1980s. However, the overall type of attacker is more diverse than one might think. Means, motive and opportunity needs to be considered when identifying the typical attackers. In many cases, people that have previously been employed in a company, are the most likely to be an aggressor, since they might often have both the means, since they know the system, and the motive. The classic hacker/cracker has however not completely disappeared. Their motives vary from being a so-called black-hat hacker, who means to cause damage, or a white-hat, who simply sees it as a challenge to locate vulnerabilities, and will inform the owners once any weaknesses are located. Hackers vary in organization, and many might be part of groups. Groups can be criminal organizations or even terrorists. "Script-kiddies" is another group of attackers. Their knowledge with programming and finding vulnerabilities are limited, and will have to use existing tools to hack into systems. This often means that they cannot find new vulnerabilities like the professional hackers can.

Another, and until recently not very mentioned group, is government militaries. Even though it was known that military agencies had cyber divisions, it was largely unknown to what extent their surveillance or activity was. Whistle-blower Edward Snowden has since then revealed that the scale is much larger than many might have expected. Government spies have the technical means, and almost unlimited resources. [PP11]

Penetration Testing

It is difficult to see the exact differentiation between network security and web security. The latter is focused on the HTTP and the server client structure of web communication. Typical components consists of a server that provide access to web-content, and a client, which normally uses a web-browser to access server-content. Testing web content can be a difficult task since it is dependant on both user interaction, and a connection between server and client, and also often the technologies used on both server and client end.

One common way of testing security is to actually try and exploit vulnerabilities yourself. This process is called penetration testing, abbreviated pentesting. Even though it is not limited to web, it is often used when testing web-applications. The idea of pentesting is to find vulnerabilities before they are found by other malicious agents, and thus prevent data-breaches etc. Other examples of why to use pentesting could be to simulate an attack, and survey whether or not it will be detected by intru-

sion detection systems. Developers usually know that they need to consider security when creating applications, but will often not be able to verify that all elements are safe. Vulnerability inspection or scanning might also help find weaknesses, but until actually tested it is not possible to be completely sure.

Pentesting can be done at different levels, and can be divided into white-box, and black-box testing. In white-box testing, some of the system is already available, and the testing system knows some background information, before performing an attack. Black box testing is the opposite; the tester has no or limited information about the system, and will often need to refer to social engineering or other non-technical approaches, in order to get additional required information[9].

When penetration testing there are a series of steps that are usually performed. Firstly, the tester needs to have a goal, which could be to breach a database or similar. Another element is reconnaissance and discovery, locate what is available and how to access it. The next step is to attempt to exploit vulnerabilities, for instance by injection or fuzzing input data (Fuzzing means to input semi-random data). Another important step when pentesting is evidence gathering, reporting and ways of remediating any weaknesses, since this is the entire point of executing a penetration test.

There are various tools that can be used to perform pentesting, which include tools such as OWASP ZAP, Burp Suite or Metasploit.

Common Web Vulnerabilities

There are many known vulnerabilities when dealing with web security. Interestingly enough, many vulnerabilities have existed for many years, and are still easy to find in numerous websites[10].

- **Injection**: Usually means database injection such as SQL-injections. The problem can be that data is not handled properly, meaning that data, for example, will be appended directly on SQL commands, making it possible to escape the data, and write commands yourself directly through a input parameter. This can usually be handled by escaping special characters before using strings in a database. Injection can make it possible to breach most of the security goals (see chapter 2.3). Injection security flaws can make attackers gain access to data, corrupt or even gain access control rights.
- **Cross-site Scripting (XSS)**: Similar to injection, cross-site scripting refers to user data containing script content. Cross-site scripting occurs when data is not handled properly, and can affect both server or client. There are two types of this vulnerability: *Reflected XSS*, where the current input will be returned

[9] http://en.wikipedia.org/wiki/Penetration_test
[10] https://www.owasp.org/index.php/Top_10_2013-Top_10

with the response message. *Stored XSS* means that the script will be stored on the server and could potentially reoccur whenever this data is fetched. To solve this, data should once again be escaped properly by removing any special characters that could be used to invoke scripting, or encode the content as text, so the server will not interpret it as a script.

- **Broken Authentication and Session management**: Customized or custom made user management systems often has ways of exploitation or circumvention. If this is the case, an attacker can gain the privileges of one or more users of a site, thus being able to access what the users normally can. There is no easy way of preventing this. Instead, it is recommended to create a robust authentication system or use external well tested authentication systems.

- **Cross site request forgery (CSRF)**: CSRF attempts to exploit a users currently active session or authentication scheme. This is done by making a request through the user to an external site where the target currently have privileges.

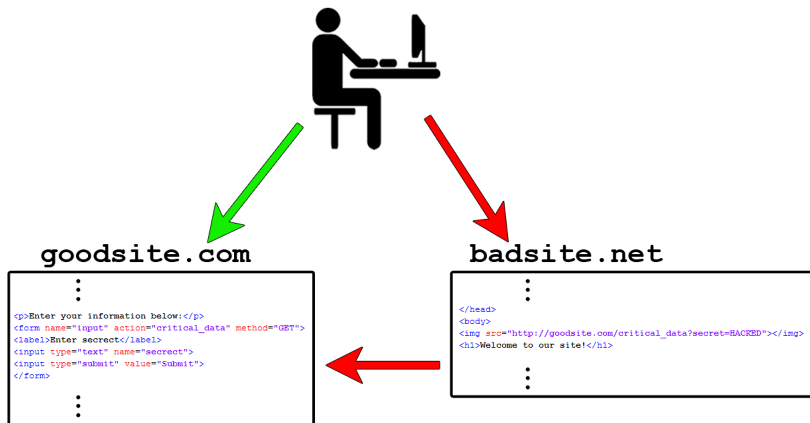


Figure 2.3: A sample CSRF attack

Figure 2.3 show an example CSRF attack. In this case the user has logged into a non-malicious site. A little later the user accesses another site, this site however contains a request for the previous site. Since the user recently have logged in this means that the request will be executed with that users privileges.

To avoid CSRF, sites can implement encrypted form tokens, often called anti-CSRF tokens. These tokens are unique nonces[11] and usually stored as a hidden form value that are sent together with a form request. Alternatively, they can be stored as cookie values that also need to be validated, once the server receives

[11] `http://en.wikipedia.org/wiki/Cryptographic_nonce`

a request. Another way of countering CSRF is to make sure that the user must continuously validate themselves, which could be by providing user-name and password or using a CAPTCHA field[12].

2.4 The Open Web Application Security Project

The Open Web Application Security Project, or OWASP, is an international organization, that focuses on improving the security of software. They describe their purpose as being the thriving global community that drives visibility and evolution in the safety and security of the world's software [13].

OWASP was founded in September 2001, at a time where the Internet was a very vulnerable place. It was created out of the need for a place for developers to freely document their experience with, and knowledge of, web application security [14].

OWASP is based in the United States of America, but has many regional and local chapters all around the world, including one in Denmark.

The organization has a four-point set of core values [15], that sums up what the OWASP community is about:

- **OPEN** Everything at OWASP is radically transparent from finances to code.
- **INNOVATION** OWASP encourages and supports innovation and experiments for solutions to software security challenges.
- **GLOBAL** Anyone around the world is encouraged to participate in the OWASP community.
- **INTEGRITY** OWASP is an honest and truthful, vendor neutral, global community.

Members of the OWASP community are divided into three different groups that define what their core focus points are, although some overlapping does occur. The groups are labeled as **Builders**, **Breakers** and **Defenders**. The idea of OWASP Communities is to bring together experts in the area that they are best at with the common goal of advancing the state of application security. This approach allows similar groups of professionals and experts to tackle security problems with the involvement of the most relevant stakeholders. The intent is to drive high quality output that is immediately usable by the target audience. This division is illustrated

[12] http://en.wikipedia.org/wiki/CAPTCHA
[13] http://www.owasp.org/index.php/About_OWASP#Core_Purpose
[14] http://blog.sourceclear.com/the-start-of-owasp-a-true-story/
[15] http://www.owasp.org/index.php/About_OWASP#Core_Values

in Figure 2.4[16].

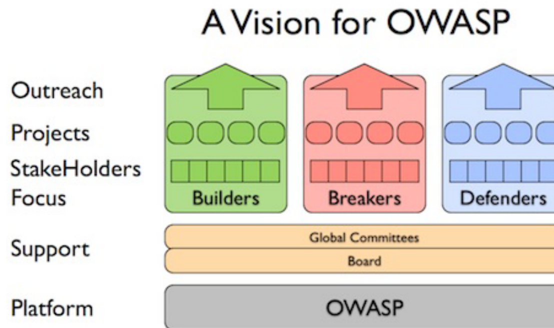


Figure 2.4: The division of the OWASP Community groups.

OWASP Projects

A lot of different projects are under the OWASP brand, most of which are related to developing security software. Projects fall into three different categories [17];

- **Flagship Projects**: The OWASP Flagship designation is given to projects that have demonstrated strategic value to OWASP and application security as a whole.
- **Lab Projects**: OWASP Labs projects represent projects that have produced an OWASP reviewed deliverable of value.
- **Incubator Projects**: OWASP Incubator projects represent the experimental playground where projects are still being fleshed out, ideas are still being proven, and development is still underway.

Projects usually start out as Incubator Projects, where ideas are shared, experiments are carried out, and Proof-of-Concept software is developed. Projects are regularly evaluated, and if proven valuable they are promoted to Lab Projects. Some projects have been abandoned, while others have merged with or forked into other OWASP projects, e.g. the **OWASP DirBuster Project**[18]. Flagship Projects are intended to be projects that have matured well and produced useful and valuable content. The OWASP website states that projects are currently being evaluated and

[16] http://www.owasp.org/index.php/File:OWASP-vision.jpg
[17] http://www.owasp.org/index.php/Category:OWASP_Project
[18] http://www.owasp.org/index.php/Category:OWASP_DirBuster_Project

the list of Flagship Projects should be updated by Mid-August 2014, but at the time of this writing, it is currently empty[19].

Project examples

One of the mentionable Lab Projects is the **OWASP Broken Web Application Project**[20]; A collection of purposely vulnerable web applications. They are designed to educate about security flaws and how to prevent them. Each web application is packaged as Virtual Machines, so they are easy to set up.

Another prominent project is **OWASP Mantra**[21], a web browser designed specifically for testing security in web applications. It provides ability to modify HTTP message headers directly in the browser, and other functionality that would normally require other tools or browser extensions.

OWASP also has a project called **The OWASP Top Ten**[22]. The project releases a list of vulnerabilities of what represents a broad consensus about what the most critical security flaws in web applications are. The primary aim of the OWASP Top Ten is to educate developers, designers, architects, managers, and organizations about the consequences of the most important web application security weaknesses.

2.5 The Zed Attack Proxy Project

The *Zed Attack Proxy*, or ZAP, is a project under OWASP. It brands itself as an easy to use integrated penetration testing tool for finding vulnerabilities in web applications[23]. It is one of the most active OWASP projects, and has a very active community for both developers and users.

In the early 2000's, an application called Paros Proxy was developed, which was a popular tool for intercepting HTTP messages, spidering (or Web Crawling[24]) websites, and scanning for vulnerabilities in web applications. Unfortunately, the developers of Paros Proxy stopped maintaining the application around 2006[25]. ZAP v.1 was released in 2010, as a fork of Paros Proxy, extending the functionality and enhancing the user interface. Since its initial release, ZAP has become very popular, and is even included in some distributions of Linux[26].

[19] http://www.owasp.org/index.php/OWASP_Project_Inventory#Flagship_Projects
[20] http://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project
[21] http://www.owasp.org/index.php/OWASP_Mantra_-_Security_Framework
[22] http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[23] http://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
[24] http://en.wikipedia.org/wiki/Web_crawler
[25] http://sourceforge.net/projects/paros/files/Paros/
[26] http://www.kali.org/

The Paros Proxy project was abandoned, but the founders of ZAP made sure to establish a solid community. There are very active forums for ZAP, aimed at both the users and the developers of ZAP. The community encourages anyone who is interested in contributing to the development of ZAP to do so, and participates in student programs like **Google Summer of Code**[27] and **Mozilla Winter of Security**[28].

ZAP Features

The main features of ZAP are to a large extent the same as the features of Paros Proxy; serving as an intercepting proxy, for examining HTTP messages, spidering websites to discover URL's, and performing active and passive scans on web applications to report vulnerabilities. However, as web application platforms evolve, new vulnerabilities are introduced, or old security flaws are discovered (e.g. HeartBleed[29]). The tools that are designed to discover these vulnerabilities need to stay up-to-date with techniques on how to reliably discover and report any weaknesses in the security of a web application, and hence ZAP is constantly evolving to adapt to the circumstances.

One of the major changes in ZAP is the introduction of scripts. Scripts in ZAP can be defined in different widely used languages (e.g. JavaScript, Python, etc.), and can be used for different purposes. Scripts defined as Active Rules or Passive Rules will be included when performing Active or Passive scans respectively, while Authentication Scripts provide an easy way to perform a login request. Proxy scripts are run on each request proxied through ZAP, which can be useful for identifying specific patterns in the headers or body of HTTP messages.

Another useful addition to ZAP, is the handling of sessions and authentication. This enables the user of ZAP to quickly switch between sessions and users for a particular web application, or force all requests to use a specific session and/or user.

In recent versions of ZAP, a new concept has been introduced, called Contexts. Previously, ZAP could be configured to operate on specific websites, based on their URLs. A context aims at defining an entire web application, by maintaining a collection of URLs, users, authentication methods, and other information related to a single web application. ZAP can then be configured to operate using the specified information in Context.

ZAP has also been extended to include an API. This means that ZAP can also run as a daemon, with no graphical user interface, but through commands through a terminal, and via REST HTTP messages. This in turn means that ZAP can be

[27] `https://www.google-melange.com/gsoc/homepage/google/gsoc2014`
[28] `https://wiki.mozilla.org/Security/Automation/WinterOfSecurity2014`
[29] `http://en.wikipedia.org/wiki/Heartbleed`

configured to run in an environment with Continuous Integration[30] and perform regression tests[31].

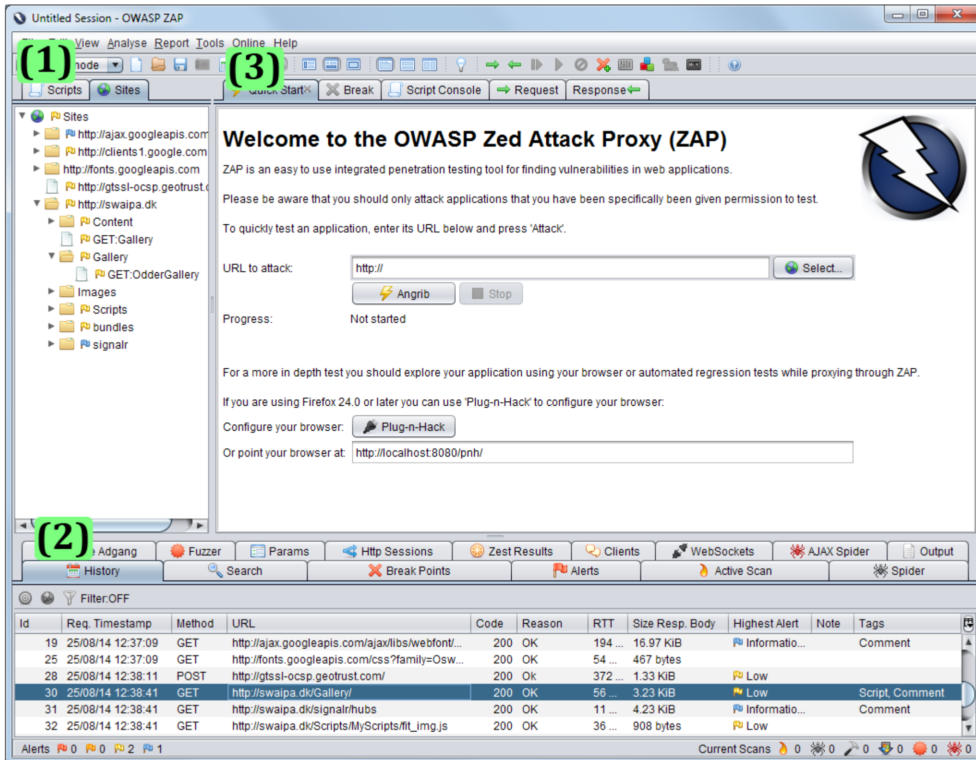


Figure 2.5: A screenshot of ZAP, after visiting a few sites.

Figure 2.5 shows a screenshot of ZAP when the program has just been started and a few sites have been visited. The main screen is divided into three different sections, each containing a set of tabs. To the left at **(1)** is the sites tree and script tabs. Each visited page shows up in the Sites Tree as hierarchical nodes. Different sites can be filtered out, by creating a context and setting the Sites Tree to only show nodes related to that context. The Scripts tab shows all loaded scripts, and any script templates.

At section **(2)**, a lot of tabs are grouped together. ZAP starts on the **History tab**, which shows a table with all the pages visited during a ZAP session. Some

[30]`http://en.wikipedia.org/wiki/Continuous_integration`
[31]`http://en.wikipedia.org/wiki/Regression_testing`

other prominent tabs are the **Active Scan** and **Alerts** tabs. The **Active Scan** tab shows the progress of an active scan, and it is possible to monitor which scan rules are performing their attacks and when. If a vulnerability is found during an active scan, an alert will be raised, and visible on the Alerts tab.

The remaining section **(3)**, has the tabs **Quick Start**, **Request**, **Response**, **Break** and **Scripts Console**. From the **Quick Start** tab, it is possible to initiate an active scan or setup Plug-n-Hack, which is a feature for interacting with ZAP directly through a browser. If a webpage is selected in the Sites Tree or on the History tab, the request and response messages can be seen on the **Request** and **Response** tabs respectively. The **Scripts Console** shows the output of any scripts that are run in ZAP. Some scripts do not produce any output, but raise an alert to notify the user of the result of running the script.

Scanning with ZAP

ZAP can perform its scans on entire web applications, or simply on a single HTTP message. Scans are intended to reveal any vulnerabilities or security flaws, or simply to inform that the site reveals some kind of information about its setup. Scan Rules are defined as different types, indicating what the main purpose of the rule is, or how the attack is performed. The categories are Information gathering, Client browser, Server security, Injection and Miscellaneous. Scans are executed in two different manners, depending on how they search for vulnerabilities; Passive Scans and Active Scans.

Passive Scans

In ZAP, Passive Scans are performed each time a HTTP Message response is received. Passive Scans do not alter the messages in any way and do not send any additional HTTP messages, and are in that sense safe to use. Passive Scans only inspect the messages to find certain patterns or tokens, that might contains vulnerabilities to the web application.

All Passive Scans are by default enabled, but can be configured to suit specific setups. Passive Scan Rules can also be created within ZAP, and do not require compiling or building new files. Passive Scans also adds tags to the HTTP Requests, that are visible in the History tab in ZAP, so it is possible to see which specific request has a potential security flaw.

Active Scans

Active Scans are issued to perform known attacks using HTTP messages, but the messages are usually modified and sent to the server again, to search for a specific

vulnerability in the response. In that sense, Active Scans are not safe to execute, and should only be performed on Web Applications where you are given permission to do so.

Active Scan Rules are typically more elaborate than Passive Scan Rules, as the flaws they aim at exposing require advanced techniques. Therefore Active Scan Rules will normally require some coding, compiling and building of classes to include in ZAP, in order to use them. However, it is also possible to define an Active Scan Rule as a script, directly in ZAP.

If you want to implement an Active Scan Rule, you can create a class that extends from a class called **AbstractPlugin**. The superclass provides all the relevant methods to use during a scan, but the subclasses are handled very differently;

- **AbstractHostPlugin** - Only scans once for every host. This is useful for finding vulnerabilities that are not page-specific, such as the HeartBleed vulnerability.
- **AbstractAppPlugin** - Scans every page in an attack. This attack is useful for finding vulnerabilities in parameters that are not already specified.
- **AbstractAppParamPlugin** - Scans every parameter in a request. This is useful for testing if some parameters are vulnerable to injection attacks.

Active scans can be initiated in several ways. On the startup screen, a tab called Quick Scan enables the user to enter a URL to scan. When a Quick Scan is executed, the site is initally spidered, to find all pages on the site. When the spider has finished crawling the site, the scan starts.

An active scan can also be initiated by right-clicking on a site or page in ZAP, either in the Sites Tree or on the History tab, and selecting Attack in the context menu, and then selecting any of the preferred ways to attack. You can choose to scan the single page selected, all of its sub-pages, or all pages. The context menu also allows for starting scans using the Advanced Active Scan Dialog. From here, it is possible to tweak all the parameters of the active scan, which is very useful for setting up site or page specific scans.

CHAPTER 3

Analysis

In the previous chapter, background information about this project and its core concepts were presented. This chapter will elaborate on the concepts, and give an analysis of the ZAP application. The chapter begins with an description of some scenarios, where scanning with sequences may be relevant.

The following sections will investigate the architecture of the ZAP codebase, some of the core functionality of ZAP, and provide an analysis of the User Experience using ZAP. After this, an analysis of the target audience of this project will be presented. The results of these sections will be presented in the Requirements section, and an indication of the priorities of each requirement will be described. The chapter concludes with a description of the testing strategy for this project.

3.1 Scenarios

In this section, a few scenarios will be presented, where a vulnerability may not be discovered by ZAP, because it is not yet possible to handle sequences of HTTP requests.

ZAP treats each request as stand-alone, meaning there is no prior knowledge about what actions the user has performed in the web application. Even if there is a request, where the user submits some information, the data may not be persisted until further actions have been performed. Furthermore, a user could use ZAP as a proxy, and reach a certain page in the web application, but this page may only be accessible to users that have performed some action immediately before visiting the page. If a user then attempts to scan the specific page through ZAP, the attack-request may be rejected, as the previous action was not performed.

Vulnerabilities that can be found using sequences often fall between two chairs, as it is not a new type of vulnerability that will be exposed. It is the same known vulnerabilities, that can be exposed by performing a specific set of actions before executing the actual attack. At the same time, it is not explicitly a logic flaw in a web application either. So vulnerabilities that can be uncovered using sequences fall into a category between logic flaws and known vulnerabilites.

Conceptual Examples

A typical scenario where sequences will be useful, is in cases of persistent cross-site scripting (XSS). This is usually the case when a user submits some data, but the data is not persisted until several steps later. A common example is on web forums, where it is possible to review a post, before actually submitting it. Figure 3.1 shows an example of this, where a post is written, containing some JavaScript. When submitted to the preview page, the JavaScript is simply shown as text. However, when finally submitted and persisted, the JavaScript will be run for every user that visits the forum post.

The example script simply displays an alert-box. While this is harmless to the users, it can be used as a primer to detect if sites are vulnerable to embedment of scripts. This is also how ZAP detects if persistent XSS is possible. However, ZAP will attempt to inject a value, when submitting for preview. This means that although the JavaScript was correctly posted, it is never persisted, because ZAP does not by itself submit the previewed page.

Similar examples include Webshops, where the payment process is usually spread across several posts, and online surveys, where a lot of informations is collected, before being presented to the user for final submission. If these examples contain vulnerabilities, they will most likely not pose a threat to other users, since these pages are not directly available to these users. However, they could reveal information about the servers setup, that a hacker could potentially exploit.

Real World Examples

There are many real world examples[1] of XSS attacks. Although not all of these examples would have benefitted from being able to scan sequences of requests, some of them surely would. The MySpace *Samy Worm*[2] for example, required multiple requests to complete its actions, by automatically sending "friend requests" when users were logged in.

The issue that initiated the concepts of sequences in ZAP, originated from a post on the ZAP Forum[3], where a user could not detect a vulnerability in a web application called WackoPicko[4], which is an intentionally vulnerable web application. WackoPicko contains a wide variety of vulnerabilities, and is also part of the OWASP Broken Web Applications project. WackoPicko was created for a research paper, that

[1] `http://www.xssed.com/xssinfo#Real-world_examples`
[2] `http://namb.la/popular/tech.html`
[3] `https://groups.google.com/d/msg/zaproxy-users/apr9GnZLQM0/dTEv0rLubGAJ`
[4] `http://github.com/adamdoupe/WackoPicko`

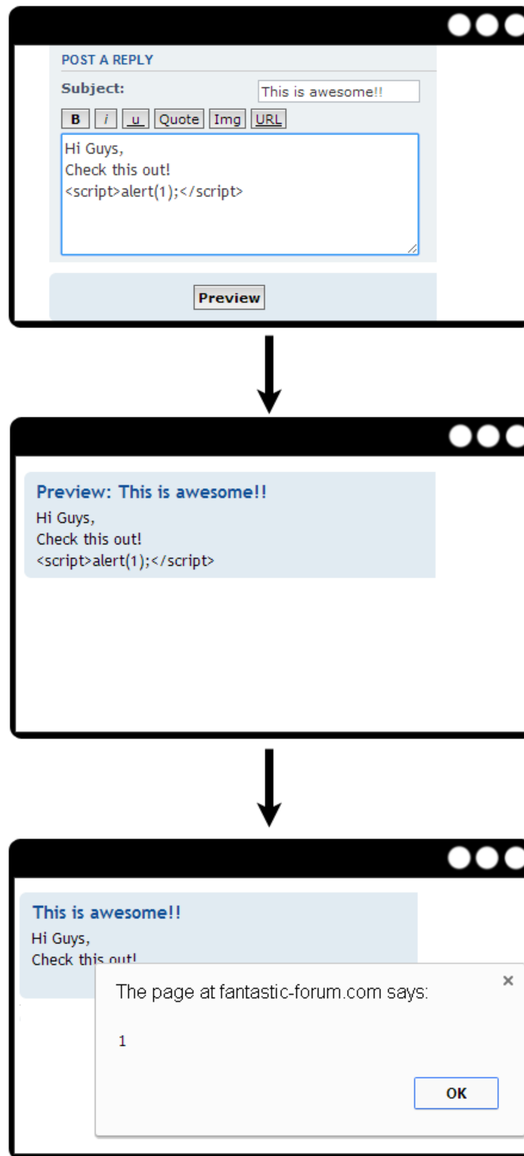


Figure 3.1: An example scenario, where sequences can be used to discover a vulnerability (in this case, Persistent XSS).

investigates the strength of different pentesting tools [DCV10].

The issue in WackoPicko is that a user can write a comment on a picture, submit it for reviewing, and then finally submit it again for persisting. If the comment contains an embedded script, the script will not run on the preview page, but it will run on the actual post, once the comment is persisted.

3.2 ZAP Workspace

One of the main concepts behind all OWASP projects is the concept of transparency. This also means that all of their projects are developed as open source projects. Developers who want to help can freely do so by submitting updates to a repository or similar. However any committed updates must be approved by leading developers higher in the project hierarchy.

ZAP is developed in Java and is structured in such a way that both the source-code[5], stored on a SVN-repository, and a full Eclipse workspace[6] is available to download. Both are currently stored on google code, but since storing space here is limited there have been suggestions of moving elsewhere such as GitHub in the future. If developers want to commit any updates or changes however they will need to request commit-rights from existing ZAP developers. Naturally the ZAP released version (Executable JAR) is available to download from several locations.

ZAP is a fork of the Paros Proxy project[7] which has much of the same basic functionality, however Parox Proxy is no longer maintained. ZAP is structured in such a way that it has kept almost all of the Paros Proxy core and ZAP has then been build on top of this by adding or extending content. The general guideline for handling the legacy Paros project code has been to attempt to change this as little as possible, however it has not been completely possible to avoid changes. From a coding point of view any changes to the Paros core needs to be well-documented both as comments at the relevant location, but also in the comment-header stating the date and reason for changing or adding functionality. This is a requirement from the Paros Proxy license, but it also makes sure that any changes are well-founded.

A feature that ZAP recently has emphasized is the ability to create extensions. This means that users can install and update extensions independently of the current ZAP version, through the interface. By doing this, extensions can be updated much more frequently and will not have to wait for a complete ZAP update. Another advantage is that the downloadable version of ZAP does not have to include all extensions by default making it slightly smaller in size.

[5] https://code.google.com/p/zaproxy/source/checkout
[6] https://code.google.com/p/zaproxy/wiki/Downloads?tm=2#ZAP_Workspace
[7] http://sourceforge.net/projects/paros/

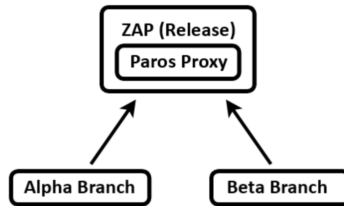


Figure 3.2: The ZAP workspace structure

Figure 3.2 shows a general view of how ZAP is structured. As mentioned previously ZAP contains the forked code from Paros Proxy. Further more ZAP has been divided into an alpha and beta branch. These branches have been created so that extensions that are not completely finished can be used without being mixed with completed ones. Extensions start in the alpha branch, meaning that they are in a very early stage of development. Then they will become part of the beta branch, indicating that they are close, but not quite ready, to a final release.

3.3 ZAP Functionality

As previously stated ZAP is built on the older (unmaintained) Paros Proxy project. Some of ZAPs functionality originates from what was part of Paros. However many features have been added and most of the original features have been heavily enhanced.

Contexts

A relatively new concept in ZAP is the ability to define contexts. Context are meant to be a way of grouping related URLs. Ideally context should represent a specific web application that the user wants to test, but this is not a requirement. Contexts are defined by regular expressions that tells what URLs a given context does and does not cover. What makes contexts unique is the ability to define various settings. One of these settings is that it is possible to define both users and a login-method; this could for instance be that a given site needs a user to login via a traditional form where a user can provide a username and password. In the context it would then be necessary to set the current authentication style to form-based, and then input the login-page URL together with what parameter that represent a username and password respectively. At the same time the user can inform ZAP what to look for indicating whether the current user is logged in or not by inputting a string with a regular expression. For instance, if a site contains the word ”login”, it could be likely that the site wants the user to login, and oppositely with a ”logout” string. Contexts contain other options such as setting the current session-management scheme.

Active Scanning

One of the most unique and important features of ZAP is the ability to run a full penetration test. This can be done without any knowledge of how to hack or exploit vulnerabilities. There are several different ways an active scan can be initiated. ZAP does provide a quick scan option which will use a user provided URL to start a spider and afterwards perform an active scan on the found site-tree. However this is mostly provided for new users, and it is usually expected that active scans are started through either the site-tree (located on the left side of the screen) or the history tab (in the button). In order to populate the site-tree users normally proxy a browser through ZAP meaning that everything that the browser discovers is also input into ZAP. Alternatively it is also possible to define a context, and then run a spider separately on this context adding all sites the spider explores.

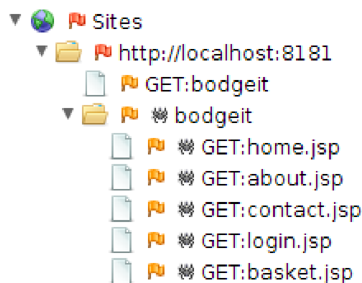


Figure 3.3: The ZAP site-tree

A site-tree is, as the name suggests, a tree of HTTP messages that ZAP knows of. Figure 3.3 shows how a basic site-tree looks like, it is structured with the host on top. All sub-paths are then added as nodes below the host or top-node. At the lowest level are the actual HTTP messages. The reason for this structure is both to group related HTTP messages and thus related web functionality, but also to make sure that web applications can be tested individually, since these web applications ,in some cases, can be hosted on the same host. ZAP provides the possibility so save the current session, so the site-tree and history can be reloaded at a later time.

An active scan can be invoked by right-clicking any node in the site-tree, or in the history reference tab. Most nuances of active scans are related to selecting a starting node to scan. However, it possible to open an advanced scan dialog, shown in Figure 3.4. From this dialog, you can customize the scan considerably. The Scope tab defines a context, start node and a possible user. Furthermore the input vector tab lets the user define exactly what the scanner should attack. Policy is a list of plugins that are run during an active scan.

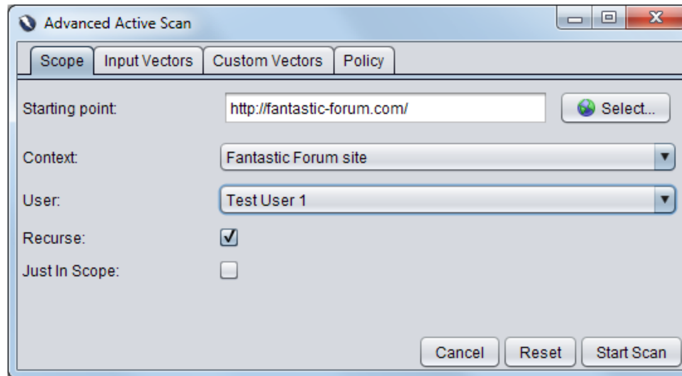


Figure 3.4: The advanced scan dialog

When a scanning is started ZAP essentially starts the scanner originally created for Paros Proxy. The scanner will start a thread for each host (*HostProcess*). Each host-process will explore the remaining site-tree (the tree for each host), and simultaneously loop through every plugin. A plugin is essentially a term describing a class designed to find a specific vulnerability. Plugins can be one of three main types, **AbstractHostPlugin** which is run for ever host, **AbstractAppPlugin** which is run for every site, and **AbstractAppParamPlugin** which is run for ever parameter for every site. When a plugin finds a vulnerability, it raises an alert in the alert-tab. The alert is presented with a risk-assessment indicating the risk severity of the vulnerability but also information of where it was located and how such a vulnerability could be addressed.

Passive Scanning

Another way of scanning web sites is the ZAP passive scanner. In contrast to the active scanner the passive scanner does not actively try to inject or invoke anything on a web site. Everything is handled in a background thread that monitors every HTTP message that ZAP spiders or proxy. That way the passive scanner can analyze a HTTP message and thus look for any known vulnerabilities that would show just by viewing the site content available in the client. Passive scan plugins are classes that implement the **PluginPassiveScanner**, this makes the plugin implement two methods, one that is called for every HTTP message that is sent and one for every receive. Examples of vulnerabilities that are possible to find through the passive scanner could be related to header information for cookie settings, or inclusion of scripts located on another domain.

Scripting

In ZAP, scripts are created on the scripts tab, located on the left side of the main window. If a script produces any output, it will be shown in the **Scripts Console**, which is a tab, grouped together with the **Request** and **Response** tabs. The tab also shows the contents of the script. Some scripts can be edited directly on this tab, while others require interaction through context menus on the Scripts tab.

Scripts can be embedded within ZAP, and access some of the data structures used in ZAP, or execute certain functionality in ZAP. A number of scripting languages are supported, including JavaScript, Groovy, Ruby and Python. More can be added, as long as they support *JSR 223*[8].

ZAP supports a set of different script types. These types are not related to a scripting language. Instead, they define a specific function in ZAP, however some types are only available for specific languages. Scripts that are of the type *Proxy Rule* will run on each request that is proxied through ZAP. This can be useful for modifying requests before sending, or searching for specific tokens or patterns in the response of the request. Another script type *Script Input Vector* lets the user define exactly what parameters ZAP should attack, during an active scan. There are also script types for defining *Active Rules* and *Passive Rules*, that will be run during Active and Passive scans. *Standalone* scripts can be run independently of any ZAP actions. A full overview of the scripting languages, the script types available and their usage, can be found at the ZAP help pages for the scripting add on[9].

Figure 3.5 shows a screenshot of the Scripts tab, where a few scripts have been created, and the Scripts console, where the output of any scripts is shown. In the screenshot, two scripts have been loaded. One is a proxy script that investigates whether a response contains a 'set-cookie', and prints a message in the console if that is the case. The other script is a stand-alone script that performs the same action each time it is run. In this case, it would add an item to the shopping cart of a specific webshop.

Zest Scripting Language

Zest is a new experimental scripting language, created for the purpose of aiding web security testers and developers of web applications. Zest is a project by Mozilla Security Team[10], and developed as Open Source, so anyone interested in contributing can do so. The creator of Zest is Simon Bennetts, who is also the Project Lead for ZAP. While Zest is definitely developed with ZAP in mind, it is intended to be independent,

[8] `https://www.jcp.org/en/jsr/detail?id=223`
[9] `https://code.google.com/p/zaproxy/wiki/HelpAddonsScriptsScripts`
[10] `http://developer.mozilla.org/en-US/docs/Zest`

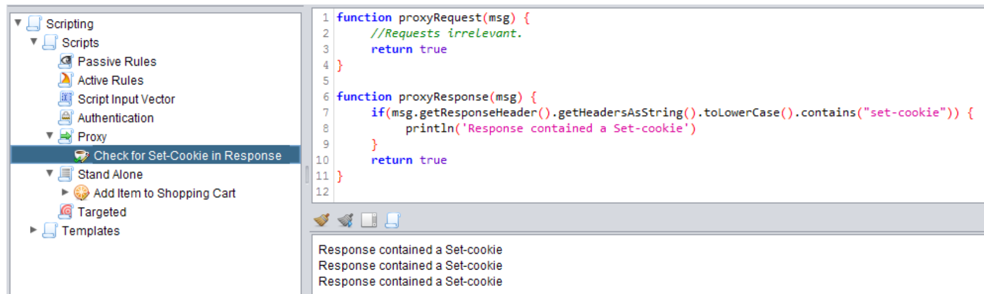


Figure 3.5: The Scripts tab and Scripts console, with a few loaded scripts.

and is also proposed as a standard way of reporting vulnerabilities and bugs in web applications, as they are usually difficult to reproduce from textual descriptions only.

The contents of Zest scripts are in JSON[11] format, but Zest scripts are not meant to be written using text editors like most other scripts. Instead, Zest is designed to be a visual language. Zest scripts are created and modified through a graphical user interface (GUI) in the tools and applications that support Zest scripts. The graphical representation of Zest scripts is not handled by the language itself. Applications that integrate Zest, must define a way of presenting the scripts contents in the GUI.

Zest is integrated in ZAP, and most of the existing script types are supported by Zest. Internal functions of ZAP, such as starting an Active Scan, can also be invoked from a Zest script. Figure 3.6 shows a screenshot from ZAP, where a Zest script has been created. The script in the screenshot contains a few GET and POST requests, two assignments to Zest variables using values from a form on one of the pages, and a call to start an Active Scan on one of the HTTP messages. The script concludes with a print statement, that will be shown in the **Scripts Console** of ZAP. By default, ZAP adds assertions to HTTP requests in Zest scripts. These are used to validate that the response have the expected HTTP Status code or that the content is of a specific length.

Zest is not an expansion of the existing scripting mechanism of ZAP. It is a separate extension altogether, and as such, has to define how to handle the different script types of ZAP. This is done by creating a runner for each Script Type that Zest will support. The language contains a class called **ZestBasicRunner**, which has all the required functionality to run Zest scripts. ZAP adds a layer, by extending this class with a **ZestZapRunner** class. The primary objective for this class is to react to actions from Zest scripts, and update other parts of ZAP. An example of this, is if

[11] `http://json.org/`

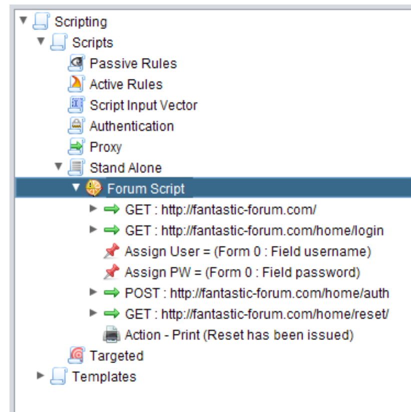


Figure 3.6: A Zest script called Forum Script created in ZAP.

an assertion in a Zest script fails, a notification of this will be sent to the Zest tab in the ZAP GUI.

For Zest to support specific script types in ZAP, a runner for the type must be created. For instance, **ZestProxyRunner** handles scripts of the type *Proxy Script* in Zest format. A runner can simply pass the entire script to the Zest scripting engine for execution by overriding the **run** method which takes a **ZestScript** as an argument, but it is also possible to execute every statement of the script one by one, by overriding the **runStatement** method, and passing each individual **ZestStatement** to the method.

Extensions

Extensions and plugins can be downloaded and installed directly through the ZAP interface. Extensions are developed as a separate Java package and are ideally self-contained. However they are limited in implementation to where ZAP provides mechanisms to hook or inject functionality into.

An extension package can be in one of three states, either alpha, beta or release. This indicates how mature a given extension is. Each extension contains a **ZapAddOn.xml** file which informs ZAP of certain values before loading it. This includes extension name, version and description, but also definitions of where to locate the main extension file and any relevant dependencies may be present. Also the extension may define which versions of ZAP it will work together with. Extensions are built using Apache Ant[12], which builds an extension as a *.ZAP* file (which essentially is a renamed .zip file). Extensions that are to be loaded in the current running

[12] `http://ant.apache.org/`

ZAP execution should be located in the *plugin* directory.

Each extension package must contain a file with name beginning with 'extension' followed by the name of the current extension, for instance **extensionSpider**. The main extension file must inherit the **ExtensionAdaptor** class. By doing this the extension will be loaded during start-up. The **ExtensionAdaptor** simultaneously contains a **hook** method which also will be run at a later stage in the start-up process. This method provides access to an **ExtensionHook** class, which in turn provides various ways of hooking into other parts of ZAP. This includes hooking into various parts of the interface, but also other functionality such as adding a listener class for predefined actions.

Another way for extensions to interact with other parts of the program is to load other extensions using the **ExtensionLoader** class, which can be acquired through the static **Control** class. This will return a reference to the primary extension class. Interaction with this class limited to functionality that has been implemented in the individual extension.

3.4 ZAP User Experience

One of the main selling points of ZAP is that ease of use is a priority[13]. But how does this come into effect in the ZAP user experience design? The primary idea behind user experience is to make interactions easy for the user, while also making everything useful, every functionality and graphical element need to come together and work well with each other. How this is achieved however is very debatable and is often very dependent of the individual users point of view [Gar10]. Every interface wants to provide users with some kind of functionality and at the same time provide a consistent look and feel. ZAP uses many default interface elements from Java and has inherited its overall application structure from the Paros Proxy tool, which can be seen in Figure 3.7.

Compared to ZAP, Paros had much less functionality, which is also reflected in what is displayed in the main application window. However the Paros frame still serve as the main structure of the ZAP interface, which contains a main site-tree, a tab-panel for HTTP requests and responses and finally a tabular container for information provided by Paros/ZAP. The structure does somewhat align itself to overall structural guidelines such as the gestalt laws[14]. The gestalt laws are a set of general rules for how users perceive an interface. The primary elements are proximity, similarity, continuity and closure, which essentially addresses placement of related functionality. In this case the tabs on the top tab-panel are all concerned with informing the user of the content of http-messages, where the button tab-panel contains all additional

[13] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project#tab=Features

[14] http://en.wikipedia.org/wiki/Principles_of_grouping

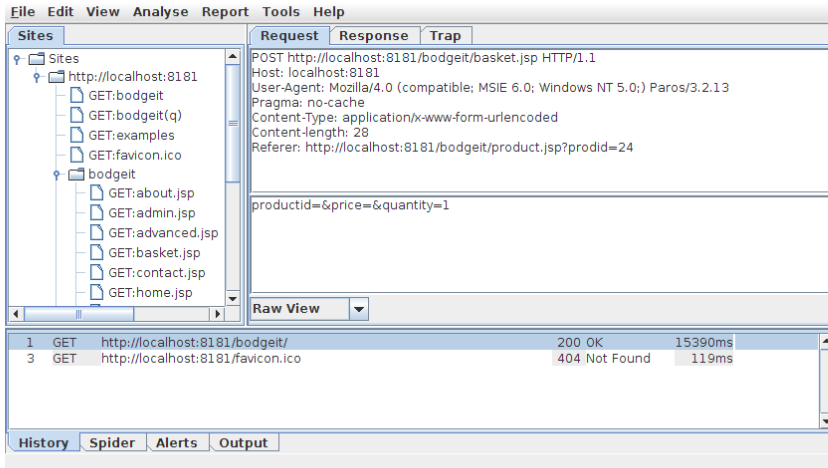


Figure 3.7: Screenshot of the Paros Proxy tool

information regarding those messages. Once users are aware of this connection the program is likely easier to use, however from a visual point of view there is nothing to indicate this relationship.

Since forking Paros, ZAP has received numerous updates and thus many new features. However ZAP still uses the same structure as Paros did. While Paros had a limited amount of information it needed to display for the user, ZAP has much more. As a new user attempting to use ZAP this could prove confusing and might even discourage some. While ZAP does provide a quick scan tab, it is almost a requirement to read a tutorial or the provided documentation in order to use it properly.

One way of designing user-experience is by having a set of scenarios representing a specific action a user would perform) together with a target audience, also called user-stories. Then based on a user-story a certain functionality can be designed and implemented. However ZAP's approach is mainly to function as a security tool that tries to achieve a balance between providing advanced features together with ease of use, which are two goals that are slightly contradictory. ZAPs target audience varies between new users that are inexperienced with testing web-applications to professional penetration testers, which is a very large spectrum of possible users. Regardless, when looking at the actual ZAP interface all functionality, even advanced, is displayed on the main page. This seems to indicate that more advanced users, such as pentesters, are more targeted than less experienced users, as new users may be overwhelmed by the many options available right from the main window. The ZAP community has recognized this and is now providing a much lighter version called ZAP core, which is

more targeted towards newer users[15]. This version is a smaller version of ZAP where some of the more advanced features have been removed.

Even though ZAP does have a steep learning curve it does provide many advanced features without hiding them. For a pentester that knows how to use ZAP it is undoubtedly a useful tool, and the interface will not attempt to hinder the user in any way, thus providing a good user experience for advanced users.

3.5 Target Audience

In this section a list of personas will be presented, indicating different types of users of ZAP. Using these personas, a number of user-stories will be derived. These will describe the benefits each persona may have, from a mechanism for handling sequences in ZAP. These user-stories help shape the requirements of the sequence mechanism.

Personas

Table 3.8 shows a persona, Alice, who works with penetration testing. Her requirements for a pentesting tool include that it produces reliable results and is easy to configure. Table 3.9 and table 3.10 show two personas, Bob and Carol, that are related, in that they deal with web applications. However, Bob is the creator of web applications, and wants to deliver a secure product, while Carol as a web-master wants to ensure that the web application is still secure, even after upgrading server hardware or software. The final persona shown in table Table 3.11, is a software developer, that wants to extend the functionality of ZAP.

A few other personas were considered. One of these were a Student persona, that wanted to learn about web security and vulnerabilities. However, the functionality that this project will introduce, will most likely be for advanced users. Furthermore, the functionality does not aim at discovering a new type of vulnerability, but rather expose those that are already known. A student would benefit more from learning how known vulnerabilities can be discovered, before advancing to find out how they can occur in sequences.

A hacker persona was also considered. Hackers would likely use ZAP to discover vulnerabilities in web applications. Whatever their intentions might be, their requirements to the functionality created in this project, will likely be covered by the other personas, that have genuinely good intentions.

[15] https://code.google.com/p/zaproxy/wiki/Downloads?tm=2#ZAP_2.3.1_Core

Name	Alice
Role	Pentester
Motivation	Confirm that company software is secure. If not; be able to find specific weaknesses.
Usage	• Alice wants an easy-to-use system where she can get precise reports of security problems in tested applications. • Doesn't really want to spend too much time setting up the system and would like to save tests for later reuse. • Expects ZAP to work together with other testing software.

Table 3.8: A persona called Alice.

Name	Bob
Role	Web-developer
Motivation	Creates web-applications and wants to be sure that it is secure before release.
Usage	• Bob want to be able to quickly get an overview of any security problems that his application might have. • If any security problems are encountered Bob wants to know as precisely as possible what problems there are and where they are found.

Table 3.9: A persona called Bob.

Name	Carol
Role	Web-master
Motivation	Maintains a web-site or a web application, and wants to make sure that security is sufficient.
Usage	• Uses OWASP ZAP as a tool to run a few times to find any security flaws. • Want to get an overview of how severe any problems might be. • Wants to be able to very easily test all of a website with minimum personal effort.

Table 3.10: A persona called Carol.

Name	Dave
Role	ZAP-developer
Motivation	Wants to expand and further develop the OWASP ZAP tool as part of the open source development community.
Usage	• Develops new functionality of the ZAP tool using the Java project files publicly available online.
	• Expect different parts of the code base to follow the same overall guidelines.
	• Wants access to previous created content, and wants to easily be able to extend its functionality.

Table 3.11: A persona called Dave.

User Stories

Using the information obtained from the personas, a collection of user stories have been created.

- As a user of ZAP I want to be able to test sequences of requests, where output is dependent of input, like web based wizards and large input forms etc.
- As a pentester I want to be able to save tested sequences and be able to easily run them at a later time.
- As a pentester I want any ZAP based extensions (as with a sequence extension) to be able to work with external tools and be part of a larger testing scenario. (Support the ZAP API system)
- As a pentester I expect to be able to easily use and understand scripted tests without having to learn any specific scripting language.
- As a Web-Developer I want to use an application that can explore an web application and present an orderly list of problems together with suggestions of where they are found and how to fix them.
- As a ZAP-Developer I want to be able to quickly understand the code of extensions and simultaneously easily be able to access and extend functionality of these.
- As a Web-master or administrator I want to be able to very quickly test my site for any severe security problems with a minimal effort.
- As a Web-master I want get an overview of the overall security state of my web-application.

By creating these user stories, a set of user-related requirements to this project have described. These will be taken into consideration, while forming the actual requirements for the development process.

3.6 Requirements

Based on the observations found in the previous sections of this chapter, a set of requirements can be derived. These will serve as guidelines for the Design and Implementation phases of this project. The requirements are categorized as either *Must have* or *Nice to have*. The former category has requirements that must be reached, for this sequence mechanism to be implemented successfully. The latter category contains requirements that the sequence mechanism would greatly benefit from having in place, but are not necessary for the sequence mechanism to function as intended.

Functional Requirements

This section describes the requirements related to realizing the sequence mechanism, in terms of what it is supposed to be able to do.

Basic Functionality

With the sequence mechanism in ZAP, it should be possible to define a sequence of HTTP requests, and save it as a Zest script. When performing an active scan, it should be possible to scan any sequences that are defined in ZAP. If a vulnerability is discovered during an active scan, and it was during a scan of a sequence, it should be noted on the **Alerts** tab, that the vulnerability was found during a scan of a sequence. This is to differentiate between other vulnerabilities that were discovered while not scanning a sequence.

API Functionality

ZAP has a GUI that presents all the functionality and scan results. Most of the core functions of ZAP is also available via an API. The functionality of this extension should also be available via the API. Defining scripts may prove to be difficult through a command line terminal, but loading an existing sequence should be possible.

Non-Functional Requirements

This section describes the requirements related to other aspects, than the actual functionality. Although the goal is to achieve a functional sequence mechanism, these requirements are not unimportant and should not be neglected when creating new functionality for ZAP.

Deployment

ZAP already has a mechanism for handling feature extensions. Each extension is compiled to a file with the **.zap** file extension. These .zap-files can then be loaded independently. This is useful, if a user wants to only use the core functionality of ZAP, and avoid the UI being cluttered with too many things to keep track of.

Documentation

ZAP and its extensions are well documented. The Java code contains comments, and help-descriptions for a lot of the functionality is present, both in the application and online. The sequence mechanism should adhere to the same documentation practice.

Maintainability

ZAP is an Open Source project, and anyone can download the codebase from their repository and inspect it. To new developers, this is useful as the particular coding-style of the project can be viewed and adapted, before starting to work on extending the functionality of ZAP. The sequence mechanism created in this project needs to be well written, so that it can be reviewed by other ZAP developers, or if it ever needs to be altered or extended by other developers. Furthermore, ZAP is coded with modularity in mind. The sequence mechanism should also strive towards being as modular as possible.

The sequence mechanism should be self-contained, meaning that it should not be dependent on other extensions in ZAP, apart from those that are part of the core. However, the Zest extension is not part of the core, so this needs to be handled without altering the architecture of ZAP, simply to reach the requirements of this project.

Performance

The mechanism is loaded into the application as an extension, either at startup or through the extension manager. When a sequence is scanned, the actual handling of the requests is performed by the active scanner. Sending several requests, the active scanner will then most likely be the bottleneck of the functionality. Furthermore, the performance of the active scanner greatly depends on the network connection of both the client and target machine. As a result, the performance of the sequence extension is negligible, as it greatly relies on the performance of external factors. However, the sequence mechanism should not slow down the system unnecessarily, which should be kept in mind while developing the code.

Testability

Within the ZAP Repository, a project for testing functionality is available, including a dummy web application for testing purposes. To verify that the functionality of an extension works as intended, a scenario can be set up in the web application in which the extension can be used. A set of unit tests can then be written, to assert that the functionality of the extension produce the expected results. However, the existing unit tests are quite simple, and only a small fraction of the overall functionality is covered. This indicates that it might be difficult to set up unit tests for advanced functionality in ZAP.

Internationalization

ZAP has been developed with internalization in mind. For each language that is supported, a **Messages.properties** file is created. Each line contains an internationalization key, and the actual text. When a text is to be displayed in the GUI, the key is applied, and the text is looked up in the corresponding **Messages.properties** file. Not all texts have been given a internationalization-key yet, and not all texts have been translated to other languages (in which case, ZAP will fetch the default text from the English file). For development purposes, the GUI texts can be hardcoded, but it is essential that they will eventually be given an internationalization key for translation purposes.

Usability

For new users of ZAP, the graphical user interface can be slightly overwhelming since so much functionality and information is available right from the main screen. Therefore it is important to make a sequence mechanism have the same look and feel as the rest of the ZAP interface, so users will be more familiar with how they should interact with the system.

Beyond this, any functionality should be intuitive to use, so users will not necessarily have to consult the documentation in order to use any basic functionality. More advanced features should be easy to find, however not clutter the basic view.

Overview

Based on the requirements defined, Table 3.12 summarizes the importance of each specified requirement.

3.7 Test Strategy

Before the release of an extension or a different kind of functionality in ZAP, it is important to make sure everything works as expected. Code that is committed to the ZAP repository is naturally tested by the core team, however these tests can be

	Must have	Nice to have
Create, modify, persist sequences as Zest scripts	•	
Active scan sequences	•	
Mark alerts as scanned with sequence		•
Sequence functionality included in API		•
Sequence mechanism available as one .zap file	•	
Code well documented	•	
Code follow same style as the rest of ZAP	•	
Code not dependent on non-core extensions	•	
Avoid unnecessary use of resources		•
Verify functionality with tests	•	
GUI texts internationalized		•
Ease of use	•	

Table 3.12: An overview over which requirements are Must have, and which are Nice to have.

superficial, and should ideally only be for verifying and making sure that code structure is as it should be. Therefore it is important to make sure that the implemented code is sufficiently tested before actually committing it.

The ZAP workspace provides an additional project which is only intended for unit-testing. Ideally this extension should likewise have targeted unit-test, however since the developed functionality is comprehensive and is dependent on several external factors, it might not be possible to create satisfactory unit-tests. Alternatively effort should be put into creating test-cases that proves that scripted sequences support ZAP functionality. The following features should be functional and tested before an actual release:

- Session handling
- Anti-CSRF tokens
- Authentication

Furthermore it should be tested that it is possible to locate unique vulnerabilities using sequence scripts. The problem was initially that it would not be possible to find persistent XSS in cases where data would require several steps in order for it to be persisted. This behaviour was observed on the WackoPicko test site, which would therefore be ideal for testing a sequence solution.

CHAPTER 4

Design

This chapter serves to explain some of the choices for the design of the sequence mechanism for ZAP. The chapter begins with an explanation on how to extend functionality in ZAP, followed by a section on hooking into specific functionality. The following sections will explain how to handle sequences, and how to define a useful user interface for custom functionality in ZAP, thereby creating a good user experience.

4.1 Extending ZAP

Adding new functionality to ZAP can be done by creating a class which extends **ExtensionAdaptor**, which is a class that originates from the Paros Proxy part of the codebase. By overriding the **hook** method and using the **ExtensionHook** parameter which the method provides, an extension class can gain access to different parts of the ZAP functionality. This means that an extension can use data and functionality that is stored elsewhere in ZAP, and create GUI elements in places that would normally require an update to the core of ZAP.

Newly created extensions will usually be placed in the Alpha development branch. It is possible for new extensions to reference other extensions, in order to use any functionality they provide. However, this creates dependencies for the newly created extension. Guidelines from the ZAP development team suggest that it is not a problem to reference extensions that are part of the core of ZAP. However, if a reference is created to an extension that is not part of the core, problems might occur if both extensions are not in the same package or development branch.

The sequence mechanism should ideally be created as a self-contained extension, using only functionality available from the **ExtensionHook** class, and from core-extensions. However since this sequence extension will interact with the Zest extension (which is currently in the Beta branch), the extension may need to be placed in the Beta branch.

Along with the **ExtensionHook** class, ZAP also provides a few interfaces for classes to implement, to react to certain events in ZAP. One of these is the **ScannerListener** interface. This interface provides methods that will be invoked by the scanner when different events occur. Classes that want to react to these events, can register themselves with the **Scanner** class, and when an event happens, all the

classes that implement **ScannerListener** will have the corresponding method run. An example is the **scannerComplete** method. When an active scan is completely finished, the scanner iterates through all the classes that have registered as a **ScannerListener**, and runs the **scannerComplete** method. This is useful, if classes want to update GUI elements, or otherwise use the complete results from the active scan.

The sequence extension will handle all interactions regarding sequences, while performing an active scan. The actual sending of HTTP requests during an active scan, is handled in the plugin classes, that define an active scan rule. Not all scan rules send requests, as some scan rules simply aim to find specific patterns or tokens in the response of a message. Accordingly, the handling of sequences should only be relevant, for plugins that send requests to a server. The **AbstractPlugin** class, that all plugins extend from, has a method, **sendAndReceive** which sends a request and populates the response object of a **HttpMessage** with the result. Plugins will usually modify the original request in some way, before passing it to the **sendAndReceive** method, and inspecting the response afterwards. In order to handle sequenes, the message needs to be intercepted before it is sent.

4.2 Hooking into ZAP

To be able to run a sequence before a plugin sends a message, the extension need to be able to hook into the scanner. To do this, a new hooking mechanism can be defined, which can hook into the specific parts of the system that the sequence extension requires. The hooking mechanism should also be usable for other extensions, that want to intercept messages before they are scanned. At the same time, the hooking system should also be easily expandable, if other extensions need to implement functionality that would be similar, but not yet covered by the hooking mechanism.

As the sequence extension will primarily need to hook into the scanner, the **ScannerListener** interface could be expanded to support this functionality. However, **ScannerListener** is intended to be used by core classes. It is also not possible for extensions to register themselves as **ScannerListener**s, as there is no reference to the Scanner through the **ExtensionHook** class. Also, the methods of the **ScannerListener** interface are intended to only inform the registered listeners about certain events. The objects which the interface methods pass as arguments can not be used to modify any data that will be used later.

Instead, the hooking mechanism will be created as a new interface, called **ScannerHook**. To execute functionality before a plugin sends its request, the interface will have a method called **beforeScan**. To figure out if the current **HttpMessage** is part of a sequence, the method have the current message passed as a parameter.

In the same way as the **ScannerListener** interface, classes that implement the **ScannerHook** interface can register with the scanner. The scanner will then run the **beforeScan** method for all classes that have registered with the scanner. An illustration of this can be seen in Figure 4.1, where two classes have registered as implementors of **ScannerHook**, and the scanner issues a scan on a single **HttpMessage** to a plugin. The scanner plugin will then call to the scanner to call the **beforeScan** method of each registered **ScannerHook**, before actually sending the message, and reporting any scan results. Note that an active scan will usually consist of many plugins that will run for the same HttpMessage, so the flow shown in the figure will likely happen multiple times for each **HttpMessage**.

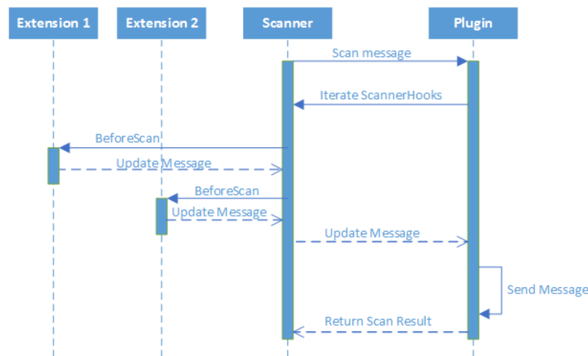


Figure 4.1: Demonstration of how plugins can invoke methods in extensions, before scanning a **HttpMessage**.

This setup will ensure that classes that want to perform actions before a message is sent in any plugin, will be able to do so. Any objects passed as parameters will be the same objects that will be used at a later time in the scanning process. The interface will also be expandable at a later time, if other classes need to preempt or respond to events in the scanner, that require modification of data.

Registering a class as a **ScannerHook** will be possible through the **ExtensionHook** class, thereby giving extensions the possibility to act on scanner events, and not restricted to core classes.

As mentioned, the sequence extension will be a part of either the Alpha or Beta development branch. However, since this hooking mechanism will require changes to the core of ZAP, the hooking mechanism cannot be deployed as a single self-contained extension. Instead, the changes need to be made directly in the core codebase, and from there, built and propagated to the Alpha and Beta branches, so that extensions can use the newly created core interface.

4.3 Handling Sequences

With a structure for getting an extension to hook into the active scanner, it is now possible to look into how the actual sequence extension should be created. This process must both investigate how to define sequences and subsequently execute a given script as part of an active scan.

Defining Sequences

The first part of creating the extension is to explore what possibilities there are for handling and defining sequences in ZAP. To do this it is necessary to know what a sequence actually is. In theory HTTP-messages are not dependant on previous steps, this means that ideally requests would not be dependent on each other, unless of course some form of data are persisted on the server for later use. In its current state ZAP considers each HTTP message as a individual element, meaning that ZAP does not have any knowledge of how some messages could be dependant on each other. In many cases this assumption is true, but not always. In some cases not calling HTTP messages in the correct order would yield incorrect responses from the server, such as a redirect or a error return message.

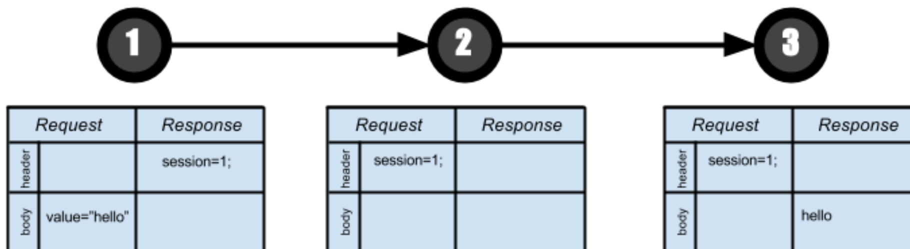


Figure 4.2: An example sequence of HTTP messages

The simplest way to explain a sequence is to define it as a series of HTTP messages including requests and responses, however there are various ways a sequence could be structured. In figure 4.2 there are three steps in a sequence. In the first message the user posts a value through a web site and is replied with a message that sets a cookie on the clients browser. This post could for instance be a login form or similar, but could also be any other kind of data input from the users point of view. The second step functions as an intermediate step, where the user is required to perform an action in order to be allowed to continue. The final step then returns the initial value. In this case the server likely maps the current user to the session thus storing any user-specific values based on this. There are theoretically numerous other ways a server could handle this besides cookie values, but nonetheless if the sequence is not

performed, the third step might not return the value from the first step.

There are various ways a sequence may fail if the requests do not provide the correct content at the right time. This could include anti-CSRF tokens or other POST data. HTTP messages in sequences needs to be aligned to the previous step, in order to update the correct values such as session and other data received by the server.

To sum up, a HTTP message must contain a request and a response, even though the latter might not be populated. Each of these respectively has both a header and a body. Theoretically there could be information in any of these, such as cookies, POST data or even hidden form fields, required for a sequence to run successfully. In order for ZAP to be aware of a sequence it naturally must be defined by the user and afterwards be stored somewhere in the application for it to be used when an active scan is executed at a later stage. Since ZAP currently does contain the concept of HTTP messages, a viable solution could be to store a sequence as a list of message objects. This would however mean that much of the functionality must be implemented manually in the sequence extension, such as an interface for defining sequences and saving it for later use.

A different approach that would expand existing functionality, instead of creating new, would be to make use of scripts in ZAP. Scripts already need a concept of HTTP messages in order to be useful in ZAP. Furthermore scripts can be saved to files which would make it possible for users to store a script for later use. Additionally since scripts essentially are text files this makes it possible for users to share them. This would also mean that if a penetration tester finds a vulnerability that require a certain sequence, they can share it with another person. This could for instance be the developer responsible for the tested application. Since a script is not dependent on ZAP it would also not require the developer to have any knowledge of ZAP. This flexibility is one of the primary selling points of the Zest scripting language which arguably is the most integrated scripting language in ZAP. Furthermore its structure aligns itself very well with the concept of sequences. Zest is therefore a good choice for an initial implementation, since it is very simple, built using JSON, and contains a native way of handling HTTP messages. Zest scripts can be created and edited through the ZAP interface, which makes them easy to handle in the ZAP environment.

Figure 4.3 shows how ZAP handles scripts. A script can be one of many script types, such as an active scan rule, proxy scripts, etc. Each script type has a subset of possible engines available, but this differs for each script type as it is dependant on what is implemented in ZAP. Usually ZAP at least has support for both Zest and Javascript, but it is not a certainty. Even though existing Zest script types can store sequences, their execution is already defined in ZAP, and having scripts which have multiple functions would likely confuse many users. That is why the best solution would be to create a new script type for sequences, to keep the functionality separated

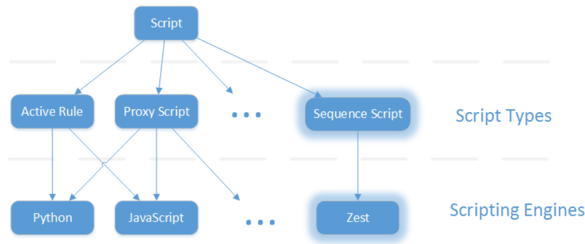


Figure 4.3: Script structure in ZAP

from other script types in ZAP. The structure of scripts in ZAP also means that even though the current extension will only implement a sequence script engine for Zest it would be simple to implement other scripting languages in future versions if it would be required.

Running Sequences

When a user has defined a sequence script through the ZAP interface, and subsequently is running an active scan with sequence scripts enabled, the extension needs to use the previously defined hook mechanism. This means that every time a plugin attempts to resend a HTTP message the sequence script extension will be invoked. The extension will then have to do a series of actions in order to make sure the next HTTP message that a plugin wants to send, is done so correctly, as part of a sequence.

A sequence extension will be started by a plugin using the newly implemented hook mechanism, as seen in figure 4.4. The extension gets a reference to the next HTTP message that is about to be sent. The extension will then have to investigate if the received HTTP message is actually part of a sequence. Since there are no pre-existing way of doing this the extension must search all nodes in all sequence scripts for a matching HTTP message. HTTP messages will have to be matched on a set of unique values, since there are no direct reference from the HTTP message a plugin uses to the one stored in a script. If no message is found the message is not part of a sequence and the extension will stop since there is no script to be run. On the other hand if a message is part of a script, the extension will save the script in which the match was found.

Once a matching script has been found, the extension needs to run all previous statements in the script, before the current HTTP message. To do this a sub-script containing all nodes prior to the current message is constructed. A feature of Zest is that the Zest execution engine can handle mapping of stored data in the script, such as session IDs, cookie data, etc. This means that the sub-script execution is partly handled by the Zest engine, instead of by the sequence extension. Finally when the

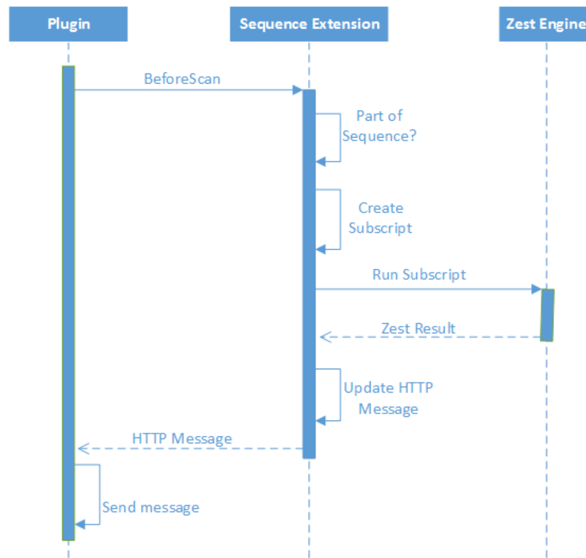


Figure 4.4: Running sequence scripts overview

sub-script has been run, the extension will have to map data between the final script result, and the HTTP message received from the plugin, which has not yet been sent. This data includes any cookie values, anti-CSRF values or other changing content.

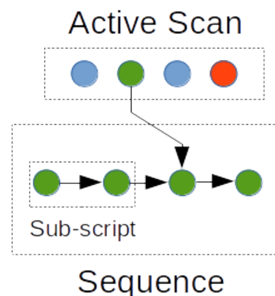


Figure 4.5: Connection between HTTP messages in the active scanner and sub-scripts

Sub-scripts are created from matching the current attacked HTTP message. The sub-script will then contain all Zest elements before the current message, as shown in figure 4.5.

Various alternative variations of how a sequence should be run have been con-

sidered. One way was to make the Zest script override the plugin message sending (*sendAndRecieve*) completely. Even though this would map sequence data automatically it would require a mapping between any data changes from the plugin HTTP message, such as injection data or similar. Furthermore this would make the scanner hook mechanism work in a different way that likely would make it less useful for future extensions to use.

4.4 User Experience

This section will focus on how ZAP users will interact with the sequence mechanism, through the graphical user interface (GUI) of ZAP. A lot of the core elements in the ZAP GUI has been made pluggable. This means that extensions or other non core classes can insert GUI elements. An example of this is the bottom tabs of the main screen, seen at **(2)** in Figure 2.5. Here, a lot of tabs have been added through extensions, such as a tab for Fuzzing, a tab for showing Zest results, a tab for showing an overview of Http Sessions, and many more.

There are many places in the ZAP GUI where it is possible to add elements dynamically. This is useful for ensuring that ZAP has the same look and feel, while still letting developers define how to control the functionality of their extensions, using predefined building blocks. However, not all parts of the ZAP GUI are pluggable, which means that developers will have to hardcode their changes, even though it might not be for core functionality.

Sequences in ZAP will be defined as Zest scripts. Therefore, it would be beneficial to use some of the GUI elements that the scripts and Zest extensions already have in place. For example, ZAP supports adding HTTP Messages from the History Tab or the Sites Tree, to a Zest script, using a right-click context menu. This is demonstrated in Figure 4.6 where the context menu allows for adding a HTTP Message to a predefined script or create a new one. This can be useful for creating sequences from existing HTTP requests. However, this is not supported for all script types or even all Zest scritps, so the context menu needs to be expanded to include sequence scripts defined as a Zest script.

Sequences will be scanned while performing an active scan, so it should be possible to define how and which sequences to include, in the GUI. Some script types, such as Active Rules or Script Input Vector, can be individually *enabled* or *disabled*. This is indicated in the GUI with a small green check mark or a red cross, next to the script in the script tree. Figure 4.7 shows an example, where four Active Rules have been loaded, but only two are enabled. If an active scan is executed, only the two enabled scripts would be included. Sequence scripts should also use this enabling mechanism, to determine which sequences will be included in an active scan.

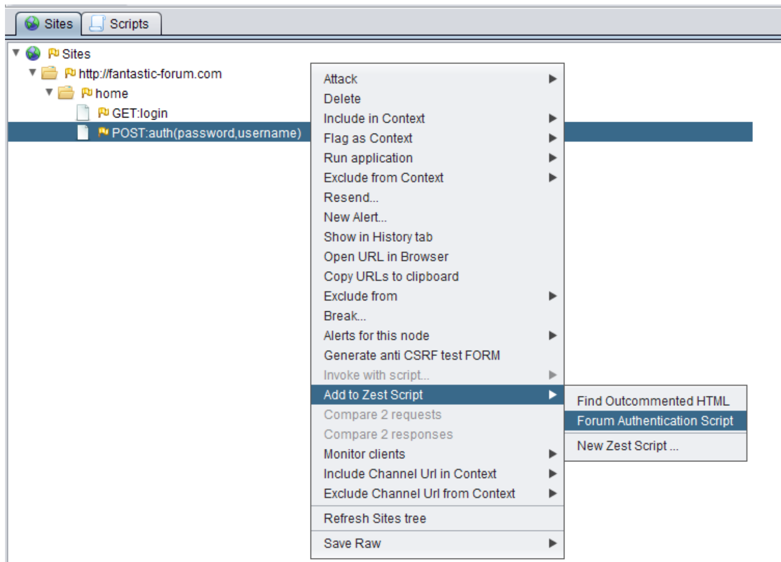


Figure 4.6: A screenshot of the sites tree context menu, where it is possible to add requests to new or predefined Zest scripts.

Scripts that are enabled can still be excluded from an active scan, by modifying the Scan Policy. This might be confusing, since a user would probably expect a script to be included if it is enabled. However, scripts are included by default, so this will only be a problem if an advanced user chooses to change the Scan Policy.

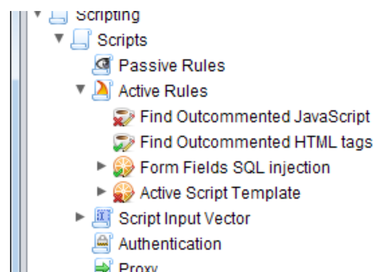


Figure 4.7: A screenshot, where four Active Rule scripts have been loaded, but only two of them are enabled.

Even if a sequence script is enabled, it might not be included in an active scan, if the scanner doesn't target any URLs that are present in the sequence script. It should be possible for the user to simply execute an active scan on the sequence script itself, so that all URLs in the script will be scanned. This can be done by inserting

an option in the context-menu, that appears when right-clicking a sequence script.

Scanning sequences might seem as an advanced feature for new users of ZAP. In this case, it might make more sense to scan sequences, through the *Advanced Active Scan* dialog. This dialog (which can be seen in Figure 3.4), contains all settings related to an active scan. From here it is possible to tweak all parameters of a scan, and customize which properties to target. However, the dialog is not pluggable. A tab can be created, for setting up sequences for an active scan, but the tab will be present in the dialog, regardsless of the sequence extension being loaded or not. This creates a scenario with high coupling, which is not ideal. However, modifying the dialog to be pluggable will be a large change to the core of ZAP, which is out of the scope of this project.

User Perspective

In the analysis chapter, a selection of user stories was presented. A few of these, focus on the need for ease of use, in an application like ZAP. Although the ZAP GUI can sometimes feel a bit complicated to navigate, the proposed implementations of GUI support for the sequence extension will attempt to make the process of creating and scanning sequences as easy as possible. An intermediate user of ZAP, who is used to working with other script types, will have no trouble using sequences.

Advanced users, such as professional penetration testers, will most likely use the *Advanced Active Scan* dialog to perform customized scanning. In this case, a tab for setting up sequences to scan would be ideal, but would either require hardcoding the tab into the dialog, or a major overhaul of the entire code for the dialog, in order to make it pluggable from extensions or other non core classes.

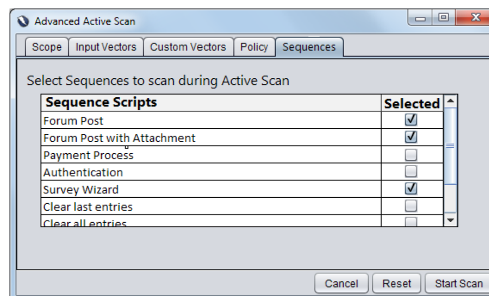


Figure 4.8: A mock up of a tab for choosing which sequence scripts to include in an active scan.

The dialog is quite cluttered with settings as it is. If a tab for sequences is to be

implemented, it will have to be very easy to understand and use, in order to be useful. A mockup of a suggested design of the Sequence tab can be seen in Figure 4.8. The tab on the mockup consists of a table of all loaded sequences (enabled or not), and it is possible select which scripts to include in a scan.

To sum up, when introducing new functionality in ZAP, additions to the GUI should be kept as simple as possible. To ensure the same look and feel as the rest of ZAP, some of the pluggable GUI elements can be used. If a change in the GUI needs to be hardcoded, it should primarily be for core functionality, and not for extensions that can be loaded/unloaded dynamically.

The sequence extension will add functionality to scripting and Zest, but will use the existing GUI elements for creating scripts and manipulating data in Zest scripts.

CHAPTER 5

Implementation

This chapter explains how some of the key elements of the solution has been implemented. Based on the design choices made in the previous chapter, the implementation was done in four main iterations. Each iteration had a main focus that the following iteration would build and expand on, increasing the complexity and functionality of the overall solution.

5.1 Iteration 1: ScannerHook

To be able to intercept HTTP messages before they are sent during an active scan, the **ScannerHook** interface has been created. To be able to use it from Alpha and Beta extensions, the interface has been created in the Release branch of the codebase. As it is designed to plug into the scanner, it is located in the same package as the scanner, defined in the Paros package. The interface is shown in Listing 5.1. It simply consists of two methods, that implementors of the interface will define; **beforeScan** which runs before a message is sent, and **scannerComplete** which runs when the active scan process has terminated.

```
1 public interface ScannerHook {
2
3   void scannerComplete();
4
5   void beforeScan(HttpMessage msg);
6 }
```

Listing 5.1: ScannerHook.java

The **Scanner** object, which manages the active scan, does not exist before running an active scan, so the hooks can not be stored here. Instead, the **ExtensionHook** class of Paros has been expanded to include a list of **ScannerHook**s, and public methods for retrieving the list, and for adding and removing hooks from the list. Custom extensions can then add themselves as scanner hooks, through the reference to the **ExtensionHook** class that is provided in the **hook** method of an extension.

When an active scan has been started, the **Scanner** will then need to fetch the **ScannerHook**s from the **ExtensionHook** class. However, as there is no way to directly reference the **ExtensionHook** class, the scanner hooks are fetched through

the **ExtensionLoader** class, which is available through a static singleton class, called **Control**, from anywhere in ZAP. In the **ExtensionLoader** class, a method has been created for passing the scanner hooks to the **Scanner**. This method, **hookScannerHooks**, is shown in Listing 5.2

```
public void hookScannerHook(Scanner scan) {
    Iterator<ExtensionHook> iter = extensionHooks.values().iterator();
    while(iter.hasNext()){
      ExtensionHook hook = iter.next();
      List<ScannerHook> scannerHookList = hook.getScannerHookList();

      for(int j = 0; j < scannerHookList.size(); j++){
        try {
          ScannerHook scannerHook = scannerHookList.get(j);
          if(hook != null) {
            scan.addScannerHook(scannerHook);
          }
        } catch(Exception e) {
          logger.error(e.getMessage(), e);
        }
      }
    }
  }
}
```

Listing 5.2: The hookScannerHook method.

The **hookScannerHook** method takes a **Scanner** object as a parameter, and adds all references to scanner hooks to the **Scanner**, so it can be used in an active scan by every plugin.

The **Scanner** class represents an active scan, but the actual control of the scanning process is handled in an instance of the **HostProcess** class. This class controls which HTTP messages are going to be scanned, and passes them to each plugin that wants to perform an attack. Plugins that send a request in their attack, use the **sendAndReceive** method, which is declared in the superclass that all plugins extend from. This method has been altered, so that just before the message is sent, a call to the **HostProcess** is executed, in order to call the **beforeScan** method of the **ScannerHook**s.

An example of an extension that implements the **ScannerHook** interface can be seen in Figure 5.1. The top part of the figure shows the code for the extension, that registers itself as a **ScannerHook** through the **ExtensionHook** class, and overrides the two methods from the interface. In the **beforeScan** method, a message box is shown, that will pop up before a message is scanned. The bottom part shows the result; a message box is shown, during an active scan, but before the message was sent.

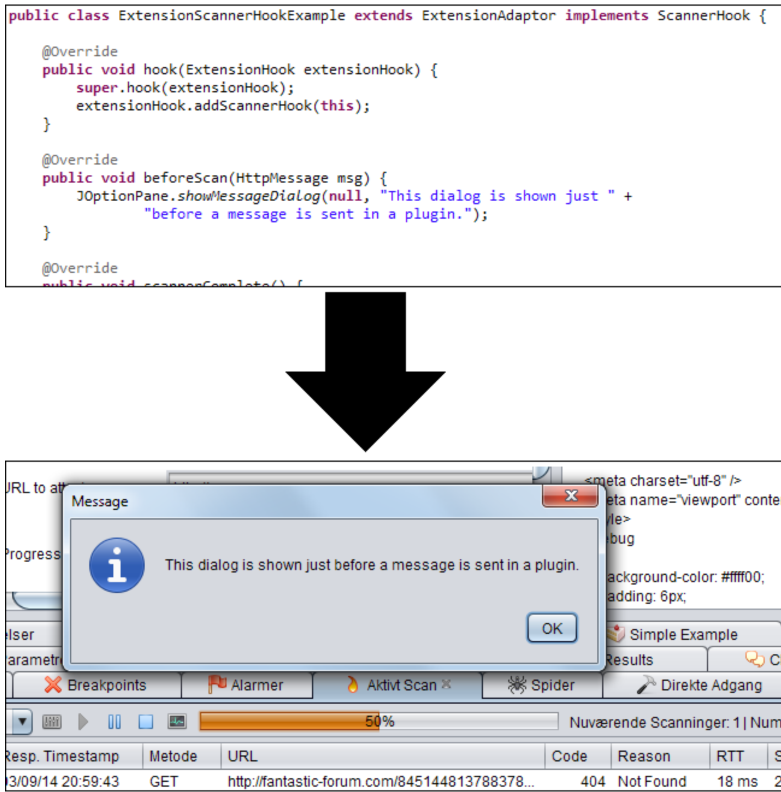


Figure 5.1: An example extension that implements **ScannerHook**, and the resulting Message Box that appears during an active scan.

5.2 Iteration 2: Proof Of Concept

With the basic implementation of a hook mechanism created, it was possible to start developing a proof-of-concept of the sequence extension. The goal was initially to hook into the scripting system, and afterwards use these scripts while executing an active scan. This iteration was also used for investigating what was necessary to include, in order to create useful HTTP requests in sequences. There were two primary focus areas in this iteration; the implementation for the independent sequence extension, and the implementation of a mechanism for handling sequence scripts in Zest. This following sections will explain the two separately.

The Sequence Extension

The first step was to create a new extension package, which should serve as the basic starting point for functionality relating to sequences. As previously mentioned, all extensions must extend from the **ExtensionAdaptor** parent class, and ensure that the class overwrites the **Hook** method. This method is run before the ZAP GUI has loaded, and is logically responsible handling all initial set-up between the extension and the ZAP runtime. The **Hook** method is a good place for hooking into the active scanner using the newly created **ScannerHook** interface. It is also a good location for setting up the new script type associated with sequences. Since the functionality of handing scripts also is stored in an extension, it is possible for the sequence extension to reference it through the static **Control** class. Through the script extension, it is possible register a new script type, however the actual implementation of a script type has to be done in the respective scripting language extensions. Furthermore, the extension adds a right-click context menu in the ZAP GUI. This menu is only displayed if the element, that was clicked, is a the top node of a sequence script. The addition to the context menu is essentially just a button making it possible to run an active scan on a specific sequence script, as seen in Figure 5.2. Beyond the direct active scanning, sequence scripts can be toggled on and off. If a sequence script is toggled on, it should automatically be included during an active scan.

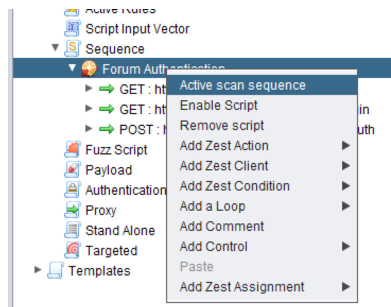


Figure 5.2: User interaction with sequence scripts

When run directly, through the right-click menu, the extension will start a new active scan with only the HTTP messages contained in the sequence-script. Apart from this, the scanner is run in the same way for both a direct scan or a ordinary scan. When an active scan has started, the hooked sequence extension will automatically be called whenever a plugin makes a **sendAndReceive** call, which is described in more detail in the previous chapter. Since the extension is called for all HTTP messages in a active scan, the first task is to investigate if the message that is about to be scanned, is part of any sequence scripts that are toggled on. The extension iterates through all available sequence scripts and examines whether or not a sequence contains the current message. To do this, access to content of each script is required, which the extension does not have directly since it should only function as a inter-

mediate controller between the hook mechanism and the actual implementation of various scripting languages. The way scripts are currently handled is by using the script extension to provide an interface for which the extension can communicate with any number of possible scripting languages. The script extension provides a **getInterface** method which returns a interface based on a provided script. The interface created for sequence scripts can be seen in Listing 5.3.

```
public interface SequenceScript {

  HttpMessage runSequence(HttpMessage msg);

  boolean isPartOfSequence(HttpMessage msg);

  void runSequence();

}
```

Listing 5.3: SequenceScript interface

The **Sequence** interface specifies how it is possible to interact with sequence scripts, since they must implement this in order for them to be run correctly. The **isPartOfSequence** method should return true or false depending if the message passed as argument, is part of the sequence script. If the method returns true, the extension should proceed and run that sequence by using the **runSequence** method with the current active HTTP-message as parameter. The other **runSequence** method without arguments is used when a sequence is run directly. Currently this functionality is primarily meant for debugging and the placement of direct scans will be changed in later iterations.

Zest Sequences

The sequence extension will invoke sequence scripts, however it is the script implementation that is responsible for actually making sure sequences run correctly. In this solution only Zest will support sequences, however if it would later make sense to expand to other scripting languages it could be possible. Other script languages are not as integrated in ZAP as Zest is, so this would likely require restructuring elsewhere in ZAP, but it is essentially possible.

Zest is structured in such a way that each supported script type has their own *runner* class which will handle exactly what is going to happen, when executed. The runners contain a reference to a **ZestScriptWrapper**, which is an object that represents the actual script, that the user has created through the interface in ZAP. All runners extend the **ZestZapRunner**, which in turn extends the external **ZestRunner** defined in the Zest library. The **ZestSequenceRunner** class also implements the **SequenceScript** interface (shown in listing 5.3) as this is the reference that the

sequence extension will receive, for accessing a script.

The first task assigned to the runner, is to inform the sequence extension to whether a given HTTP message is part of the current script or not. The script consists of Zest elements where HTTP messages are saved as **ZestRequest** objects. In order to compare the two, it must be defined what actually makes a HTTP request unique, to know what properties should be compared. For this implementation it was decided to compare the URL and the request-method. For a final release, this should be expanded to include comparison on parameter names as well, since the corresponding responses in some cases could be dependent on what parameters are included in the request.

```
1  @Override
2  public HttpMessage runSequence(HttpMessage msg) {
3    HttpMessage newMsg = msg.cloneAll();
4    try {
5      msg = getMatchingMessageFromScript(msg);
6      ZestScript scr = getSubScript(msg);
7      this.run(scr, EmptyParams);
8
9      List<HttpCookie> cookies = getCookies(scr, this.getLastRequest(), this.
           getLastResponse());
10     String reqBody = msg.getRequestBody().toString();
11     reqBody = this.replaceVariablesInString(reqBody, false);
12     newMsg.setRequestBody(reqBody);
13
14     newMsg.setCookies(cookies);
15     newMsg.getRequestHeader().setContentLength(newMsg.getRequestBody().length()
           );
16   }
17   catch(Exception e) {
18     logger.error("Error running Sequence script: " + e.getMessage());
19   }
20   return newMsg;
21 }
```

Listing 5.4: The runSequence method in the ZestSequenceRunner.

When a sequence is run through the **ZestSequenceRunner**, the first action is to create a subscript of the given sequence. A subscript contains all Zest elements prior to the current message and will be run using the Zest engine. When a subscript has been executed, the Zest engine saves the last request and response (as *ZestRequest* and *ZestResponse* objects). Finally the original HTTP message needs to be updated in order to correctly be part of the sent sequence. There are various parts of the last requests and response that need to be considered if this is to be achieved. First of all, it needs to be established whether or not a session has been initialized, which is most often the case, when dealing with sequences of communication. Most frequently, sessions are stored as cookies, which are set through a previous response header. To

handle this, each time a 'set-cookie' is received, it is saved by the runner. Furthermore, sequences support anti-CSRF tokens, which are encrypted values that can only be decrypted by the server. These values can be updated in every response, meaning that values in the latest message needs to be located and stored. ZAP already contains an extension for handling Anti-CSRF tokens, which has various functionality for extracting these values, which can be used in this example. The **runSequence** method can be seen in listing 5.4

5.3 Iteration 3: Optimization

The focus of this iteration, was to improve the sequence mechanism, based on the feedback for the previous iteration, from the ZAP developers. As scanning sequences is a somewhat complex feature, it was suggested that the selection of sequences to include in an active scan, be moved to the *Advanced Active Scan* dialog. While this required hardcoding a sequence tab into the dialog, it does follow the same scheme as in the rest of ZAP, where toggling of advanced settings for an active scan is located. It also makes it possible for a user to simultaneously include a sequence script in an active scan, while also selecting which scripts to scan directly. This avoids any confusion that might have arisen from having the possibility to both enable a script, and active scan it directly from the context menu, created in the previous iteration.

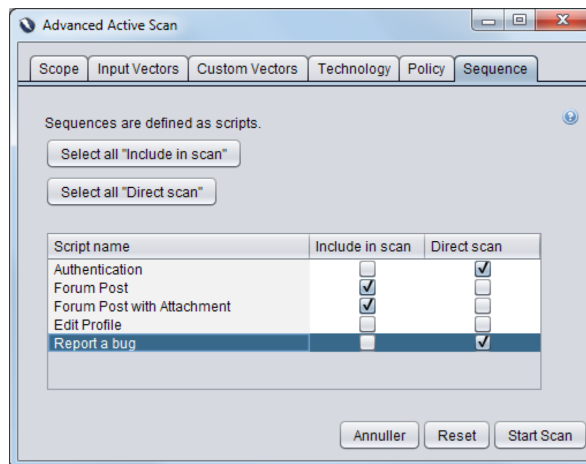


Figure 5.3: The updated Advanced Active Scan dialog, now includes a tab for sequences.

Figure 5.3 shows the updated dialog, with the sequence tab. Moving functionality to the dialog also made it possible to place a Help button on the sequence tab. The button is visually similar to other Help buttons in ZAP, and opens a page in the ZAP

help, specifically for sequences.

Improvements have also been made in the **ZestSequenceRunner** class. In the previous iteration, the HTTP message to be scanned needed to be updated, with any session cookies or Anti-CSRF tokens. This was done by extracting this information from each message sent, and mapping it to a sequence using the sesssion ID as a unique key. While this approach worked, it was definitely not ideal, as it required a session. It is not hard to imagine a scenario where a cookie is set, but where sessions are not used. The Zest engine has a mechanism for specifying which **HttpClient** to use for sending requests. ZAP and Zest use the same type of client, created by Apache. However, they use two different instances of the client and thus keep track of cookies independently of each other. By specifying the Zest engine to use the same client as ZAP, the manual extraction of cookies, session IDs and anti-CSRF tokens can be avoided, and let the client handle this. This is also useful, if the *Forced Session* feature of ZAP is enabled, to make all requests use the same session - including those sent through a script.

This change required a small addition to the **HttpSender** class of ZAP, to be able to fetch the **HttpClient** object. When a scanning of a sequence is about to begin, all cookies in the current **HttpState** of the client are cleared, in order to receive fresh values from the server. When a sequence is finished, there is no need extract any session or cookie data to update the message that will be scanned, as this is handled by the **HttpClient** and will automatically be included when the next message is sent.

The performance bottleneck of the sequence mechanism is definitely when the **ZestSequenceRunner** sends all messages of the script. There is very little room for improvement in this area, as it is mostly handled by other entities than the **ZestSequenceRunner**, such as the Zest engine and the **HttpClient**. However, other parts of the runner can be improved to reduce execution time, while scanning sequences. One idea was to cache the generated subscripts. During an active scan, many plugins will usually scan the same **HttpMessage**, and if it is part of a sequence, then a subscript will be created each time. To avoid the **ZestSequenceRunner** creating the same subscript several times for the same **HttpMessage**, subscripts are now stored in a HashTable, where the **HttpMessage** is used as a key. The **getSubScript** method then checks if the HashTable contains the HttpMessage as a key, and if it does, simply returns the already generated subscript. If the key is not found in the HashTable, the sub-script is created and then inserted into the HashTable to use it for the next call that will use the same sub-script.

Similar caching-like behavior could be implemented for the **HttpMessage**s that are to be scanned. Every time a plugin wants to send a message, it has to investigate if the message is part of any sequence script. This is done by iterating through all sequence scripts that are loaded, and running through each request in every script, to investigate if they are equal. Instead, the scanned message could be marked as part

of a sequence in the **HostProcess** class, the first time it is scanned by a plugin. This way, a lot of searching through lists can be avoided.

5.4 Iteration 4: Injection

The previous iteration showed significant improvements to the sequence mechanism, in terms of handling data that was generated during an active scan. By using a purposely vulnerable test-site, it was possible to detect vulnerabilities that would not have been discovered without the usage of sequences in an active scan. However weaknesses related to injections were not discovered, even though they were enabled in the test-site.

The solution so far was developed on the premise that some pages could not be reached if a sequence of prior steps was not conducted. The test showed that every page was accessed correctly but injection data was not committed. Especially testing the active scanner did not always show expected results, particularly when attempting to scan with the persistent cross-site-scripting (XSS) plugins. The way ZAP scans XSS is by using three different plugins that have different tasks in the process. The XSS "prime" plugin injects a unique value on every parameter during a scan, at the same time it saves the value as well as the parameter and request. Next, ZAP runs a XSS-spider plugin, which looks for injected values on all sites. Whenever one of the unique values are discovered, it saves the injection request, which parameter was injected and the request that the value was located on. Finally the actual XSS-scan plugin uses the values saved by the spider. First it verifies that injection was possible by injecting a value and viewing that the value was located on the result response. If this was possible it will attempt to inject various forms of scripts and then check to see if these were located on the corresponding response pages.

When looking at how such a scan is performed, it can be noted that data are not always correctly submitted to the server when injecting. This is because, even though all messages are reached, some data are not persisted until a later step in a sequence. Currently the sequence extension only makes sure that prior messages are sent, but if injections are to be persisted, it will also need to send subsequent messages. To support this the solution has been expanded with an **afterScan** method in the **ScannerHook** interface.

Listing 5.5 shows where the **afterScan** method is invoked. At the same time this means an update in the sequence extension as well as in the Zest implementation. Functionality is implemented in a similar way as the **beforeScan** method, but now the subscript consists of all messages after the currently scanned message instead. Simultaneously, **afterScan** does not have to update any return messages as this is done after as the final step. The reason to implement this as a separate method, is to make the hook have more purpose for other extensions wanting to use it. Further-

```
1     protected void sendAndReceive(HttpMessage msg, boolean isFollowRedirect,
          boolean handleAntiCSRF) throws HttpException, IOException {
2       ...
3         //ZAP: Runs the "beforeScan" methods of any ScannerHooks
4         parent.performScannerHookBeforeScan(msg, this);
5
6         parent.getHttpSender().sendAndReceive(msg, isFollowRedirect);
7         // ZAP: Notify parent
8         parent.notifyNewMessage(msg);
9
10        //ZAP: Runs the "beforeScan" methods of any ScannerHooks
11        parent.performScannerHookAfterScan(msg, this);
12      ...
13  }
```

Listing 5.5: Updated **sendAndReceive** method.

more, when looking at the sequence implementation, finalizing the sequence currently only seems relevant for injection plugins, but this should likely also be toggleable for active scan scripts. This way ZAP does not have to send unnecessary requests by running the remaining script for all sequence requests.

The caching of subscripts has been removed in this iteration. It was discovered that there was no way to clear cached subscripts when an active scan was complete. The cached subscripts were stored in the **ZestSequenceRunner**, but only the sequence extension could react to the event that occurs when the scanner has finished. Since there is no direct reference to the runner, the cached scripts could not be cleared. This would be a problem if two sequence scripts contained the same messages. If one script was enabled during one active scan and created a subscript for the message, and the other script was enabled in a following active scan, the previously cached subscribt would be retrieved, which is not part of the enabled script. Until a mechanism for clearing cached subscripts can be found, this feature has been removed.

CHAPTER 6

Test

While developing software for any kind of application, it is important to verify that the functionality works as intended. This chapter will describe how the sequence mechanism was tested during development. The chapter begins with a follow-up, on the Test Strategy, described in the Analysis chapter of this report. This is followed by a set of example scenarios that have been useful for testing the sequence mechanism.

The Test Strategy defined various requirements for testing the sequence mechanism, to ensure that the quality of the solution is satisfactory. As mentioned previously, it would have been preferable to set up unit-tests in order to verify that the mechanism works as intended. This would also be useful to ensure that the sequence mechanism still functions correctly, if changes to other areas of the ZAP codebase are implemented at a later time. However, the current unit-tests in the ZAP test project are very simplified, and are not optimal for testing advanced functionality of ZAP, such as that of the sequence extension. Altering the ZAP test project to fit the needs of this project would have been very comprehensive and non-trivial. It was concluded that it would be better to focus on black-box testing the mechanism, using real web applications.

These web applications would need to implement the functionality, that is described in the Test Strategy. Two features that were mentioned, were to handle sessions and anti-CSRF tokens. ZAP monitors incoming and outgoing HTTP messages, but not all messages are displayed in the GUI. Messages sent through a script, will not show up anywhere in the GUI. It was important to know exactly what the contents were of all messages sent and received during a scan of a sequence, to verify that the session IDs matched, and that the correct Anti-CSRF tokens were included in the requests. To do this, a tool called **TCPMon**[1], which was developed by Apache, was used. This tool serves as a proxy for monitoring HTTP-messages, much like ZAP, but shows all messages sent and received on a specific port number. ZAP listens on port 8080 by default, so TCPMon was set up to listen to port 80 and forward all messages to port 8080. An overview of this setup can be seen in Figure 6.1 Using this approach, it was possible to inspect all messages sent by ZAP, and verify that the messages had the expected content.

[1] http://ws.apache.org/tcpmon/

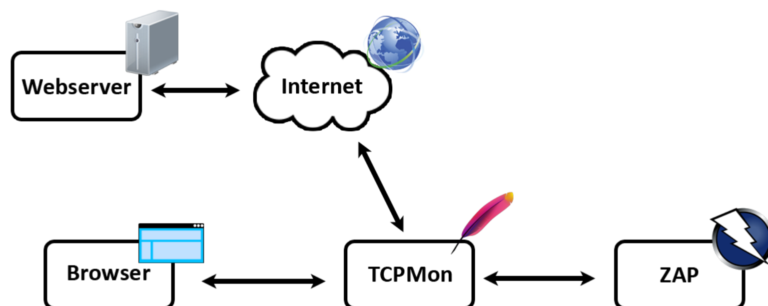


Figure 6.1: An overview of the setup, where TCPMon is used to intercept messages sent by ZAP or the browser.

Another important feature to support, is authentication. The authentication process could simply be included in a sequence script, thereby authenticating users on-the-fly, while scanning a sequence. However, this limits the authentication to the user specified in the script. Instead, the concept of **Context**s in ZAP can be used. ZAP Contexts provides functionality for defining several different users for a single web application. These contexts and its users can then be selected through the Advanced Active Scan dialog, before starting a scan that includes sequences. This is more useful than defining authentication directly in the script, as it opens the possibility of scanning a sequences with different users logged in. If users have different privileges on the web application, then a specific vulnerability may only be discoverable for certain users. Sequences can then be created, and scanned with different users logged in, to discover any vulnerabilities that are related to a specific privilege in the web application.

Example Web Applications

To test the functionality of the sequence mechanism, a web application was required, where scanning sequences would discover vulnerabilities that would not have been found without sequences. To achieve this, a web application was created in ASP.NET, so that it was possible to customize different sequences to ensure that the sequence mechanism worked with various features enabled. As the web application consists of different 'wizards', the application was named *Wizards R Us*, and the source code is available online[2].

The wizards on the web application all look the same to the user, but have different properties on the server side. Each wizard consists of 4 steps, where each step contains an HTML form. The first step has a textbox, and if a specific string (the

[2] `http://github.com/LarsKristensen/WizardsRUs`

word "dummy") is entered, a paragraph will show up on the third step, containing the words "Dummy Vulnerability". An Active Scan rule has been created to search for this string in the response of a HTTP request. This is naturally not a real vulnerability, but it was an easy way to test if a sequence would find a vulnerability, that would otherwise not have been discovered.

The web application currently consists of three different wizards. The first one simply stores user submitted data in the session, until it is persisted on the final step of the wizard. The second wizard is set up to place Anti-CSRF tokens between each step. This will ensure that it is not possible to jump directly to e.g. step2, and that the wizard must start at the first step. The third wizard requires the user to login before it is possible to start the wizard. A screenshot of the start page of the web application can be seen in Figure 6.2.

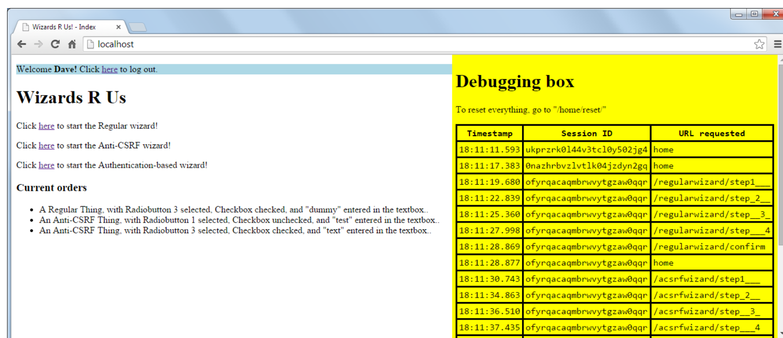


Figure 6.2: A screenshot of the 'Wizards R Us' Web application, after navigating through a few wizards and persisting some data.

Using the web application for testing provided a fast way to ensure that a vulnerability is only discovered, by scanning using sequences. However, one of the most relevant types of vulnerabilities for sequences, is cross-site scripting (XSS) vulnerabilities. It was not possible to implement XSS vulnerabilities in the ASP.NET web application, as ASP.NET has built-in mechanisms for preventing this, and it would require a lot of effort to circumvent this. Instead, the ZAP test project was expanded to include a persistent XSS vulnerability, which can be seen in Figure 6.3.

This was an important vulnerability to be able to discover, as it is the same type of vulnerability that the original issue for ZAP mentioned, in the purposely vulnerable web application, Wacko Picko. It was also possible to discover the Persistent XSS vulnerability in Wacko Picko, but it was a little more complicated. The issue is that Wacko Picko contains a commenting system for pictures that are uploaded. When writing a comment, it is possible to include HTML script tags in the comment. When

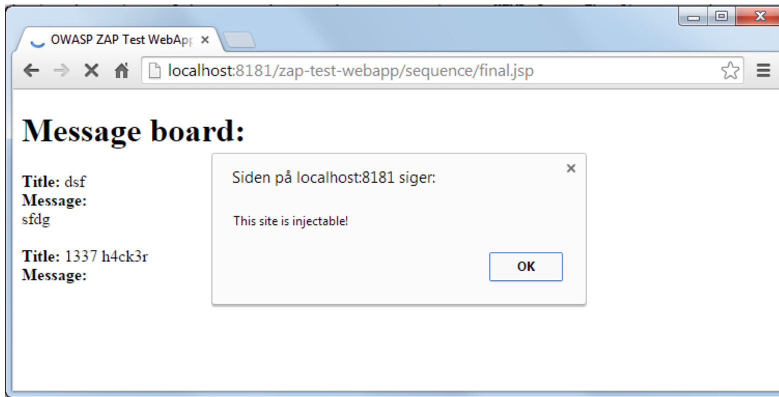


Figure 6.3: A screenshot of the ZAP test web application, with an injected javascript alert box.

the comment is submitted, a preview page of the comment is shown.

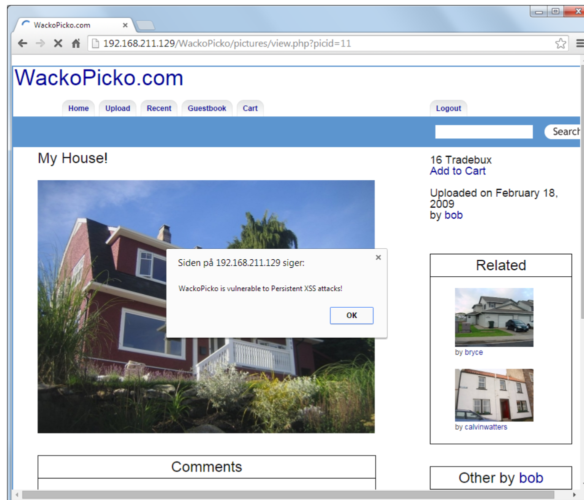


Figure 6.4: A screenshot of the Wacko Picko web application, with an injected javascript alert box.

On the preview page, only the entered text is shown, and no scripts will run. When the comment is finally submitted and persisted, the script will run for all users that view the image comments, as shown in Figure 6.4. The problem is that ZAP can only submit the comment the first time, but not continue from the preview page.

With a sequence script, it is possible to define the steps that are required to submit a comment, and the XSS injection Scan Rule will try to inject a script in the textbox, in order to find the vulernability. Also, the form on the preview page contains a hidden field with a previewID, which is different for each submitted comment. The sequence script needs to locate this value, and include it in the final request of the script. This functionality is already included in Zest, so the values just need to be mapped in the script. Once this is done, the vulnerability can be found, using the sequence script. Figure 6.5 shows a s screenshot og ZAP where the persistent XSS vulnerability is found, using a sequence script that has been included in an active scan.

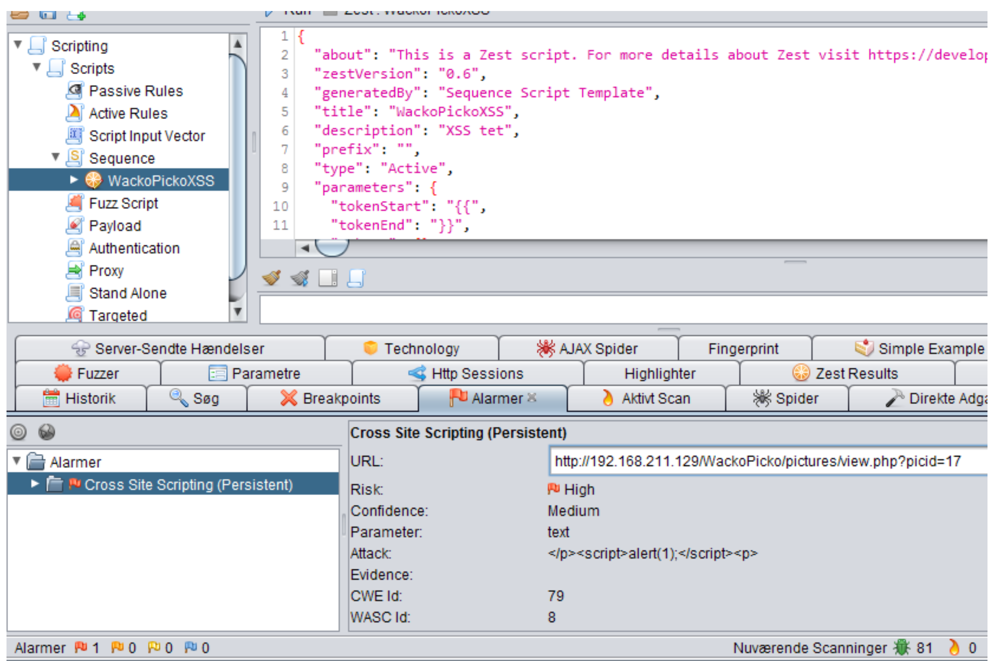


Figure 6.5: A screenshot of ZAP where the Persistent XSS vulnerability is found in WackoPicko, using a Sequence Script.

CHAPTER 7

Conclusion

This chapter will summarize the results from all previous chapters of this report, as well as evaluate how the thesis problem definition has been solved. A description on how the goals have been achieved will be presented, and conclude whether or not they has been achieved adequately. Furthermore, an overall conclusion on the project as a whole will be made, which will include a perspective on development structure and experiences working with OWASP ZAP. Finally the future project course will be elaborated, containing what improvements the developed solution are required to fulfill, in order to be finally released as part of a new ZAP version.

- **Is it possible to make ZAP implement sequences in such a way it would find vulnerabilities that would otherwise not be discovered?**

  It can be concluded that this definitely is possible and it has been proven, by the solution created for this project. Both project research and extensive test cases has shown that various vulnerabilities in some cases would not be found, if a sequence was not applied. Elaboration of this will be done in the answers for the subsequent questions.

1. **Why are sequences important while performing an active scan using stateless HTTP?**

   HTTP is stateless by design, but there are various ways this has been circumvented, e.g. by setting cookie values and various forms of data to indicate that ongoing communication has been established. This means that in some cases if a sequence is not followed, users might not reach a anticipated page. When executing an active scan this would in some cases mean that not all pages would be tested as they should, as the server would reject the requests.

2. **Which vulnerabilities are possible to expose using sequences?**

   Since some pages perhaps could not even be reached if a sequence is not followed, theoretically all page specific vulnerabilities may not be discovered. Static information, such as website certifications and other server related data, do not require sequence scripts in order for the scanner to find any issues. Usually, web sequences are used in cases where users must post data or evaluate previously submitted information. This means that the most common website weakness, related to sequences, likely are variations of injection attacks, such as database injections or cross-site scripting.

3. **How is it possible to extend ZAP to support sequences?**

   ZAP has been constructed in a way that tries to make it as easy as possible for developers to extend functionality. This is primary meant to be done by creating extensions which are self-contained packages within the application, which can be loaded and unloaded independently of the rest of ZAP. However, because of the separation from the rest of ZAP, extensions are limited in functionality to features provided through various forms of interfaces. Since the implementation of sequences need additional functionality, that is currently not present in ZAP, such as a relation to the active scanner, it has been necessary to build this feature independently of the sequence extension. Furthermore, the primary way of storing and executing sequences has been done using the Zest scripting language, which means that the way a sequence script is run, is done through the Zest extension which has been expanded to support sequences. The sequence extension mainly functions as a intermediate controller for running sequence scripts while the Zest sequence implementation contains most of the functionality. However if future versions of ZAP would contain sequence scripts implemented in other script languages this implementation would make it possible.

7.1 Project Conclusion

Based on the project goals, a solution has been developed that makes it possible to both create, store and execute sequences through ZAP. Users can create sequences through the script tab in ZAP, and manually set these up to match exactly what the user requires. It is possible for users to start an active scan, which is able to take sequences into account during execution. Every time a message is part of a sequence the scanner will now make sure that prior and subsequent messages will also be sent. This is done in order for the server to produce the right response, and simultaneously make sure that any sent data is stored correctly on the server.

Design structure and development has been performed in collaboration with the ZAP community and under the supervision of the ZAP project lead. Information to the community has been provided by posting on the developer forum while simultaneously posting additional information on an external weblog[1] created for this purpose. Likewise regular online meetings with the project lead has made sure that development has been on the right track, and the proposed solution would match the exceptions of the ZAP development team.

The extension was developed so be as self-contained as possible. This was partly achieved by creating the hook mechanism for the active scanner. However, the way ZAP makes it possible for extensions to hook into the interface and at the same time the way scripts are executed, means that changes had to made elsewhere in the ZAP

[1] http://zapmultistep.wordpress.com/

codebase. Nonetheless, everything has been created in a way that did not disturb any other functionality. It could be debated, however, whether or not the sequence extension should be merged with the Zest extension, since the current implementation may be seen as a controller between the hook and the sequence runner in the Zest implementation. This would however make future development of sequences inconvenient, and if other scripting languages should be implemented it would be dependent on the Zest extension.

Overall this project has given us a comprehensive understanding of how to develop and expand functionality in large open source project such as ZAP. At the same time we have obtained much experience working with legacy code, since a large continuously developed application such as ZAP inevitably will contain outdated or obsolete elements. Furthermore, working with a tool such as ZAP has unavoidably also given us a large amount of experience working with web security. Initially, we both had some knowledge of the topic, but in order to understand a problem such as this, it has been necessary to learn much more about the structure of the internet, HTTP requests, and various types of vulnerabilities. Finally, it can be concluded that the goals that were set in the beginning of the project process have been reached in a satisfactory manner, and we believe that with little future development, this solution will become a permanent addition to ZAP.

7.2 Future Work

The proposed solution should only be considered a proof-of-concept that shows that the vision and thesis definition has been possible to implement and is usable in practice. However, the solution would likely not require many changes in order to be acceptable as part of public ZAP version. The following sections show some of the future work required to finalize the implementation.

Referencing Site-Nodes

The current implementation uses **HistoryReference** objects in ZAP, to know whether a site-node is part of a direct scan or not. This is not an optimal way of using **HistoryReference**s, since these should be seen as a saved state and not a direct reference. To improve this, a unique id or reference should be created on a site-node (in the site-tree).

Pluggable Advanced Scan Dialog

Currently, the new sequence tab in the advanced active scan dialog, is implemented directly in the active scan extension. In order to improve this, it should be possible for extensions to plug into the advanced scan dialog. By implementing this into the extension, it would likewise make other extensions able to use this functionality.

API Support

ZAP can be set-up so it works as a REST-API which is callable through a network. In this solution it would not be possible to run sequences through the API. Much of this functionality are already in place, so it would likely not be a comprehensive task to support this in a future version.

Non Scanned Message in Active Scan

At the moment all messages sent and received during an active scan are displayed in the active scan tab. With the addition of sequences this list also includes all messages that are sent as part of a sequence, even though they are not scanned. A nice extension would be to make it visible which messages were part of a sequence. This is currently limited by the way ZAP handles tab interfaces, since they are not accessible from other parts of ZAP.

Sequence Discovered Alerts

When a vulnerability is found, an alert is raised, that is visible on the **Alerts** tab. If a user wants to know if a vulnerability is only present if scanning with a sequence, they would have to perform two active scans; one with sequences enabled and one without. The results from each scan would also have to be saved and then later compared, to observe any differences. An expansion could be made, so that it is possible to indicate that an alert was raised during the scan of a sequence. That way, a pentester could investigate if the vulnerability is related to the sequence, of if it is always present on the given page.

Client-Side Zest Support

In a upcoming update for the Zest library it will be possible to script client-side actions alongside HTTP requests. This means that interactions, which users usually would make through their browser, that do not include a request, can be scripted. This is a natural extension for sequences since it would make it possible to include client side interactions that could have an impact on website output in some situations.

Automatic Discovery of Sequences

The proposed solution makes it possible for the active scanner to detect vulnerabilities in cases where sequences of HTTP requests exist. The users must however know where likely sequences are before starting a scan and at the same time create scripts for any likely sequence. There is currently a project being developed that focuses on automatically detecting sequences. If this project is successful, it should be combined with the extension developed in this project, for fully automatic scanning of sequences.

APPENDIX A

User Manual

The project includes a code folder which contains all the source code as well as compiled version of ZAP that can be run independently.

Running ZAP

In order to run ZAP the following must be done:

1. In the included project folder go to the ”Executable” folder
2. From here open the ”ZAP_Dev Build” folder.
3. Depending on the operating system run either the zap.bat (Windows) or the zap.sh (Linux or Mac)
4. Zap will now start within a few seconds.

Testing

In order for having a testing environment it is necessary to have a web-server running to which it is possible to perform an active scanning. In the provided source folder a test folder can also be found. Inside this folder an example web-application can be found with the name ”zap-test-webapp.war” file. To run a ”.war” file the user must have an Apache Tomcat server running. ZAP uses port 8080 as default for proxying, which is also usually the default port Tomcat uses. To avoid issues, it is recommended that the Tomcat server be set to running on port 8181. If a different port is used, the provided script file must also be updated with the corresponding port number, in order for it to work as expected.

Loading a Test Script

1. When ZAP is running, load a script through the load script menu located in the top part of the scripts tab, as seen in figure A.1.
2. The script is located in the test folder and is called ”testsequence.zst”
3. When loading a script a pop-up will appear that will ask what type of script this is, make sure this is set to sequence as seen in Figure A.2. Press save on both dialogs, and the script will be loaded.

Figure A.1: How to load a script in ZAP

Script name: testsequence.zst
Script engine: Zest : Mozilla Zest
Type: Sequence
Description:
Load on start:
Cancel Save

Figure A.2: Loading a script in ZAP.

Setting up Firefox to Proxy through ZAP

In order to be able to run scripts, ZAP must have at least one site-node. To add a site in ZAP the easiest way is to proxy communication from a browser through ZAP. Figure A.3 and A.4 shows how to set this up in Firefox.

1. Set the Firefox to a use manual proxy connection.
2. Make sure that Firefox proxies through local-host. By default ZAP is running at local-host.
3. Set Firefox to use port 8080 as this is the one ZAP uses by default.
4. Clear the "No Proxy for" field, as it by default is set to not proxy local-host traffic.
5. Click OK.

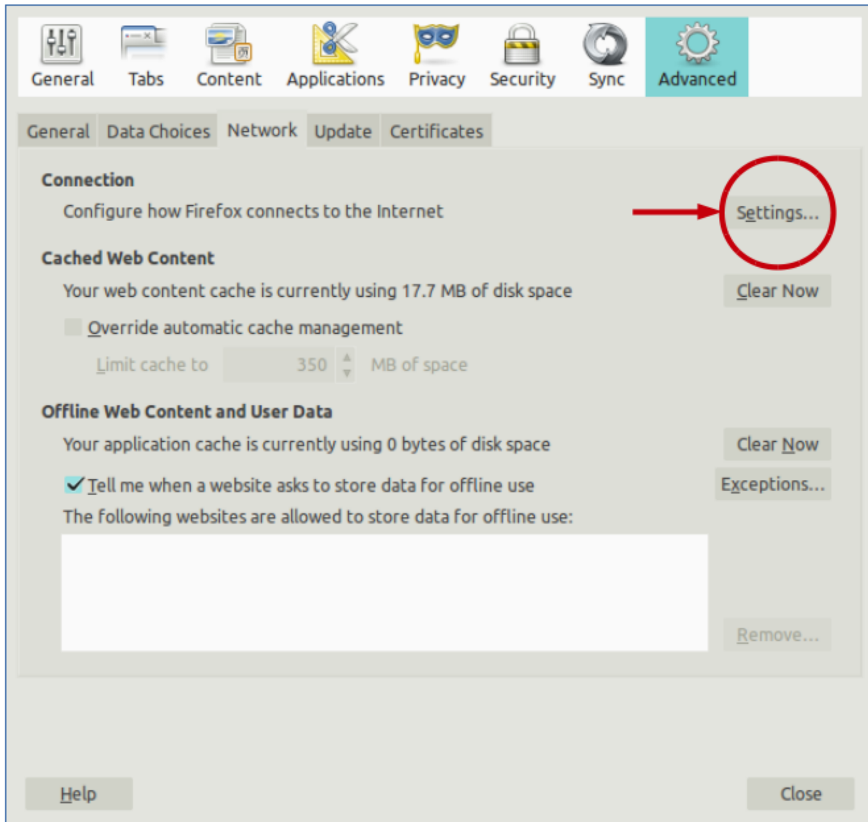


Figure A.3: Open the advanced connection menu.

6. Make sure that ZAP is running. In the Firefox URL bar write `http://localhost:8181/zap-test-webapp/sequence/` to add a site-node in ZAP. This can be seen in figure A.5

Running the Sequence

Everything should be ready for running the scan.

1. Open the advanced scan dialog by right-clicking the site-node, then hover over "Attack" and then click the "Advanced Active Scan.." menu.
2. The Advanced scan dialog is now open. As seen in figure A.6 the amount of plugins in can be controlled through the policy tab. It is advised to turn everything off except the three persistent XSS plugins. This is not required for

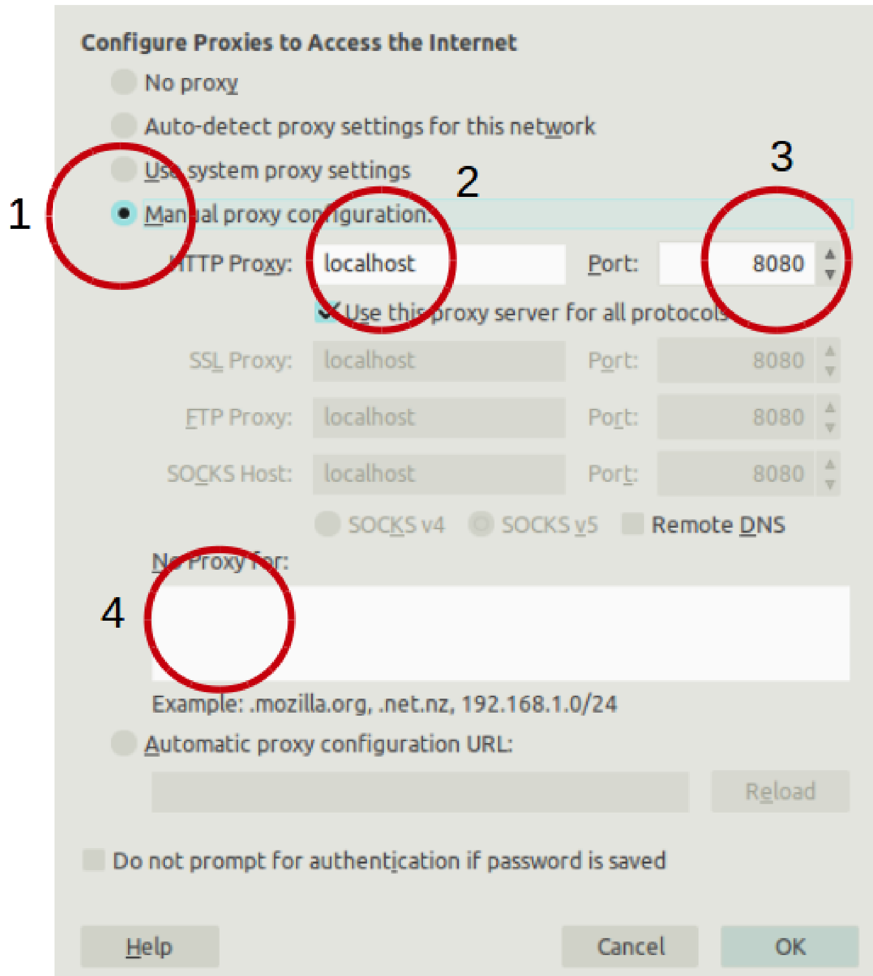


Figure A.4: How to make firefox proxy through ZAP

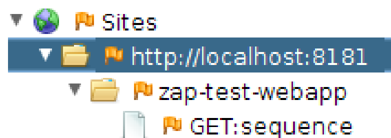


Figure A.5: A site-node in ZAP

the test to work, but it will limit the amount of time it takes to execute the test.

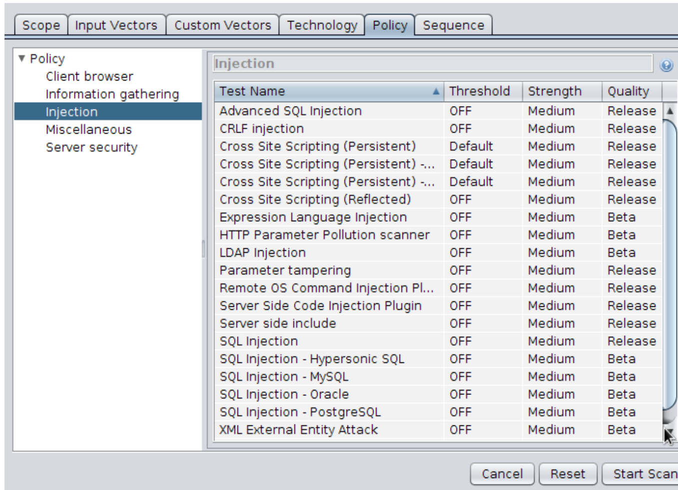


Figure A.6: The policy tab in the advanced scan dialog.

3. Go to the sequence tab and make sure the direct scan check box is checked for the previously loaded script. As seen in figure A.7.

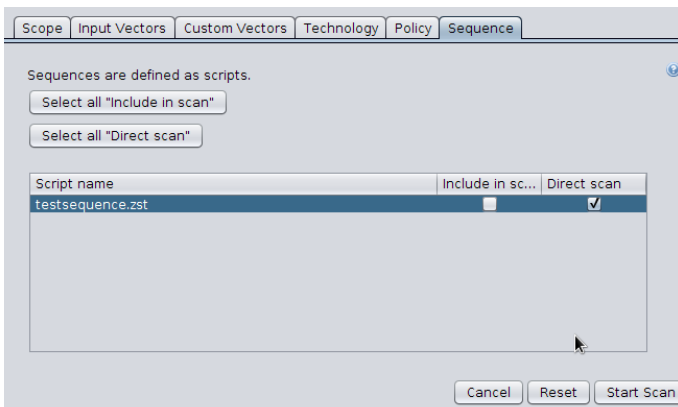


Figure A.7: The sequence tab in the advanced scan.

4. Finally click "Start Scan". To verify that the scan performs as expected go to the alert tab to see what vulnerabilities have been found. Figure A.8 shows that a persistent XSS has been detected.

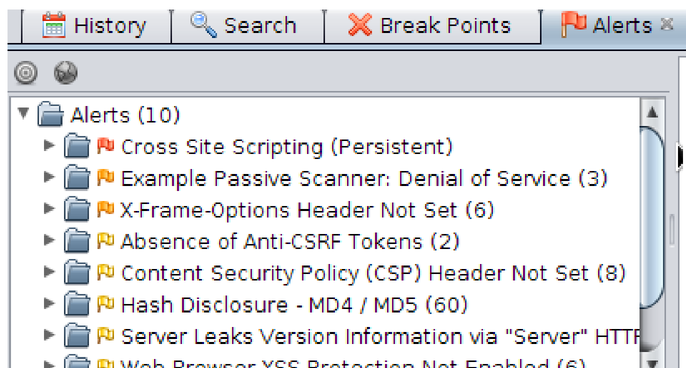


Figure A.8: The alert contains all found vulnerabilities, in this case a persistent XSS has been detected.

Bibliography

[Day13] Bhavya Daya. History, importance, and future. 2013.

[DCV10] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. 2010.

[Gar10] Jesse James Garret. *The Elements of User Experience*. New Riders, 2 edition, 2010.

[Gri13] Ilya Grigorik. *High Performance Browser Networking*. O'Reilly Media, 2013.

[LCC+09] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, and Daniel C. Lynch. A brief history of the internet. *ACM SIGCOMM Computer Communication*, Volume 39:22–31, 2009.

[PP11] Charles P Pfleeger and Shari Lawrence Pfleeger. *Analyzing Computer Security: A Threat / Vulnerability / Countermeasure Approach*. Prentice Hall, 2011.