



NTNU – Trondheim
Norwegian University of
Science and Technology

AppSensor

Attack-aware applications compared against
a web application firewall and an intrusion
detection system

Pål Thomassen

Master of Science in Computer Science

Submission date: June 2012

Supervisor: Lillian Røstad, IDI

Co-supervisor: Erlend Oftedal, BEKK

Norwegian University of Science and Technology
Department of Computer and Information Science

Sammendrag

Oppgaven tar for seg OWASP AppSensor-prosjektet. OWASP AppSensor er en ide om angrepsdeteksjon inne i applikasjonen. Den sammenligner OWASP AppSensor mot både en web applikasjons brannmur og et inntrengningsdeteksjonssystem. Sammenligningen er basert både på et kort litteraturestudie og et eksperiment utført. Eksperimentet tok for seg en rekke angrep basert på OWASP sin topp ti liste som ble utført mot en enkel nettbankapplikasjon. I eksperimentet ble inntrengerdeteksjonssystemet, nettapplikasjonsbrannmuren og AppSensor detektorene inne i applikasjonen testet for å se hvilke angrep de klarer å oppdage. Resultatet var veldig bra for både nettapplikasjonsbrannmuren og AppSensor siden begge klarte å detektere mange angrep, men AppSensor sin deteksjon var litt bedre.

Abstract

The thesis takes a look at the OWASP AppSensor project. The OWASP AppSensor project is about the idea of detecting attacks inside the application. The thesis compares OWASP AppSensor against both a web application firewall and an intrusion detection system. The comparison is based both on a short literature study and an experiment performed. The experiment was a set of attacks based on OWASP top ten list which were executed against a simple bank web application. In the experiment the intrusion detection systems, web application firewall and the AppSensor detection points inside the application was tested to see which attacks they were able to detect. The results were quite satisfying for both the web application firewall and AppSensor meaning that they detected many attacks but AppSensor's detection was slightly better.

Acknowledgements

A big thank to Hanne Ekran Thomassen for reviewing the thesis and for helping with both the language and structure of the thesis.

My external supervisor Erlend Oftedal from BEKK Consulting for supervisory and help. Both in the construction of the experiment as well has helpful feedback during the discussions and comparison. Also a big thank for pointing me to the idea and concept of OWASP AppSensor and providing me some interesting information about it.

My supervisor Lillian Røstad for help with the creating good research questions, help with the experiment and regular feedback during the thesis! She also helped me keep in track with both the timing and the scope of the thesis.

My wife Stine Mari Enlien Thomassen, for being supporting and understanding of the time required to finish the thesis. The work would have been a lot harder without her support!

Contents

1	Introduction	1
2	Background	4
2.1	Network IDPS	4
2.2	Detection Techniques in IDPS systems	6
2.2.1	Signature-Based Detection	7
2.2.2	Anomaly-Based Detection	7
2.2.3	Stateful Protocol Analysis	8
2.3	Summary Network IDPS	9
2.4	Web Application Firewalls	9
2.4.1	Usage of Terms	10
2.4.2	WAF Features	10
2.4.3	WAF in a Network	15
2.5	Defense in Depth	18
2.6	OWASP AppSensor	19
2.6.1	AppSensor Usage	21
2.6.2	OWASP AppSensor Detection Points	21
2.6.3	AppSensor Response Actions	24
2.7	Other Systems for Defeating Web Application Attacks	25
2.7.1	Language Defense	25
2.7.2	Platform Defense	26
2.7.3	Web Application Frameworks	26
3	Research Method	28
3.1	Research Strategy	28
3.2	Different Research Methods	29
3.2.1	Qualitative and Quantitative Strategy	29
3.2.2	Experiments	30
3.2.3	Chosen Research Method	32
3.3	Experiment Description	32

3.3.1	Environment for the Experiment	33
3.3.2	Setup of Snort	34
3.3.3	Setup of ModSecurity	34
3.3.4	Building in the Detection Points	35
3.3.5	Using the Detection Points	39
3.3.6	Execution of the Experiment	39
4	Scenario and Experiment	41
4.1	Business Case	41
4.1.1	Description of the Bank Web Site	42
4.1.2	Implementation Details	42
4.1.3	Attacks	44
4.2	Network Configuration	48
4.2.1	Firewall	48
4.2.2	IDPS	49
4.2.3	Web Application Firewall	49
4.2.4	Web and Application Server	50
4.3	Snort as IDPS	52
4.4	ModSecurity as WAF	53
4.5	AppSensor Implementation	54
4.6	Drawbacks with the Scenario	56
4.7	Scenario Summary	57
5	Attacks	58
5.1	Injection Attacks	58
5.1.1	SQL-Injection	58
5.2	Cross Site Scripting	59
5.3	Broken Authentication and Session Management	60
5.3.1	Authentication	61
5.3.2	Session Management	61
5.4	Insecure Direct Object References	62
5.5	Cross-site Request Forgery	64
5.6	Security Misconfiguration	65
5.7	Insecure Cryptographic Storage	65
5.8	Failure to Restrict URL Access	66
5.9	Insufficient Transport Layer Security	67
5.10	Unvalidated Redirects and Forwards	67
6	Results	69
6.1	Cross Site Scripting	69
6.2	SQL-Injection	70

6.3	Broken Authentication and Session Management	71
6.4	Insecure Direct Object Reference	72
6.5	Cross Site Request Forgery	73
6.6	Failure to Restrict URL Access	75
6.7	Unvalidated Redirects and Forwards	75
6.8	Testing Other IDS Systems	76
	6.8.1 Suricata With Emerging Threat Rules	77
	6.8.2 OSSEC Detection	78
6.9	Validity of the Results	79
7	Discussion	80
7.1	General Trends from the Experiment	80
	7.1.1 Unsatisfactory Snort Results	80
	7.1.2 Trying to Get Snort to Detect the Attacks	82
	7.1.3 Very Satisfying AppSensor Results	83
	7.1.4 Drawbacks with OWASP AppSensor	86
7.2	Response Mechanisms and Blocking	87
	7.2.1 Snort Blocking	87
	7.2.2 ModSecurity Blocking	88
	7.2.3 AppSensor Response Mechanisms	89
8	Comparison	90
8.1	IDPS Overview	90
8.2	WAF Overview	91
	8.2.1 Anomaly Detection	92
	8.2.2 A Note About Stateful Packet Inspection	93
8.3	AppSensor Overview	93
8.4	Recommended Use of the Security Mechanisms	95
	8.4.1 Defense in Depth	95
	8.4.2 Why Not Take All of Them	96
	8.4.3 IDPS or WAF	97
	8.4.4 AppSensor or WAF	97
8.5	Comparison Summary	99
9	Conclusion	100
9.1	Research Questions	100
	9.1.1 Research Question 1	101
	9.1.2 Research Question 2	102
	9.1.3 Research Question 3	103
	9.1.4 Research Question 4	103
9.2	Further Work	105

9.2.1	Larger Experiment	105
9.2.2	Study of AppSensor in Practice	105
9.2.3	More Comprehensive IDPS Experiment	105
9.2.4	Another Reference Implementation	106
9.2.5	Testing AppSensor on a Production System	106
9.2.6	Make AppSensor and a WAF Cooperate	106
9.2.7	Test Anomaly Detection Against AppSensor	107
Appendix A Zip-file Contents		i
Appendix B Screenshots of the Simplebank Application		iii

List of Tables

2.1	WAF Attacks	11
2.2	Table from [32] with the detection points	22
2.3	AppSensor Response Actions	25
3.1	Virtual Server Environment Specifications	33
3.2	Which detection points where implemented	36
4.1	The OWASP Top Ten	44
6.1	Suricata Results	77
6.2	OSSEC results	78
7.1	Which Attacks where Detected	81
7.2	Response Actions	85
7.3	ModSecurity Actions	88
8.1	Comparison of Strengths and Weaknesses	99

List of Figures

2.1	WAF and IDPS both setup in a inline fashion	16
2.2	WAF and IDPS both setup in a monitor/out-of-band mode . .	17
2.3	Simple Architecture diagram from[32]	20
3.1	Scientific Method, from [36]	31
4.1	Handling of HTTP requests and responses	43
4.2	Network Diagram for the Scenario	48
4.3	Data Flow Diagram for the Bank	51
4.4	Java EE Requests and responses with Filters	55
5.1	Showing Cross Site Scripting on the simplebank	60
5.2	Creating a new session cookie	62
5.3	Tamperdata tampering with the transfer money request	63
6.1	ModSecurity Rule with Javascript code it embed	74
8.1	Defense in Depth Illustrated	95
8.2	Showing the main differences between IDPS, WAF and AppSensor	96
B.1	Frontpage for the Simplebank	iv
B.2	Loginpage for the Simplebank	iv
B.3	Register new user page	v
B.4	Account overview page	v
B.5	Create new account page	vi
B.6	Transfer money page	vi

Chapter 1

Introduction

In the recent years, exploits against web application have risen. Today these types of attacks are still on the rise[39], that have given birth to many new web application security products. Although traditionally companies have used intrusion detection and prevention systems which monitors the network in general, there is now a widespread use of web application firewalls that monitors and protects only web applications. While these are general solutions to the security problems, there are still many attacks against specific business logic and functionality of a web application. The thesis will take a look at OWASP AppSensor project, which is a framework that can be integrated into a custom web application so the application can defend and react to attacks that is based on its own unique business logic and not only to general attacks. An example of this is Cross-site scripting or SQL-injections.

The Open Web Application Security Project (OWASP) AppSensor project¹ is a OWASP project regarding their AppSensors technology. OWASP AppSensor is an application level intrusion detection, which from here on is called AppSensor. This is in contrast with network Intrusion Detection and Prevention Systems (IDPS) which operate on the network level. The advantage with AppSensor is that it is closer to the application and the data that it is trying to protect. AppSensor can be used in custom software to get custom knowledge of the application and how it operates while more generic IDPS systems only can prevent general attacks that is application agnostic.

The overall goal of the thesis is to take a close look at which types of attacks and threats AppSensor can protect against and compare with other solutions. The other solutions will be both traditional network-level IDPS, Web

¹https://www.owasp.org/index.php/OWASP_AppSensor_Project

Application Firewall (WAF) , host-based IDPS and others. An evaluation of where and how to use AppSensor effectively alongside with the other technologies will be given. The evaluation will take into account, both which types of attacks can be prevented where and if more than one technology should be used to get satisfactory protection. The thesis will use a scenario as a basis for the comparison. The scenario describes both a network setup as well as the type of web application and some attacks to it.

The scenario is quite simple and will be implemented and used as a basis for comparing the different technologies and make it easier to see real world benefit from the different detection techniques. By having a scenario it is easier to see both the benefits and the drawbacks with the different techniques for detecting threats. Another benefit with implementing the scenario is that experience with implementing AppSensor into a web application will be gained. It will also help when comparing it against other drop-in solutions. Since AppSensor need to be built into an application it can make the development time larger and in almost all real life project there are time constraints. If a general solution like a WAF can detect most of the attacks and require no development time only some time to setup and deploy it can be much more effective from a business standpoint.

A drawback by using a scenario and perform tests is that it is hard to make a comparison between many different types of IDPS systems and web application firewalls. It would require more time and resources than this project have access to at the moment. Hence the thesis will compare a few such systems against each other and it will be noted that other systems might perform differently. This means other systems may detect other attacks or fail to detect attacks that the systems in the thesis detects. However it is hoped that the systems used in the experiment will be fairly comparative to other similar products.

To break down the overall goal it were transformed into four research questions. These questions is what the thesis is answering and the sum of them gives the overall goal.

- RQ1: What is the current state of application-based intrusion detection and prevention systems?
- RQ2: How does OWASP AppSensor compare to other IDPS technologies?
- RQ3: In the given scenario, what are the benefits of using AppSensors compared with trying to stop the attacks in a IDPS or WAF?

- RQ4: How hard is it to built AppSensor into an application?

Thesis Outline

The thesis start with a background chapter, which goes through some background in intrusion detection in a web application context. It discusses intrusion detection and prevention systems, web application firewalls and OWASP AppSensor in detail. Chapter 3 describes the research method chosen for the research questions in the thesis. It also contains some details about why and how the experiment was done. Chapter 4 explains the scenario developed and how it relates to the experiment. The next chapter details the attacks done in the experiment which were extracted from OWASP top ten.

The results are presented in Chapter 6. The results chapter explains if and how the attack was detected. The discussion Chapter 7 discusses the results and also highlights some other properties about the different security mechanisms that was not revealed by the experiment. In the comparison chapter these different security mechanisms are compared, this chapter was made so that the discussion chapter would not become too large but also to give a thorough answer to research question 2 and 3. The last chapter is the conclusion, which summarizes the biggest discoveries in the thesis and contains suggestions for further work.

Chapter 2

Background

This chapter starts with an overview over traditional network-based IDPS systems. The objective is to give the newer web-application based IDPS systems more context and also highlight the different techniques which have been used over the years. Traditional IDPS systems have become very sophisticated over the years and modern ones incorporate many different techniques for detecting and preventing threats and attacks. The chapter will also help understanding the evolution from network-based IDPS systems and how and why more specialized systems such as a Web Application Firewall were developed.

2.1 Network IDPS

The NIST Special Publication 800-94[50] gives an overview over traditional intrusion detection and prevention systems. It explains the purpose of, the common techniques used and types of IDPS systems.

Intrusion detection and prevention systems have several uses. Their main purpose is to identify and prevent incidents. This means that a pure Intrusion Detection System (IDS) should be able to identify possible attacks, for example a malicious payload from an incoming packet. If the IDS is also a Intrusion Prevention System (IPS) it should be able to respond to the attack, by for example blocking malicious incoming packets. In the thesis the term IDPS is used whenever these differences are not important. Three other uses from the publication was also mentioned:

- **Security policy violation.** For example if certain traffic should be

blocked by a firewall, but is not, the IDPS system might alert the administrators.

- **Documenting existing threats to an organization.** Meaning that an IDPS will log and alert the administrators about threats it detects. Understanding the threats will help in identifying which security measures need to be applied to which resources.
- **Deterring individuals from violating security policies.** If individuals are being monitored by a IDPS system they may be less likely to commit violations against the security policies because of the risk of detection.

The thesis will focus on the primary tasks of IDPS, which is the detection and possible prevention of attacks. When an IDPS responds against a threat, it can use several response techniques, again from the NIST special publication[50]:

- **The IDPS stops the attack itself.** This can be done, for example by terminating the network connection or user session used for the attack. Block access to the target from the offending user account, IP address or other attack attribute. Block all access to the targeted host, service, application or other resource.
- **The IDPS changes the security environment.** The IDPS can change the configuration of other security controls, for example a firewall, in order to stop an attack.
- **The IDPS changes the attacks content.** The IDPS can change the attack content so it is no longer dangerous. For example by stripping a malicious email attachment from an email before delivering it.

Another point about IDPS is that they are not accurate in their detection. False positives and false negatives are therefore an issue when using these types of systems. In many cases when one lower either false positives or false negatives rates the other type raises, meaning that a decrease in false negatives will most likely increase false positives. Altering the configuration of an IDPS to improve the detection accuracy is called *tuning*. If the IDPS is also responsible to prevent attacks, false positives may have severe consequences. For example if the main web-server is moved from one physical host to another and is then blocked by the IDPS because of too much traffic in a new place, it can be devastating for a business. The organization using IDPS systems must therefore be careful when configuring them to prevent them from harming the business in case of false positives or false negatives.

There are several techniques used for detecting intrusions, these techniques

are used in varying degree across all the different types of intrusion detection systems including web application firewall.

- **Signature-Based Detection.** A signature is a pattern that corresponds to a known threat. Signature-based detection compares the events the IDSP observe against a set of known signatures. The strength is to identify known threats with high accuracy, but the weakness is it can not detect unknown threats.
- **Anomaly-Based Detection.** An anomaly-based detection compares what it considers normal behavior and alerts when it detects abnormal behavior. Anomaly-based detection will be discussed more in Subsection 2.2.2
- **Stateful Protocol Analysis.** Stateful protocol analysis¹ is the process of comparing predetermined profiles of generally accepted definitions of benign protocol activity for each protocol state against observed events to identify deviations.

There are also several types of IDPS systems. These types of IDPS systems have different areas which they monitors and in the publication it is recommended to use a variety of these types in order to cover most cases. In the Nist special publication these four are mentioned:

- **Network-Based** - Monitors network traffic for some or all parts of a network. It analyses the network and application protocol activity to identify suspicious activity.
- **Wireless** - Monitors wireless traffic and analyzes it to identify suspicious activity
- **Network Behavior Analysis** - Examines network traffic in terms of flow and identify attacks based on network flow, such as distributed denial of service attacks (DDoS) or other malicious network activities.
- **Host-Based** - Monitors a single host and the events occurring within that particular host for suspicious activity.

2.2 Detection Techniques in IDPS systems

As mentioned in Section 2.1 there are several techniques or methodologies in detecting threats. This section will walk through them in more detail and

¹Stateful protocol analysis is sometimes called *deep packet inspection* by some vendors.

explain their advantages and disadvantages.

2.2.1 Signature-Based Detection

Signature-based detection compares the signatures of detected events against a list of known malicious signatures. This method for detection can be very effective against known threats, but ineffective against unknown threats. Another weakness is that known threats can use evasion techniques, such as changing slightly, giving it a new signature but still performs the old malicious action. A simple example can be an email-attachment named freekisses.exe is malicious and the IDS have a signature looking for freekisses.exe. If the attacker where to change the filename to for example freekissessuperversion.exe the signatures would no longer match, but the executable file would still be the same².

These limitations make it trivial for an attacker to circumvent the signature-based detection, however it creates a cat and mouse game between the IDS updating its signatures and the attacker changing the malware. Another strength is that there are fewer false positives when doing pure signature-based detection.

2.2.2 Anomaly-Based Detection

Anomaly-based detection is to compare what is called a normal state of behavior against the observed behaviors and check for significant deviations from what is considered normal. An IDPS system which uses anomaly-based detection must have a profile of what it considers normal³. Such a profile is created by having a training phase were the IDPS system learns what is normal behavior in the network it is monitoring. This learning phase is often done for some time so the IDPS learns typical networks trends, for example work-hours and backup traffic at night. Such a profile can either be static or dynamic. A static profile never changes unless the IDPS generates a new profile or updates the old one. A dynamic profile is constantly adjusting its consideration of normal behavior based on the events observed. The benefit

²A more sophisticated signature could look into the file itself, but the principle is still the same. Change something unimportant so that it does not match anymore but the malware is still functional.

³Normal is different from each case, it can be network traffic, normal hosts and the traffic between them etc.

of a dynamic profile is that it will easier cope with small changes, for example new workstations added to the network. Because of this a static profile will be inaccurate over time and there would be a need to reconfigure from time to time. This problem is avoided using dynamic profiling. A weakness by using a dynamic profile is that an attacker can perform an attack but start slowly and then increase the attack as time goes by. By doing this the dynamic profile will be updated to include the attack, one step at a time until the attack is considered normal behavior.

The anomaly-based detection mode biggest advantage is its ability to detect previously unknown threats. This because it will block all behavior that is not considered normal and an attack will most of the time not be considered as normal behavior. The biggest downside is that often, what is normal behavior will be flagged as abnormal by the anomaly-based detection method. For example a user might distribute a large email attachment to many of his coworkers, which might be considered spam and blocked by the anomaly-based detection method, resulting in a false positive. On the other hand an attacker might also trick it, either by slowly performing the attack against a dynamic profiling or it might be that the attack is not distinguished from normal behavior, resulting in a false negative.

2.2.3 Stateful Protocol Analysis

A stateful protocol analysis is the process of analyzing network traffic against predetermined profiles and rules for behavior based on the protocol state. The biggest difference between stateful protocol analysis and anomaly-based detection is that unlike anomaly detection, which uses host or network specific profiles, stateful protocol analysis relies on well known standards for how specific protocols should be used. This is typically based on the Internet Engineering Taskforce(IETF) Requests for Comments (RFC) or form protocols developed by software vendors. The word stateful in stateful protocol analysis means that the analysis tracks and changes depending on the state the protocol is in. For example when a user connects to a server using the Transport Layer Security (TLS) protocol, the protocol starts in handshake mode, where the server is authenticated. Only when the handshake is done the protocol is allowed to send data. If an attacker tries to send data before the handshake is complete a stateful protocol analysis would detect this and count the activity as suspicious.

The protocol analysis part is often done by performing reasonable checks for the protocol, such as checking the argument lengths used in the protocol. If

a threshold which is not in the specification of the protocol or just abnormal behavior, the stateful protocol analysis might flag the activity as suspicious.

There are several drawbacks with using stateful protocol analysis. The biggest drawback is that stateful protocol analysis is very resource intensive, which means that if it is placed inline in a network it can easily become a bottleneck, especially when tracking many different protocols simultaneously. Another drawback is that often vendors might differ from the standard protocol, either because of bugs or just to have their own optimizations of a protocol. This makes it difficult for a stateful protocol analysis to tell the difference between harmful deviation and benign deviation from the protocol. The last drawback is that stateful protocol analysis can not detect attacks which do not violate a protocol specification, for example an SQL-injection attack against a webserver can be delivered over a perfectly legal HTTP-protocol request.

2.3 Summary Network IDPS

The types of network IDPS systems presented here provide an overview of the IDPS systems which work at the network level, up to the 7 layer in the International standards organizations(ISO) Open Systems Interconnection (OSI) model. However they do not always inspect the data carried to the application, which other detection systems does. The different types of IDPS systems looked at here are mostly complementary to each other, meaning that an organization can deploy all four kinds if it wants to be fully covered with IDPS technology.

2.4 Web Application Firewalls

Web Application Firewalls(WAF) are firewalls that operates at the application layer from the ISO OSI model. They intercept traffic going to a web application and understand HTTP and web applications at a much deeper level than network IDPS systems⁴. The main benefit is that the WAF is able to understand how a web application works. For example tracking state through cookies rather than the stateless HTTP protocol itself. A WAF is also typically

⁴Many vendors have IDPS systems that does many of the same HTTP analysis as a Web Application Firewall, the lines between network IDPS and WAF are therefore often much more blurry in practice.

not dependent on the type of web application but can be placed in front of any type of web application.

2.4.1 Usage of Terms

In this thesis, a WAF means any web application level firewall that is in front of a web application in some fashion. This can be either on the same host, as part of the web server serving the application or a standalone appliance that can be placed anywhere in the network in the path to the web application. A WAF can also be a hardware appliance or a software component that can be installed on an existing system. The Web Application Security Consortium (WASC) in their document [33] have several examples of how a WAF can be configured in front of a web application.

A WAF can utilize several detection techniques; some of these have a lot in common with the techniques used in IDPS systems. The WAF can use a *negative security model* that means all transactions are allowed by default. The transactions which contain attacks are rejected. This method can use either signature-based, just as with IDPS or rule-based, which is more complex logic added to the signatures. *Positive security model* means that all transactions are denied by default and only those who are known to be valid are trusted. These two methods are analogous to a firewall, configured to use either a black or white-list approach.

2.4.2 WAF Features

The primary function of a WAF is to protect a web application against detected vulnerabilities and to prevent them from being exploited by attackers. The basic method for doing this is by a blacklisting approach which filters known attacks against a web application, for example to automatically sanitize a request containing an SQL-injection. In addition to support black listing, which is similar to Signature-based detection described in 2.2.1 a WAF can also support white listing or anomaly-based detection. The white listing approach is similar to the anomaly-based detection, described in Subsection 2.2.2. Both requires a learning period where they learn how the network traffic or web application traffic behaves so they are able to detect abnormal behavior and respond accordingly. A summary of the different types of attacks and how effectively a WAF can prevent them is shown in Table 2.1 from [35]. The

table gives an overview of which problems a WAF is good at handling and which it is not as good at handling.

The WAF can also perform normalization of the traffic in order to be able to detect if an attacker is trying to masquerade the attack payload in a way that looks harmless, but when decoded by the web application executes the payload.

A WAF can, in addition to protecting the web application, handle some of the work currently done by the web application. For example it can help by logging relevant security information from the normal traffic flow. The web application firewall can handle cookies, which are used to track state and login information and prevent an attacker from tampering with them or prevent information leakage, with for example using cookie encryption. By outsourcing some of this work to a WAF the web application can free up resources to handle the business logic more effectively.

Table 2.1: WAF attacks

+ very well covered by a WAF

- can not be covered by a WAF

! dependent on the WAF or application

= can partially be covered by a WAF

Problem	WAF	Countermeasure
Cookie protection	+ + ! !	Cookies can be signed Cookies can be encrypted Cookies can be completely hidden or replaced (Cookie Store) Cookies can be linked to the client IP.
Information leakage	+	Cloaking filter, outgoing pages can be cleaned (error messages, comments, undesirable information).
Session riding (CSRF)	+	URL encryption/token
Session Timeout	!	Timeout for active and inactive (idle) sessions can be specified (if the WAF can manage the sessions itself). Even if the sessions are managed by the application, the WAF can detect these and terminate them with the appropriate configuration.
Continues on next page		

Problem	WAF	Countermeasure
Session fixation	=	Can be prevented if the WAF manages the sessions itself
Session hijacking	-	Difficult to prevent, although the WAF can issue an alarm in the event of irregularities (e.g. changing IP) or terminate a session with changing IP.
File upload	+	Virus check (generally via external systems) via ICAP linked to the WAF.
Parameter tampering	+ +	In addition to/instead of data validation (see below), parameter manipulation can be prevented via URL encryption (GET) and parameter encryption (GET and POST). Site usage enforcement, meaning the possible sequence of URLs can be fixed or can be detected
Forced Browsing	+ +	Can be prevented via URL encryption. Site usage enforcement
Path traversal (URL) link validation	+ +	Can be prevented via URL encryption. Site usage enforcement .
Path traversal (parameter), path manipulation	+	See parameter tampering and data validation.
Logging	+	All or only specific/permitted parts of the data of a request and of the connected tests can be logged.
Priv. escalation	-	Privilege escalation can not be checked or can only be checked to a limited degree, for example via cookie/parameter encryption.
Logical level	-	Application logic going beyond the validity of URLs and form fields, can not normally be checked by a WAF.
Continues on next page		

Problem	WAF	Countermeasure
Anti-automation	=	Automatic attacks can be partially detected and blocked (e.g. number of requests/time interval, identical requests, etc.).
Application DoS (moderate)	= =	Transactions, IPs, and/or users can be blocked. Connections and/or sessions can be ended .
SSL	+ + +	WAF can force SSL with pre-defined encryption strength (depending on the infrastructure scenario). SSL termination on the WAF, forwarding of the SSL data (e.g. client certificate) to application. SSL connection possible from WAF to application.
Data validation (relating to field/content/context/appl)	+ + ! -	Can be tested to very detailed degree (length, constant value/range of values, e.g. for SELECT, character area); validation possible with whitelist and/or blacklist (signature). Rules can in part be generated automatically. High dependency on application, specific fields (hidden form) or pre-defined parameters in the URL; can be automatically verified by the WAF however. Risk due to false positives, problematic with business critical applications in particular.
Data validation (general/global)	+	HTTP(w3c) conformity, a WAF conducts a canonicalisation of the data so that it is available to the application in a standardised form.
Buffer overflow	+	See data validation[1]
Continues on next page		

Problem	WAF	Countermeasure
Format string attack	=	Can be detected using data validation if the corresponding characters or strings are filtered (difficult in practice, as precise knowledge of the application is required to do this). For the majority of the hidden input fields, this can be carried out without knowledge of the application.
Cross-site scripting	=	Using data validation, only reflected XSS can be detected and prevented, persistent XSS can not be detected, DOM-based XSS only to a limited degree if part of the attack is sent in parameters of the request.
Cross-site tracing	+	Restriction of the HTTP method to, for example GET or POST.
WebDAV	+	Restriction to only reading WebDAV methods possible
Code injection (PHP, perl, java)	+	See data validation [1]
Command injection	+	See data validation [1]
SQL injection	+	See data validation [1]
LDAP injection	+	See data validation [1]
XML/Xpath injection	+	See data validation [1]
Just-in-time patching (hotfix patching)	+	Using data validation (see above), the WAF can protect against newly detected vulnerabilities and/or attacks (Zero Day Exploit).
HTTP response splitting (HTTP splitting)	!	Can only be detected using data validation in URL and/or parameters if %0d%0a is filtered - however this can be carried out on virtually any input field without impairing the functionality of the application.
HTTP request smuggling	+	Is prevented via strict testing of the conformity to standards of each request.

2.4.3 WAF in a Network

A WAF can have several placements in a network. In general there is a differences between *passive* and *active* mode. In a passive mode, the WAF only gets a copy of the traffic to the web-server, via for example a network monitoring port. It is harder for a WAF to respond to an attack while being placed in passive mode, but it can still communicate with other network equipment or the web-server itself in order to prevent certain connections. The main advantage with passive mode is that the performance of the WAF would not have any impact on the web-server and another layer with proxying is avoided. Active mode places the WAF inline with the traffic going to the web-server. The main advantage of placing a WAF inline is that the WAF itself is able to drop connections and filter content before it arrives at the web-server. There are several drawbacks. The WAF can become a performance bottleneck for the web-application or the WAF can become a single point of failure. In addition the network become more complex and thereby more difficult to handle. Despite these drawbacks a WAF is most often used in inline mode, since there are many techniques to avoid these drawbacks, such as having multiple redundant WAFs and sharing the load between them.

When placing a WAF in active mode inline in a network, there are several ways to do this. The Web Application Firewall Evaluation Criteria[33] lists four different active modes.

- Bridge. The WAF is installed as a transparent bridge, often called transparent proxy
- Router. The network traffic is configured to be directed through the WAF
- Reverse Proxy. The traffic is directed through the WAF either by DNS-changes or network level changes
- Embedded. The WAF is embedded in the web-server or directly in front. Sometimes called host-based WAF

In a big corporate network there are many other configurations possible, but these cover the most used methods. The two biggest differences is between an inline fashion, meaning that the WAF is somewhere in the traffic line or out-of-band, called passive mode. These are shown in the figures 2.1 which show a sample inline configuration and 2.2 which shows a sample out-of-band configuration.

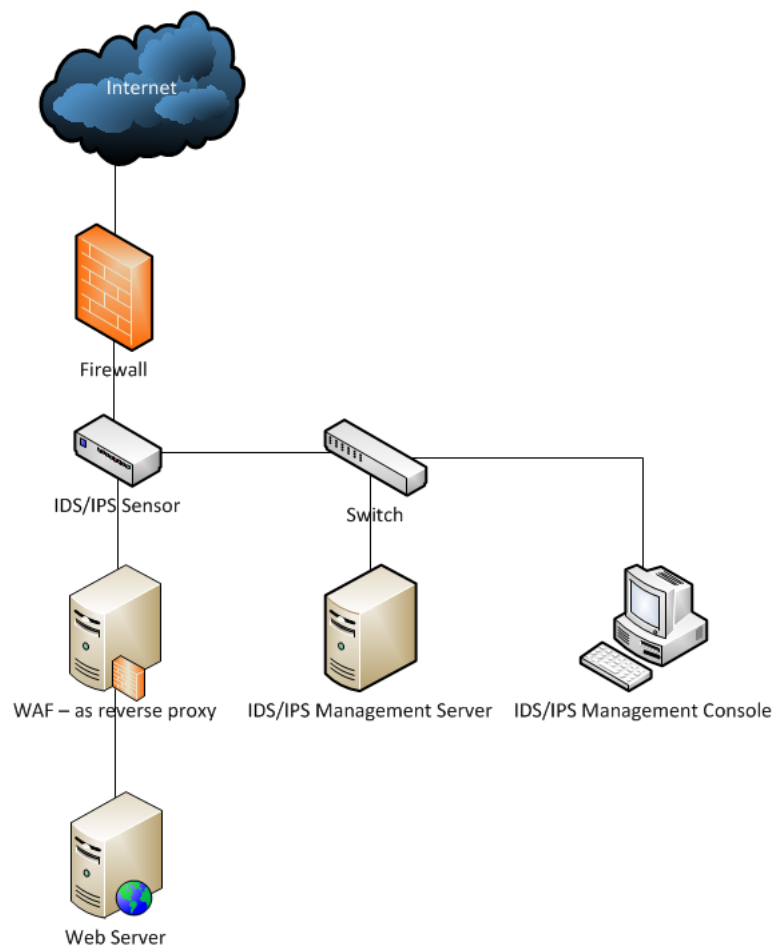


Figure 2.1: WAF and IDPS both setup in a inline fashion

There are many advantages by placing a WAF inline in the traffic. The biggest advantage is that the WAF itself can block traffic it consider suspicious. In addition to easily block the traffic the WAF can help the web application in several other ways as Jim Beechey points out in his sans paper [30], as shown below.

- Caching - Reducing the load on the web server by caching static and much used content
- Compression - The WAF can compress the content before delivery, so the web server do not have to perform the compression
- SSL Acceleration - A WAF can utilize hardware accelerated SSL processing, however the WAF needs a copy of the domains SSL-keys

- Load balancing - The WAF can spread the incoming connections among several web servers, removing the need for an extra load balancer
- Connection Pooling - This technique reduces the amount of new TCP-connection needing to be established with the web servers.

The biggest disadvantage in inline mode is that the WAF can become a bottleneck and it also features yet another proxy before reaching the web-server, which do complicate the setup.

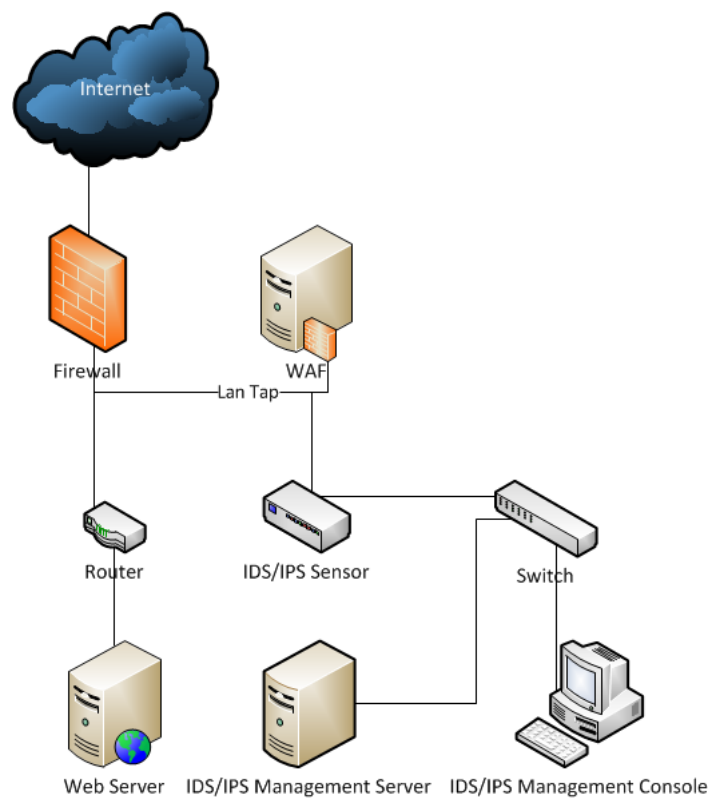


Figure 2.2: WAF and IDPS both setup in a monitor/out-of-band mode

When placing the WAF in passive out-of-band mode, it is harder for the WAF to perform blocking of suspicious traffic. However it can send packet to terminate TCP-connections or be configured to get the firewall in front to block the traffic. The main advantage is that the WAF is not in the danger of becoming a bottleneck and the rest of the network will function normally even of the WAF crashes. This mode is also ideal for testing the WAF on real network traffic or for learning a anomaly detection-based WAF.

2.5 Defense in Depth

Defense in depth is a security principle. The principle is that there should be more than one method for securing something. Bruce Schneier has a simple explanation for the defense in depth principle in an article from 2000 [51].

Provide Defense in Depth. Don't rely on single solutions. Use multiple complementary security products, so that a failure in one does not mean total insecurity. This might mean a firewall, an intrusion detection system and strong authentication on important servers.

The defense in depth principle is not only a technological principle, but affects both people, technology and operations as NSA points out in their defense in depth strategy paper [28]. They also point out several ways of performing defense in depth. The two most important ones for this thesis is listed below:

- Defense in Multiple Places. Meaning that an organization need to deploy security mechanisms in different places.
- Layered Defenses. This means to have many layers of defense. For example to have both a corporate firewall and also a software firewall on each host.

When it comes to the usage of firewalls, IDPS and WAFs the layered defenses method is one to consider, since many of these products overlap in terms of functionality. However it is for example only the WAF which understands the web application context in layer 7 of the ISO OSI model, while both a corporate firewall and IDPS system protects at a network level. They also have different responsibilities. A more thorough example of a layered defense would be the usage of both network level IDPS and in addition host-based IDPS systems. That way a threat, which slips the network level IDS can still be detected by the host-based IDS.

Defense in depth is used because there is no definitive way of securing yourself. There is no silver bullet as Leight and Hammer points out in [43]. They also mention several approaches to defense in depth. *Uniform protection* treats all systems as equally important, meaning that no special consideration or protection is made for different systems. This is the most common approach to defense in depth and is quite useful but weak against attacks targeted at special assets in the organization, the so called crown jewels. *Protected enclaves* approach involves separating the network into different separate networks. This is typically done with networking equipment such as firewall, VLANs,

VPNs and so on. *Vector Oriented* defense in depth involves identifying the vectors which threats can become manifest and deploy security mechanisms to shut down the vector. *Information Centric* defense in depth means to defend the information or any other values that needs protection. And then the defense is build around that information so that in order to get the information, each layer needs to be penetrated. In this thesis, which focuses mostly on web application the information centric approach will be used since it is assumed that the values that needs protection are in the web application database and accessible through the web application and therefore needs good protection.

In this thesis, a comparison of a web application firewall and OWASP AppSensor will be performed. The different attacks is evaluated if it should be prevented by a WAF, in the application itself with the help of AppSensor or according to the defense in depth principle should be blocked both places.

2.6 OWASP AppSensor

The OWASP AppSensor project[32] is a project which aims to help developers of custom web application to detect and prevent attacks which can happen at the business logic. The idea behind OWASP AppSensor project is that it is not a product, like a WAF would be, but rather an idea with a reference implementation in a Java framework that can tie onto OWASP ESAPI [14]. The idea is that while a WAF can protect against generic attacks for example a SQL-injection, it can not protect against business logic attacks against a custom web application, such as a POST-request against a field which should only use GET.

The idea is realized by having two parts, one part is the detection sensors in the application, which detects possible attacks and the other part is the response mechanisms, which makes appropriate responses to the attacks. These two parts are shown in figure 2.3 for a 3-layer architecture. The detection points are custom made into the application by the developers, but can use many of the already made detection points in the AppSensor framework or extend them with their own.

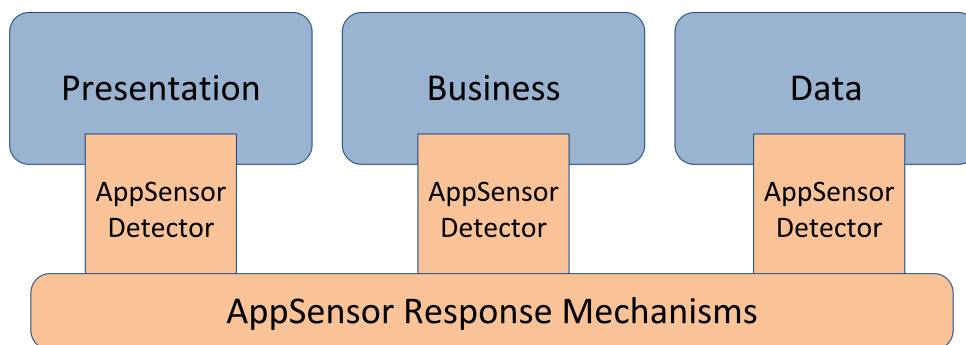


Figure 2.3: Simple Architecture diagram from[32]

The OWASP AppSensor project consists of both the idea and concept of using detection sensors in a web-application as well as a sample implementation of the AppSensor framework in Java. The concept of using detection points in an application to see what is going on is the heart of the project. It is a very powerful concept, just as a bank have surveillance cameras in addition to having a huge vault with a time-based lock. Because in addition to being secure by having a time-based lock the bank also would want to identify potential attacks well ahead of time and react to suspicious activity. A typical web-application can be coded securely and resist attacks, however by deploying sensors in the application. The application itself and its administrators, is able to know what is going on in real-time inside the application. This information can be valuable since the application or its administrators, are able to block an attacker before he can actually find a potential weakness in the application. The application can also be able to turn of some functionality or to just block one user, thereby still functioning during an attack, which helps in the availability of the service the application is serving.

AppSensor might have some of the problems traditional IDPS and WAFs have, which is both false positives and false negatives. However OWASP in their paper about AppSensor[32] argue that since it is so close to the application and its logic, the rate of false positives can be dramatically reduced. This is because some behavior can never occur when a normal user using the application. An example from OWASP in their AppSensor paper [32] is a GET request when the application expects a POST request. Another example is tampering with the data within a POST-request, which requires explicit attempts from a user to do so and would not happen in a normal use case. Such actions can therefore be detected with a high degree of confidence that the user is malicious. On the other side, AppSensor might detect that an user for example is putting a "" inside a username field. It might be an

attacker looking for a SQL-injection but it can also be just a typo from a user. This is where AppSensor will have the same problem as an IDPS or WAF in detecting what is actually an attack and what is just unusual but benign behavior.

2.6.1 AppSensor Usage

Currently OWASP AppSensor is used by several companies as several of the AppSensor projects participants points out in their crosstalk article from 2011 [57]. Since OWASP AppSensor is both a concept as well as a simple framework, with the idea of being extended and tailored for the custom application implementing it. Many companies mentioned in the crosstalk both extended the OWASP implementation and some build it themselves from scratch. As said in the crosstalk article, one adopter uses it heavily integrated with a Security information and event management (SIEM) system. The main point is that there are many possibilities in how to use AppSensors and while the project itself is a simple example implementation, it is meant for the adopters to extend its use. In a talk given by Michael Coates⁵ he points out that at Mozilla, they use the AppSensor principles and ideas while they are not based on the AppSensor projects code base. They have their own standalone implementation used in their web-applications. This points out that the AppSensor project is mostly an idea and a concept and secondly provides a reference implementation in Java, which can be used in Java-based web-applications.

2.6.2 OWASP AppSensor Detection Points

The OWASP AppSensor project comes with many detection points out of the box. These are the generic detection points supplied with the AppSensor project, but it is easy to create other detection points in addition to the ones provided by the OWASP project. These detection points must be put to use inside a custom application and are provided since they are quite generic. In the OWASP AppSensor paper[32] is a detailed description of the different detection points. In Table 2.2 the different detection points from the paper is listed.

⁵<http://vimeo.com/15726323>

Table 2.2: Table from [32] with the detection points

	ID	Event
Request	RE1	Unexpected HTTP Commands
	RE2	Attempts To Invoke Unsupported HTTP Methods
	RE3	GET When Expecting POST
	RE4	POST When Expecting GET
Authentication	AE1	Use of Multiple Usernames
	AE2	Multiple Failed Passwords
	AE3	High Rate Of Login Attempts
	AE4	Unexpected Quantity Of Characters In Username
	AE5	Unexpected Quantity Of Characters In Password
	AE6	Unexpected Types Of Characters In Username
	AE7	Unexpected Types Of Characters In Password
	AE8	Providing Only The Username
	AE9	Providing Only the Password
	AE10	Adding Additional POST Variables
	AE11	Removing POST Variables
Session	SE1	Modifying Existing Cookies
	SE2	Adding New Cookies
	SE3	Deleting Existing Cookies
	SE4	Substituting Another Users Valid Session ID Or Cookie
	SE5	Source IP Address Changes During Session
	SE6	Change Of User Agent Mid Session
Access Control	ACE1	Modifying URL Arguments Within A GET For Direct Object Access Attempts
	ACE2	Modifying Parameters Within A POST For Direct Object Access Attempts
	ACE3	Force Browsing Attempts
	ACE4	Evading Presentation Access Control Through Custom Posts
Input	IE1	Cross Site Scripting Attempt
	EE2	Violations Of Implemented White Lists
Encoding	EE1	Double Encoded Characters
Continues on next page		

	ID	Event
	EE2	Unexpected Encoding Used
Command Injection	CIE1	Blacklist Inspection For Common SQL Injection Values
	CIE2	Detect Abnormal Quantity Of Returned Records
	CIE3	Null Byte Character In File Request
	CIE4	Carriage Return Or Line Feed Character In File Request
File IO	FIO1	Detect Large Individual Files
	FIO2	Detect Large Number Of File Uploads
User Trend	UT1	Irregular Use Of Application
	UT2	Speed Of Application Use
	UT3	Frequency Of Site Use
	UT4	Frequency Of Feature Use
System Trend	STE1	High Number Of Logouts Across The Site
	STE2	High Number Of Logins Across The Site
	STE3	High Number Of Same Transaction Across The Site

The detection points in Table 2.2 must be placed inside a web application at appropriate places. For example to place a RE3 detection point requires knowledge about the web application and it is easier to implement when the application is being developed, than in hindsight. This is probably the biggest difference between AppSensor and both IDPS and WAF systems, it is not just a drop-in system that can be added to a web application, it has to be built in.

These detection points are only few of many possible. In a custom application it is possible to make custom detection points which understands the applications logic. Another big feature is that an implementer is able to specify both a custom detection point but also the threshold of either the custom detection point or some of the other existing ones. For example an online webshop might make a detection point that detects if a user trying to pay with many different credit cards which gets refused. This could signal a user that is trying to use stolen credit cards to pay for the merchandise in the webshop.

2.6.3 AppSensor Response Actions

The AppSensor project also contains a list over default response actions which an AppSensor powered application can use. These actions will be performed when a certain threshold is reached. The thresholds are counters for how many different detection points are triggered and is also defined by the implementer. For example one action can be to log out the user when he has tried 6 attempts to perform a XSS-injection. Another action can be to block the users IP-address if he tries XSS-injection again after logging back in.

The AppSensor project contains four levels it can use when taking action, *silent*, *passive*, *active* and *intrusive*. The silent actions does not alert the user but silently logs, alerts the application administrators or other silent actions. Passive actions notifies the user, but does not prevent him in any way from using the web application. The notification might be an alert, an email or the application might slow the requests from that user down. Active actions often deny the user access. It can be achieved by logging the user out, disable the user account, disable the component under attack, block the user from the application or other means of denying access. The last ones are intrusive and these actions can for example log information about the user such as their remote IP-address, browser signatures and other information. These intrusive actions are meant to be non-malicious but one got to be careful so these actions does not do something illegal. The response actions which the AppSensor team has come up with are in Table 2.3 which is a copy of the one from the OWASP wiki.

Not all the response actions in Table 2.3 are implemented in the OWASP AppSensor framework. The table is made to give users of either the AppSensor concept or the framework an idea of which response actions it is possible to take against a malicious user. The default response actions which are provided by the framework is to log, logout, disable the user account, disable a component either completely or just for a particular user, email an administrator or send an sms to an administrator through email to sms. These are the response actions which are included in the framework from the OWASP project, but as seen in Table 2.3 there are many other possibilities for response actions.

Table 2.3: AppSensor Response Actions

Type	ID	Description
Silent	ASR-A	Logging Change
	ASR-B	Administrator Notification
	ASR-C	Other Notification
	ASR-N	Proxy
Passive	ASR-D	User Status Change
	ASR-E	User Notification
	ASR-F	Timing Change
Active	ASR-G	Process Terminated
	ASR-H	Function Amended
	ASR-I	Function Disabled
	ASR-J	Account Logout
	ASR-K	Account Lockout
	ASR-L	Application Disabled
Intrusive	ASR-M	Collect Data from User

2.7 Other Systems for Defeating Web Application Attacks

There have been many other attempts to defeat web application attacks and while many of them resemble a WAF or IDPS they are not labeling themselves as such. Often there are differences which will not make them justice by calling them a WAF or IDPS. The reason they are included is to give a view of the size of the research that have gone into solving the security problem for web applications.

2.7.1 Language Defense

There have been many attempts to make language themselves protect against misbehavior. James Morris provided a means of protecting objects in languages as far back as 1973 [45]. Languages used on the web today which are often either interpreted (Perl, Python, Ruby) or run on a virtual machine (C#, Java), called managed code is almost immune toward buffer overflows which have been pestering languages such as C and C++ where the developers themselves have to check boundary conditions. The biggest danger modern languages faces is the handling of untrusted data and then deliver those data to another system or language. One example is to deliver untrusted data

from a HTML form to a database interpreter which then interprets the SQL and if the data contains SQL-commands they will be executed.

A feature of some programming languages or frameworks is *taint-tracking*. Most famously taint-tracking have been a feature of Perl [17], which makes Perl mark input from an external entity as tainted [18]. If the tainted data is used for example as an argument to a shell command, Perl will exit with an error of the data passed to the command is tainted and has not been properly cleaned. This idea of tainted data have inspired many other works such as the black box approach from R. Sekar[52], which places itself between the different web server components making it a drop in solution. Another approach that patched the PHP-interpreter itself in order to prevent injection attacks is presented by Tadeusz Pietraszek and Chris Vanden Berghe in their paper from 2005[47]. As mentioned in [47], there are many other methods that involve either changes to the language run-time or the platform or environment of the web application in order to have a generic way of preventing injection attacks.

2.7.2 Platform Defense

Many modern platforms operating systems contain some level of built in protection. While the platform defenses are targeted at a broader range of attacks than just web application attacks, some of the defenses can still improve the security of the web application. The use of mandatory access controls, which now is part of Microsoft Windows VistaTM and Microsoft Windows Server 2008TM and above called mandatory integrity control[44], is one such mechanism. This helps in separating processes from each other and can help prevent for example a command injection attack to be able to remove data from another place in the system because of limited access⁶.

2.7.3 Web Application Frameworks

Most web applications today are made by using web application frameworks to provide much standard functionality, like easy creation of forms, object-relational mappers (ORM) [10], authentication and authorization, handling of sessions and many other features. The benefits is that the developers

⁶Standard UNIX have had separate users from the beginning. In addition there are tools like AppArmor [2] and SE-Linux [21] and in some BSD-based systems the mac from [8] can be used.

can focus on writing business logic for their application while the standard plumbing is taken care of by the web application framework. Many security features can also be present in a framework, making it easy for a developer to get some protection automatically into their application without having to add any custom code. For example many ORMs does sanitation of the input before sending it to the database, thereby preventing SQL-injections. Other features can be that session-handling is managed in a secure manner by default. These frameworks are not perfect and while they can have some features, not every framework have them. So in order to use a web application framework effectively the developer needs to research what type of security features the framework contains and which it does not. Another point is that the framework itself can have bugs or errors making it vulnerable to exploits, so it is still advisable to have other layers in front of the web application despite the use of security features in a web application framework. An example of a framework that has extensively focus on security is OWASP Enterprise Security API [14].

While there are many approaches to protecting web applications from attacks, this thesis will not go any further into them. The scope of the thesis is to compare AppSensor, WAFs and IDPS and while these methods deserve a mention, they will not be used in the comparison. This is not because they are not useful, they certainly are, but the scope of the thesis was decided to leave these methods out to get a better comparison of the other three.

Chapter 3

Research Method

This chapter explains how the research questions were answered, why an experiment was performed and explains how the results was achieved. The chapter starts with listing up different research methods and strategies for answering the research questions.

The last part of this chapter will explain how the chosen method, an experiment, was set up and performed. It will walk trough the setup in detail in order to ensure as much reproducibility as possible.

3.1 Research Strategy

There are many different research strategies. When choosing which types of strategy to perform, a recap of the research questions is in order. The research questions will serve as a foundation for which research strategies will be chosen.

- RQ1: What is the current state of application-based intrusion detection and prevention systems?
- RQ2: How does OWASP AppSensor compare to other IDPS technologies?
- RQ3: In the given scenario, what are the benefits of using AppSensors to stop the attacks compared to a IDPS or web-application firewall?
- RQ4: How hard is it to built AppSensor into an application?

The first research question is answered with a simple literature review. Because of the time constraints of the thesis the literature review is not a complete structured literature review, but a short and simple one. The literature review gives an overview of the different application-based intrusion detection and prevention systems, this involves AppSensor, traditional host-based IDPS and other similar systems. Part of this literature study also helped in classifying the different types and see both the similarities and the differences with these systems.

RQ2 is partly answered with a simple theoretical comparison with the basis from the literature study. This question highlights how AppSensor stand out from other solutions based on the literature. However since this is only based on the literature and no studies were found which can approve these claims, another research strategy to answer this question will also be performed, it is highlighted in section 3.2. The third research question addresses this issue directly. The third research question is answered entirely by this other research strategy chosen and will be explained in more detail later.

The fourth and last research question is best answered by performing an analysis on either quantitative or qualitative data. However since the thesis has limited time, this question will be subjectively answered by the research strategy chosen to answer research question 2 and 3. Since there is not enough time to perform a proper quantitative or qualitative strategy. The answer to this question will therefore be very subjective and is a candidate for further research.

3.2 Different Research Methods

For the overall goal of the thesis there are several different research strategies that can be used to achieve it. These research strategies along with their strengths and weaknesses will be gone through in this section. The one chosen is the one that fits most of the research questions.

3.2.1 Qualitative and Quantitative Strategy

These two methods are well suited for data collecting studies. This involves questions which can be answered through collecting data in some form, either through a survey, interviews, workshops and so on. The types of questions which can be answered are the types of Is there a significant improvement by

using method A over method B, How is the current state of software security in the Norwegian industry. Research Questions 4, which asks, how hard is it to built AppSensor into an application. The word hard here generally means cost in time needed to implement AppSensor. However the time needed to implement AppSensor can be quite complex, because it relies on the skill of the developer, the size of the application, type of detection point and many other factors. The research questions could have been answered through a series of interviews or a survey of people using AppSensor as well as other technologies. However since AppSensor is not widely adopted yet, it would be difficult to get enough participants, which would give significant results. This makes it hard to answer any of the research questions with the use of data collecting methods. Since these methods would be almost impossible to perform, they will not be used in this thesis. But they could very well have been used if there was enough data to be obtained.

There are many other types of qualitative and quantitative strategies that could have been applied if there was any dataset or it was believed that it would be easy to get the data ourselves. These methods will not be explored any more in this thesis since it seems that the current stage of AppSensor implementations are quite few. So far the idea of building sensors or other application level defenses into an application seems relatively new.

3.2.2 Experiments

Experiments in computer science can be performed much like experiments in other scientific disciplines. As pointed out in [37] the method is mostly the same. This method is often called the scientific method [36] and involves a series of steps, as shown in Figure 3.1, but in for example systems design, you start with an idea, then perform the system design, make a concrete implementation and have an experimental evaluation at the end. This however is mainly just a variant of the classical scientific method suited for example for systems design.

Experiments is according to this article by Tichy, W.F. [55] encouraged. Fred Brooks however claims the opposite in [31] which suggested that computer science is “not a science, but a synthetic, an engineering discipline”. The argument from Fred Brooks is that computer science is just an engineering discipline and not a science. This point is argued against by Tichy [55] where he argues that while computers are made by humans, and not something pure which can only be observed in a natural form, it can still be studied. He argues that in order to study lasers, we need to construct lasers, even tough

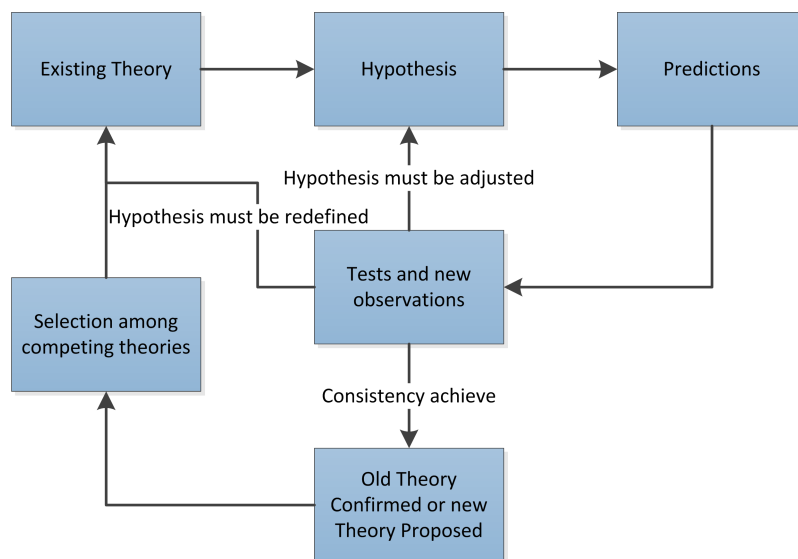


Figure 3.1: Scientific Method, from [36]

they do not appear naturally, and that the same principle is with computer science. The model we can make by studying this laser does not make it unscientific, quite the opposite. Tichy argues in his paper that experiments using the traditional scientific form, is both valuable and helps in testing theories. The traditional scientific approach for experiments is the following.

- Form a hypothesis
- Construct a model and make a prediction
- Design an experiment and collect data
- Analyze results
- If results are satisfactory, propose new theory

Experiments are not proving anything, unlike for example a mathematical proof[36] but they can be valuable because they test theories in a way so the experiment might suggest a given theory is wrong. Experiments can also lead to new insights and suggest new areas of exploration and more research. A simple experiment can also quickly show if some approach is fruitless or not. These benefits makes experiments worthwhile in scientific research, but in order to eliminate error in the experiment, it should be reproducible by other researchers so an independent researcher can verify the results in order to minimize the amount for errors in the experiment.

A weakness with experiments is that almost all experiments are, as Paul Feyerabend points out in his book [38], theory contaminated. This means that sometimes the experiment is made to fit the theory it is testing. This makes it hard to tell if the experiment is just a self fulfilling prophecy or if it actually is objective enough to be able to either help in confirming the theory or find holes in the theory. In order to have the experiments be as objective as possible it needs to be done with as few variables as possible, and the environment must be examined carefully before the experiment takes place. This is one of the reasons why experiments are important to be reproducible. The importance is mentioned by both, Tichy [55], Feitelson [37] and by Dodig-Crnkovic [36].

The arguments above suggests that an experiment should be reproducible by others and shall try to be as objective as possible and not be influence too much by the theory, however this is in its nature not avoidable as a total. The experiment performed in this thesis tries to a great extend to fulfill these criteria as much as possible.

3.2.3 Chosen Research Method

The two main methods which is pointed out above are quite different, but both can be used to answer some of the research questions. The experiment fits very well with research question 2 and 3, but not so well with 4. While the quantitative and qualitative methods fits very well for number 4, it does not fit that well with 2 and 3. The chosen method is therefore to perform an experiment; with the note that research question 4 will only be answered very subjectively in this thesis.

As the above section points out, an experiment should try to be as reproducible as possible so that other researchers can verify the results. It should also try to be as objective as possible with regards to theory. The experiment made and performed in this thesis goes to great length to try to achieve these properties.

3.3 Experiment Description

The experiment will be performed in a simulated environment. A scenario that involves a web-application for a fictional banking company will be created, and a set of attacks will be carried out against the web-application. The scenario

Table 3.1: Virtual Server Environment Specifications

Operating System	Ubuntu Server 11.10
Webserver	Apache 2.2
WAF	ModSecurity 2.6
IDPS	Snort 2.8.5
Java Runtime Environment	OpenJDK 6
Java Web Application Server	Tomcat 7
Database Server	SQLite 3.7.7
Attack system	Backtrack Linux 5 R2

is described in detail in chapter 4. The attacks are described in chapter 5 so they are performed the same way each time and are easily reproducible. Since there are no general purpose WAF or IDPS systems, some concrete ones will be used. The choices are ModSecurity [9] which is a module to the Apache Webserver [1] as a WAF and Snort [23] as an IDPS. These choices are mainly made because they are quite big and dominant in the field, and also open source and therefore free of charge to use.

The whole setup will only use one computer with several virtual machines. The virtual machines technology used is Oracle Virtualbox, which hosts both the virtual server, the web application and other protection mechanisms and the machine used for the attacks. This provides several advantages as shown in the section below.

3.3.1 Environment for the Experiment

The experiment will be conducted on a virtual machine acting as a server for the scenario website. The specification for the environment is listed in Table 3.1.

The environment is run on a virtual machine to be easy to set up and the whole virtual machine can be distributed for other researchers wanting to verify the test results or try other attacks against the environment. The usage of virtual machines also has the ability to take snapshots, so in case anything is changed during the testing it is easy to revert back to the previous version and always have a predictable starting point. Another big advantage is that the experiment can be done on an internal virtual network. The network only consists of two machines, the attack machine and the server running the web application and all the security layers. This helps in isolating the server running the web application from all other traffic on the internet and

prevent for example internet background radiation[46] to interfere with the experiment.

3.3.2 Setup of Snort

Snort is installed from the package repository that comes default with ubuntu 11.10. To make the experiment easier, Snort is only configured to be an IDS and only report anomalies and events that it detects and not respond to them. This way it is easy to see if Snort blocks an attack and the attack would still pass through Snort to the next layer of defense. By doing this it is easy to observe that Snort could blocked the attack, but since it is only an IDS it lets the attack pass trough and there is no need to turn Snort off in order to reach into the next security layer with the attack.

The configuration of Snort is the default configuration it comes with. Note here that this is the default configuration from the ubuntu package and that the official Snort download from Sourcefire might be different. The default configuration contains many rules for web-based attacks so Snort should be able to detect some of the attacks that will be performed. Snort is configured to use a MySQL database to store its findings. By storing it in a MySQL database it is easy to have an event analysis system using the database to show the different attacks that Snort detects. In addition to the default configuration, the HTTP-inspect preprocessor is configured to better suite the scenario, by using the apache profile.

To monitor the events which is recorded by Snort, a system called BASE (Basic Analysis and Security Engine) is chosen. It monitors the Snort database and can show the different events happening in Snort through a web gui. This is only to make it easy to see which events is flagged as anomalous by Snort.

3.3.3 Setup of ModSecurity

Modsecurity is, just like Snort, also installed from the ubuntu package repository. The version is 2.6.0 which is the newest 2.x number release of ModSecurity. The core ruleset which is now an OWASP project is also installed, featuring version 2.2.0 of the ruleset. From the core ruleset only the base rules are enabled. Just like Snort, Modsecurity is set up to only detect and not respond to any threats. This makes it possible for attacks to pass through Modsecurity and only be reported in the apache server logs. Just as with

Snort Modsecurity could have blocked the attack, but it is configured to only be reported and then pass through.

Modsecurity can use custom rules that can be tailored to the web application and traffic for a specific site. One example of such a rule is at a webshop site, which have a shopping cart and an amount to pay. A Modsecurity rule, which check if the amount to pay is a positive number, and will alert if there is a negative amount of money in the shopping cart. These kinds of rules have to be custom for an application but it makes Modsecurity able to understand some of the web applications context just like AppSensor.

The Core Rule Set is an Open Source Ruleset maintained by OWASP [53]. The Rule Set uses a negative security model for web applications. This means that the Core Rule Set only detects traffic that are suspicious and otherwise allows everything. This is opposed to a positive security model where nothing but a few chosen set of traffic patterns are allowed. The reason for the Core Rule Set using a negative security model is that it is supposed to be as generic as possible, and the false positive rate would be too high if a generic Rule Set tried to specify which behavior was allowed by a specialized web application. The white paper from OWASP[53] also explains some benefits by using ModSecurity for detecting web application attacks compared to for example Snort which can intercept HTTP-traffic but does not contain as much flexibility in its rule language as ModSecurity does when it comes to HTTP-traffic.

To check the audit log of ModSecurity an application called AuditViewer by JWall [7] is being used. It makes it easier to analyze and look at the logs generated by ModSecurity and also performs grouping of events that are similar. This application has nothing to do with the setup of the experiment and is just a standalone logviewer application to make it easier to see the results from ModSecurity.

3.3.4 Building in the Detection Points

When building in the detection points in the application, the reference implementation of AppSensor from OWASP was used which is hosted on Google Code [4]. The choice of using this implementation was to make it easier for other to reproduce the results, but also to lower the amount of custom implementation needed. However for some projects the reference implementation is neither sufficient, nor complying with the standards of the project or simply not in the same language that a given project is written in. So there

is probably many projects which takes the idea of OWASP AppSensor, but makes their own implementation of it. This could certainly have been done in this experiment, but in order to have it be reproducible and as close to the OWASP AppSensor projects ideas, the reference implementation was chosen.

The reference implementation is not a complete drop in framework, because the detection points and response mechanisms still need to be built into the application. The reference framework handles all the detection points and when they are triggered, it collects them and perform the appropriate responses based on the security policy file, which is a Java properties file. So even tough the reference implementation is used, it only provides a way to build in the detection points and response mechanisms, they still need to be built in manually.

The detection points that were build into the application is covered by the list given in [32], however there were some specific for the scenario application. The detection points are only configured to log events that happen and there are currently no other action the application is able to do other than detecting and logging the attacks. This was mainly because the main comparison will focus on how well AppSensor compares with an IDPS and WAF in terms of detection. However some attention to actions will also be given later in the comparison, but at a theoretical level.

Table 3.2: Which detection points where implemented

Which detection point	Implemented
Unexpected HTTP Commands	Yes
Attempts To Invoke Unsupported HTTP Methods	Yes
GET When Expecting POST	Yes
POST When Expecting GET	Never occur in application
Use Of Multiple Usernames	Yes
Multiple Failed Passwords	No
High Rate Of Login Attempts	Yes
Unexpected Quantity Of Characters In Username	Partly
Unexpected Quantity Of Characters In Password	Partly
Unexpected Types Of Characters In Username	No
Unexpected Types Of Characters In Password	No
Providing Only The Username	Yes
Providing Only The Password	Yes
Adding Additional POST Variables	Yes

Which detection point	Implemented
Removing POST Variables	Yes
Modifying Existing Cookies	Yes
Adding New Cookies	Yes
Deleting Existing Cookies	No
Substituting Another User's Valid Session ID Or Cookie	Yes
Source IP Address Changes During Session	Yes
Change Of User Agent Mid Session	Yes
Modifying URL Arguments Within A GET For Direct Object Access Attempts	Yes
Modifying Parameters Within A POST For Direct Object Access Attempts	Yes
Force Browsing Attempts	Yes
Evading Presentation Access Control Through Custom Posts	Yes
Cross Site Scripting Attempt	Yes
Violations Of Implemented White Lists	No
Double Encoded Characters	No
Unexpected Encoding Used	No
Blacklist Inspection For Common SQL Injection Values	Yes
Detect Abnormal Quantity Of Returned Records.	No
Null Byte Character In File Request	No file requests
Carriage Return Or Line Feed Character In File Request	No file requests
Detect Large Individual Files	No file requests
Detect Large Number Of File Uploads	No file requests
Irregular Use Of Application	Yes
Speed Of Application Use	Yes
Frequency Of Site Use	Yes
Frequency Of Feature Use	Yes
High Number Of Logouts Across The Site	Yes
High Number Of Logins Across The Site	Yes
High Number Of Same Transaction Across The Site	Yes

The Table 3.2 gives an overview for which detection points from the default OWASP list was implemented. These detection points are fairly generic and

can apply to most web applications. They detect a number of generic web application attacks, monitors trends in authentication and site usage. This is features nearly every web application has and is much the same across applications. One assumption which can be made is that every web application that uses OWASP AppSensor, either the reference implementation or their own, will use these detection points, since they are fairly universal and well documented by OWASP. However, as can be seen from the Table 3.2 some of the detection points did not apply for the simplebank web application. Because features like file upload or download is not used in that application, but others like the detection of multiple failed passwords would require a rewrite of the authentication method.

The unexpected quantity in the username and password detection points have been marked as partly in the table, the reason for this is that there is hard to have numbers on what is an unexpected quantity. Since the simplebank website is really insecure it does not have any restrictions on how short the password can be, and setting an upper limit is also hard. Off course people do not manually type in a 100 character password, but a password manager program might use a 100 characters or more. In the scenario the limits were set to a minimum of 2, which is far to low, and a maximum of a 100, which is reasonable. However the most useful thing to detect here is probably that an attacker is not trying ridiculous small passwords, which are smaller than the web application require, nor that the passwords are to big, hence making the HASH-algorithm taking a lot of CPU-power and in effect producing a denial of service attack.

Unexpected types of characters in the username and the password input means that an attacker for example sends a username, which is outside the ASCII range or other predetermined ranges of allowed characters. The same applies for passwords; if there are some characters that are not allowed, like carriage return line feed or some other special signs in the passwords. The problem for the simplebank is that it has no limitations either to the username or to the password. A detection point would therefore easily be generating false positives by detecting values that should be accepted by the simplebank application. When it comes to cookie detection, two points were omitted. Since the simplebank application does not have many cookies other than the session cookie, it is hard to tell if a user is deleting any other cookies or deleting the session cookie, which would happen if he closes his web browser.

Encoding and detecting both unknown encodings, double encoding or wrong encodings have not been implemented in this application mostly due to time constraints and difficulty of implementing it reliably. These were also omitted

from the OWASP AppSensor projects example applications.

One important point is that all of these detection points have been implemented manually in the application and is therefore subject to coding errors and bugs just like any other feature in a web application. However, this can also be true for any IDPS system or WAF. This is the reason why having multiple redundant security features often can be a good thing, because an error in one does not necessarily have to be an error in another.

3.3.5 Using the Detection Points

The detection points are using the OWASP AppSensor framework which handles the storage and possibly the actions of the detection points. In the simplebank application the detection points are being stored in memory by the reference implementation, this means that when the application restarts all the events that have been detected are deleted. In the OWASP AppSensor developers guide there is specific recommendation to make a better implementation of the storage so that the exceptions are stored in database or somewhere else persistent. However for this experiment the in memory storage is good enough and makes it easy to start the experiment with a clean slate if it needs to be repeated.

OWASP AppSensor also supports response actions which are actions the application can perform when certain thresholds are met from the detection points. For example if a user have tried 5 cross-site scripting attempts the user is banned from the application for a certain time period. Another example can be if the application detects an increase in the usage of a certain feature, for example a status update on a social website, the application can disable that feature until an administrator can figure out what is wrong. These response action where not used in this experiment since the experiment is about detection, not how the application can take actions against the attack. Therefore the simplebank AppSensor only logs what it detects and performs no other actions.

3.3.6 Execution of the Experiment

Since the purpose of the experiment is to see how AppSensor compares to some other defensive mechanisms in terms of web application vulnerabilities, the experiment will perform each of the attack in turn. Each attack will record were it was detected, if anywhere, and since the mechanisms are configured to

only detect, the attack will pass on through. When the attack has been carried out, the logs for each of the three defense mechanisms is checked. Where the attack was detected is written down and the next attack is executed.

Between each attack the environment will be reset, in practice meaning that the virtual machine will be rolled back to the state before the attack. This is to prevent the attacks from interfering with each other and at the same time keep the environment the same during each of the attacks. This is to make the results as isolated as possible and also making the experiment easier to be reproduced. The order of the attacks therefore becomes irrelevant and it is easy to change the order or perform new attacks from a steady state later on.

Chapter 4

Scenario and Experiment

This chapter will describe the scenario that is created to compare the three different methods,

The scenario used here is generic and the web application is also very simple and generic and it is made so that some simple attacks which are based on the OWASP top ten can be carried out against a website.

4.1 Business Case

The business case is a simple bank, which is called simplebank. The bank does all its customer interaction over a website and have it as the sole point of the business. The choice of a simple banking website was done by request from BEKK, but it is a simple website that represents a general website with some business value it needs to protect. The website have the following functionality which is essential for the bank business:

- Register new customers
- Login registered customers
- Make new accounts for customers
- Transfer money from a customer account to other accounts

These functions is what the banking website is able to do. It is very simple but still has enough functionality so the attacks can be carried out against it. In addition there is a way for the administrator account to look at all the

user accounts which are stored on the system. A more detailed description of the website is given in Subsection 4.1.1 below.

4.1.1 Description of the Bank Web Site

The bank web site features a login form, which lets a user log into his user account on the banking web site. When inside, the user can create new accounts for himself and create and look at existing bank accounts he currently has. Since this is a simple scenario, the user is able to insert money out of nowhere into his accounts, so he is able to get some initial funds. This would normally not be possible and would be considered a big flaw with the application. The user is also able to perform transactions from his accounts to other accounts, either owned by him or other customers in the bank. Screenshots of the simplebank application is included in Appendix B

The bank will be coded to be vulnerable, meaning that for example there will be no validation of the database input, it will go straight down to the database. For example the username forms field, it will be vulnerable to many injection-vulnerabilities since there will be no input sanitation. This is to make is easy to exploits the application and make the application rely on either third-party systems such as an IDPS, WAF or the built in AppSensors to prevent the attacks. In reality a web application would, hopefully, be designed much more secure by default, but for this scenario it is easier to have it be vulnerable and completely dependent on other systems for protection.

4.1.2 Implementation Details

The simplebank is implemented in the Java programming language, with the Java Enterprise Edition (Java EE) servlets, JSP and other technologies to make it a web application. Unlike many other Java web application, there are no frameworks used other than the basic Java EE features. This is because many frameworks, for example Object-Relational Mappers (ORM) often prevent SQL-injections, filter output from the application or have other security features that would prevent the web bank from being vulnerable. Since the whole point is to have a simple but vulnerable bank the usage of frameworks were omitted. The usage of the Java programming language was chosen since the OWASP AppSensor reference implementation is done in Java and made to be implemented in Java EE web applications. By programming the simplebank in Java the reference implementation of AppSensor can be

used and there is no need to make a standalone AppSensor for another language and/or frameworks.

The bank itself is badly coded; there are almost no tests since it is not critical that the bank works correctly. One of the points is that it should be poorly coded to be vulnerable. The functionality of the bank is mostly handled by Java Servlets, which handles the GET and POST requests from the client. These are implemented in the Servlet API by simply overriding the doGet and doPost methods. There are also some pure Java classes in the application. These pure Java classes are used for the database handling and some other utilities to avoid that the Servlet classes become too cluttered. When the doPost or doGet methods are finished running and want to return a result to the users request, the Java Server Pages technology is used as a template language. The JSP renders the HTML code that is delivered to the user.

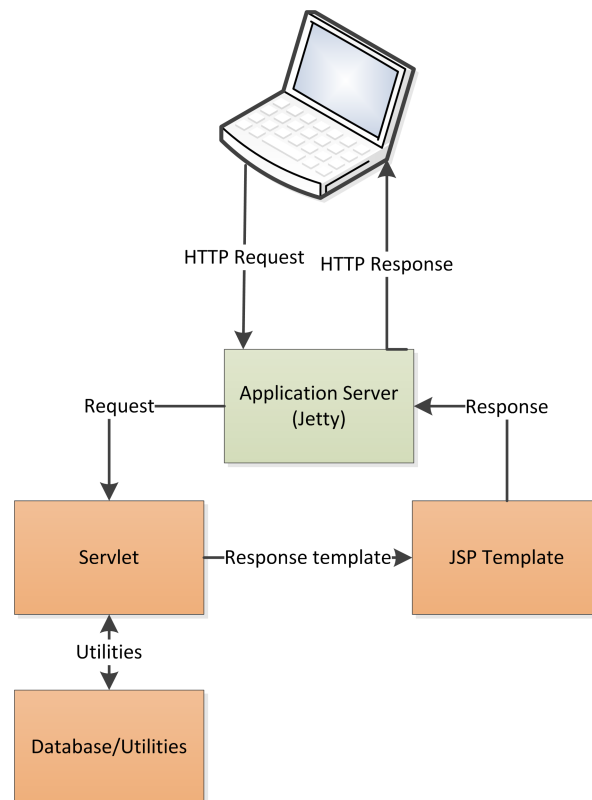


Figure 4.1: Handling of HTTP requests and responses

Figure 4.1 shows how the application works when a HTTP request comes in from a client and how a response is delivered back to the client. The application server, which in this case is Jetty [6], is colored in green and the

servlets, jsp and database/utilities are colored in orange since they contain the custom code written in the thesis. What happens in Figure 4.1 is that a request comes in from a browser. The request is first parsed by Jetty which calls the corresponding servlet. The servlet code then handles the request, and if needed perform the necessary database and or utilities functions required for that request. When the servlet is done with the request is handles it to a jsp template which renders the html which is delivered back to the client in the HTTP response. The jsp code is only responsible for rendering the HTTP template and all the data it requires are fetched and delivered to it by the servlet. While the jsp could perform much more logic, it is more clean to have most of the logic in the templates and only have the jsp render the templates, this is only done to make the code easier to read and less tangled.

4.1.3 Attacks

The OWASP top ten will be used as a basis for the attacks against the simple bank. This is done by request from BEKK. The OWASP top ten, which currently is from 2010, lists the top ten web based vulnerabilities. A short overview of the OWASP top ten will be given here with more detailed description of the attacks is presented in chapter 5.

The OWASP top ten project[48] is a concise, risk focused list of the top ten most critical web application security risks. The top ten list also describes each attack and how to defend against them. The top ten list is by no means a definitive list, it just scratches the surface of web application vulnerabilities and lists the ten most critical vulnerabilities only. However even tough there are only ten, it provide a good foundation to compare different ways of blocking the attacks. These ten are listed in Table 4.1 which is loosely based on the table from [48].

Table 4.1: The OWASP Top Ten

Attack	Description
Injection	Injection attacks occur when an application sends untrusted data to an interpreter. Common examples are SQL-injection, LDAP-injection, OS commands, XPath queries and others.
Continues on next page	

Attack	Description
Cross-Site Scripting (XSS)	XSS occurs when the web application takes untrusted data and sends it to a web browser without validation or escaping, resulting in the data being executed.
Broken Authentication and Session Management	The authentication and session management is not implemented correctly. Resulting in that attackers can compromise either passwords, keys, session tokens or other flaws.
Insecure Direct Object References	This happens when an object is not secured via access control checks. Allowing an attacker to get access to valuable information such as files, directories or other data.
Cross-Site Request Forgery (CSRF)	A CSRF attack uses a users session to send a forged HTTP request, riding on the logged in users session. This makes an attacker able to send requests on behalf of the user.
Security Misconfiguration	This point is about the configuration of everything from the application, frameworks, application server, web server, database server, operating system and so on.
Insecure Cryptographic Storage	The web application fails to use cryptographic storage of sensitive information, making it easy for an attacker to access or modify weakly protected data.
Failure to Restrict URL Access	Many web pages perform URL access rights before rendering protected links and button. However, the application need to perform similar access control each time a page is accessed to prevent an URL forging to access the page.
Insufficient Transport Layer Protection	The web application fails to authenticate, encrypt and protect the confidentiality and integrity of sensitive network traffic. Making it easy for an attacker to sniff the sensitive network traffic.
Continues on next page	

Attack	Description
Unvalidated Redirects and Forwards	Many web applications use frequently redirects and forward users to other pages and websites and use untrusted data to determine the destination pages. Without proper validation an attacker can redirect an user to a phishing or malware site.

As mentioned above, these are the attacks which will be used when comparing the different web application security products against each other. In addition *Business logic exploits* will be considered. These exploits are not of the same simple technical nature as for example XSS, which is easy to detect for a WAF, but it is a way of bypassing authentication, making fraudulent requests by sending a negative amount of money and so on. All these attacks are often targeted for a specific application and they exploit flaws in the business logic of that application. These issues comes in many forms and sizes as shown by Jeremiah Grossman from Whitehat Security in his whitepaper [40] and from the presentation by both Grossman and Ford [41]. The OWASP website contains more details for how to test for business logic attacks [13]. However some business logic exploits can also be some of the attacks from the OWASP top ten. For example bypass the authorization for a given request on a web site often would exploit how the application performs access controls and authentication. A real world example of a business logic attack from [40] is an online auction. If an online auction website prevented attackers from guessing the passwords by temporarily locking the user accounts after 5 or more tries in a given amount of time. Once the account was locked the user must wait 1 hour before the account opens again. An attacker could exploit this by placing a low bet on an item, and when a user overbid him, he could first make a slightly higher bid and then attempt to log in with that user 5 or more times, effectively locking the users account and prevent him from bidding anymore. This could be exploited to prevent many users from bidding and the attackers chances of winning the bid increased. One thing that makes this exploit hard to find is that there is no bug in the code itself, nor any seemingly dangerous traffic headed for the application.

Business logic exploits can therefore be quite difficult to catch because the system itself may be coded securely, with proper input sanitation, strong encryption of sensitive data and protect against many other common attacks but still have an exploit at the business logic level. Since business logic attacks are specific to that application there is no way to get a security appliance, like a WAF for example, and have it prevent the attacks without some serious

configuration and tuning for that application. While AppSensor can be argued to be able to detect such attacks one need to consider that the developers often have to know about such attacks in the first place to be able to detect and defend against them. This is often not the case, which is why business logic exploits is hard to detect. However if AppSensor is able to block and prevent some simple business level exploits, it might prevent the attacker from ever discovering the true vulnerabilities.

4.2 Network Configuration

The network configuration is a basic corporate network setup, with a firewall in front, an inline IDPS system, an inline WAF and an AppSensor-based web-application running with a web-server in front and application server and database behind the web-server. A diagram of the network setup is shown in Figure 4.2. In the network diagram the additional servers is the rest of the simplebank infrastructure. This might be email, directory or other types of servers needed by the bank. Since they are part of the attack surface they are present in the diagram, however since the main focus is web application security they are not included in detail but only as a cluster of different servers. Below the different devices in the network diagram will be discussed.

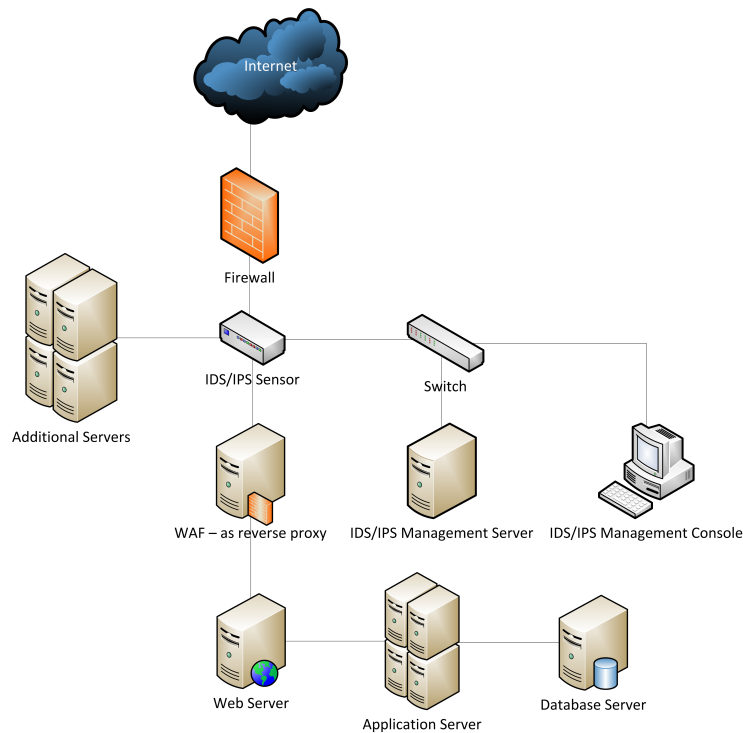


Figure 4.2: Network Diagram for the Scenario

4.2.1 Firewall

The firewall is assumed to be a basic firewall, which are able to block specific IP-addresses. It is also assumed that only three ports are exposed outside, port 80 for HTTP, port 443 for HTTPS and port 22 for SSH to do administration

of the server. All other ports are closed for inbound traffic. The firewall therefore makes for a central point in which all the traffic to the network have to pass through. Therefore in order to completely block an attacker by IP-address the attacker must be blocked in this firewall.

4.2.2 IDPS

The IDPS system is placed inline in the traffic flow to the network as shown in Figure 4.2, meaning that it will serve as the second central point which all traffic have to pass through. Because of this it receives the same power when blocking an IP-address as the firewall. This means that the IDPS will block an IP-address completely from the network if it is blocked in the IDP/IPS. In many networks one could have the IDPS and firewall be on the same hardware device, however for the sake of simplicity and to differentiate them they are separate units in this diagram.

The IDPS system has both a signature, a anomaly-based detection which detects network anomalies and a stateful-protocol analysis, as explained in Subsections 2.2.1, 2.2.2 and 2.2.3. Even though this would require a considerate amount of processing power to do all these things, it is assumed in this scenario that the IDPS system has ample resources and will not be a performance bottleneck in the network. For the level in the ISO OSI stack that the IDPS operates on, it is assumed that it does operate up to level 7. This means that the IDPS system is able to understand HTTP-traffic. This makes for an interesting comparison between the WAF, which only focuses on web application traffic and the IDPS which focuses on all network protocols including web application protocols. The huge benefit of the IDPS is that since it understands all the traffic, it can detect attacks which might indirectly attack the web application. One example of such an attack would be against the DNS-service, because the web application is useless without DNS.

4.2.3 Web Application Firewall

The web application firewall is also placed in an inline fashion, however it is only inline with the traffic to the web server. This means traffic that goes to other servers or workstations do not have to pass through the WAF. The WAF is assumed to be using a negative security model, which allows all the requests except the ones that are matched by the blacklist signatures. It is also assumed that the WAF does traffic normalization before handing the

traffic over to the web server. Even though the WAF is inline in traffic it has the same assumption about performance as the IDPS, meaning that it will not have any impact on performance.

The WAF only operates in level 7 of the ISO OSI model and is able to intercept TLS-traffic, meaning that the WAF have a copy of the web servers private and public keys. One thing to note here is that an IDPS system might also have these keys and be able to intercept TLS traffic, however it is assumed it cannot. This is done to make it simpler to distinguish between the WAF and IDPS systems because in reality there are manufacturers which make products which blurs these boundaries.

One interesting thing to note about the banking industry is that one of the requirements from the Payment Card Industry Data Security Standard (PCI-DSS) which touches organization running web applications. The requirement about security states that either the web application should have a vulnerability security assessment or the other option; a web application firewall should be used in front of the web application, specifically to prevent attacks such as SQL-injection [54].

4.2.4 Web and Application Server

The web server is a standard web server which does the delivery of web pages, it also handles caching and serving of static content for the web application, such as pictures, css and other static content. The web server acts as a reverse proxy in front of the application server which handles the application itself.

Behind the web server is the application server where the simplebank application resides. It communicates with a database server that handles the database for the web application. The separation between the application server and database server is done so that there is an external connection to the database which needs to be secured, as well as to make it easy to scale with multiple application servers while still have one consistent database. Requests are forwarded from the web server to the application server and, after being processed in the application server, are sent back to the web server before being handled back to the user. The main point here is that the web server is in front as a reverse proxy and handles static content and some page caching.

The data flows in the application are shown in a data flow diagram in Figure 4.3. The red dotted lines shows trust boundaries, these are hard to define, but in this diagram it shows several layers of trust. In an ideal setup each

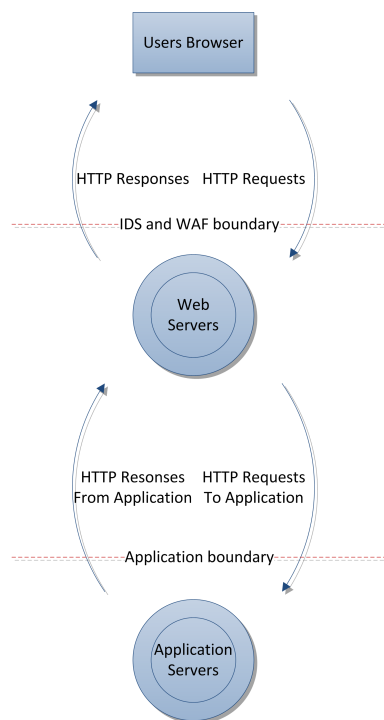


Figure 4.3: Data Flow Diagram for the Bank

system would, in addition to having the other layers of protection, not trust the other systems. For example should the web application system, in a perfect environment, not trust the data delivered from the WAF or other proxy servers. In this example however, the web application accepts all types of data and the sanitation happens further up in the diagram. This shows that when a HTTP requests comes in it is first handled to the web-server, when the data leaves the web-server and travels to the application server it is assumed to be sanitized, meaning that cross-site scripting, injection attacks and so on should be filtered. The same process is used with encoding attacks. This assumption could be dangerous, because if an attacker is able to circumvent the web-server, the application server could be compromised. As we can see there are many other calls in the diagram, and all these call could be tampered with or spoofed by an attacker. If the attacker is able to tamper with the data going to the database server the other layers of defense on top does not matter. There is no database layer in the dataflow diagram. This is because the database is hosted on the same server as the application and there is no external connection to a database server.

The web server will therefore first intercept a typical HTTP request coming

from a user and if the data is properly sanitized, it will be forwarded to the application server. The application server will determine what to do with the request, and will fetch data from the database. When the transaction is done the result is sent back to the application server which then renders a response and handles it back to the web server which in the end sends back the HTTP response to the user. Of course the application server might not need to query the database, but either way the application server handles the business logic and makes the response that is sent back to the user through the web server.

Appsensor detection points are built into the application, making it possible for the web application itself to detect and prevent attacks. Exactly how these detection points are built in will be discussed in the next chapter, many of the detection points from table 2.2 is used.

4.3 Snort as IDPS

Snort is the concrete IDPS system used in the experiment. The definition from the Snort website:

Snort® is an open source network intrusion prevention and detection system (IDPS) developed by Sourcefire. Combining the benefits of signature, protocol, and anomaly-based inspection, Snort is the most widely deployed IDPS technology worldwide. With millions of downloads and nearly 400,000 registered users, Snort has become the de facto standard for IPS.

According to the paper [50] Snort can be used as a network-based IDPS or a Host-based IDPS system. In the experiment Snort is being deployed as a Host-based IDPS and monitors all the traffic into that specific host. One of the big powers of Snort is its rule set, which is huge and updated regularly. The big disadvantage is that in order to get the latest rules, one has to become a subscribed member to the Sourcefire Vulnerability Research Team (VRT) to get the latest rules on new vulnerabilities as soon as they are discovered. Other users which are registered on the website but does not subscribe to the rules get them 30 days later. The free for all rule set is updated whenever Snort has a release. Therefore in this experiment, the rule set used is from about a month old.

Snort also support the use of preprocessors, these are code which are run to inspect the traffic before it is handled to the detection engine which goes

through the rule set which snort uses. There are many Snort preprocessors but for this thesis, the one named `http_inspect` is probably the most interesting one. This preprocessor will decode the buffer, find HTTP fields and normalize them. It works with both HTTP requests and responses. For the experiment the `http_inspect` preprocessor is setup with the apache profile since that is the webserver used in the experiment.

4.4 ModSecurity as WAF

ModSecurity is a web application firewall which works as an Apache module. Since it works as an Apache module it requires Apache to be used as a webserver. The project has its name from Apache also, since modules for Apache are generally prefixed with `mod` before the name, IE `mod_python` and therefore the web application firewall project was named ModSecurity.

According to the Web Application Security Consortiums Web Application Firewall Evaluation Criteria [33], ModSecurity is a software only, WAF which supports both reverse proxy setups as well as embedded setups. ModSecurity since it is an Apache module get the benefit of Apache decrypting TLS/SSL traffic and can therefore inspect traffic that have been encrypted with TLS/SSL. It also supports both a positive and a negative security model and this depends on how the user configures ModSecurity.

Just like Snort, ModSecurity have a specific way of dealing with its rule set and have a commercial offering. ModSecurity have an external OWASP project called Core Rule Set which provide a set of core rules for ModSecurity. In addition to these rules Trustwave Spiderlabs, which is the company behind ModSecurity, have their own commercial rules. From the ModSecurity FAQ, the differences between the rule set are explained:

The OWASP ModSecurity CRS security model is based on the concept of "generic attack detection" which means that it analyzes all HTTP transactional data looking for malicious payloads. While this technique does provide a base level of protection, there are still accuracy issues since the CRS does not correlate specific attack vector locations (such as URL and parameters) from publicly disclosed vulnerabilities. The ModSecurity Rules from Trustwave SpiderLabs focuses on specific attack vector locations, creating custom virtual patches for public vulnerabilities.

In the experiment, just like with Snort, the free core rule set will be used.

One of the biggest differences between ModSecurity and many other WAF-products is that ModSecurity is an Apache module, and therefore needs Apache in order to run. Many other WAF are standalone, either physical appliances or software which can be run standalone and does not need any other webserver. Another point is the different points of operations, and what types of security policies they will support.

4.5 AppSensor Implementation

To implement the AppSensor detection points in the application the reference implementation of OWASP AppSensor which is up on google code [4] was used. In addition the table in [32] which mentions the different detection points was used as a basis for which detection points where to be implemented. There are off course many other sensors and detection points that can and should be implemented in an application, however since this was the official table from OWASP, it was used as a basis for the implementation in the experiment. The table in [32] does only cover general web attacks, since it is not feasible to develop a table for every type of custom web application out there. Still there are some attacks which are left out in the table that could still be considered to be general web application attacks, such as Cross Site Request Forgery.

The way the detection points are implemented is by simply triggering a new AppSensor intrusion with an AppSensor exception as the argument. Some of these detection points are implemented in an aspect oriented manner, which in this experiment meant implementing them as filters for the different Servlets of the application. However some of the detection points, like for example those regarding authentication where implemented by mixing it in with the business logic. Often it is desirable to implement many of the detection points aspect oriented since that simplifies the maintainability of the detection points. The detection points that were not implemented as part of the business logic were implemented as filters in the Java environment. How filters work is explained in Figure 4.4. In the figure a request that is incoming is put through a series of filters, depending on the destination of the request. These filters can then perform for example detection of cross site scripting, or SQL-injection in all the forms destined for the application itself. This way all the detection points regarding cross site scripting or SQL-injection is handled in one place instead of being all over the rest of the business logic.

In the experiment there really was two places where the detection points were

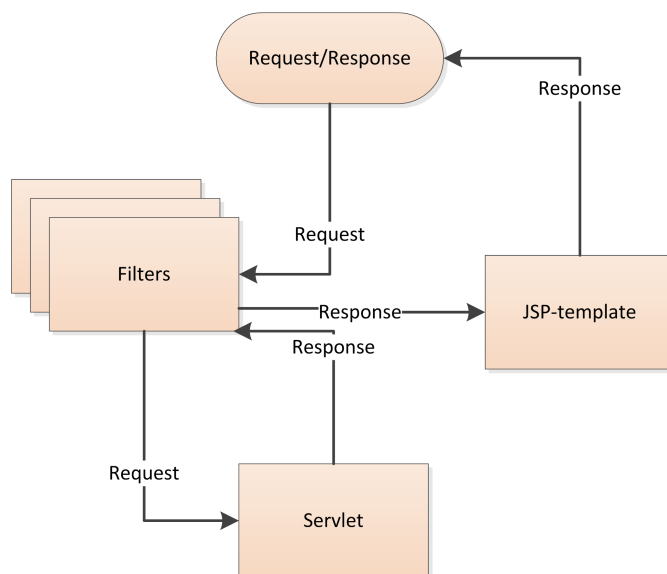


Figure 4.4: Java EE Requests and responses with Filters

implemented, it was some in filters which scanned all traffic before handing the control to the servlet, and there were some detection points embedded with the business logic. There are many other ways that detection points could have been implemented but one trend is that some points are only needed at a few places in the application, such as the detection point which alert of a large money transfer, or multiple usernames tried. Other need to be done for every input field, every cookie, or every request to check the headers, these detection points are much easier to implement in a filter or other way which intercepts every request to the web application, much the same way as a WAF does. Because of these different requirements the detection points were implemented in a different manner.

Implementing the detection points, at least most of those supplied with OWASP AppSensor document, is not a difficult job. Many of these points however, may need some functionality, which not all web applications have. For example a way of getting the user agent and IP-address used in a session. In those cases some of the detection points would be more difficult to implement because the need to implemented extra functionality need to be performed. This might not be a huge problem, but depending on the type of application and frameworks used, it could prove time consuming. There is also quite easy to simple not implement certain detection points which proves challenging, especially if they are not really critical detection points to have.

One nice side effect of implementing detection points in an application, is

that whenever a detection point is implemented, the developer have to be concerned with how to make the potential vulnerability safe. For example, if a detection point which checks incoming traffic for cross site scripting attempts, the developer also need to make sure to sanitize and prevent the cross site scripting attack from happening as well as detecting it. This point is not true with all the detection points. For example the detection point which detects a change of user agent in the middle of a session with the site does not necessarily mean an attack or that the site is vulnerable to anything, it is merely some suspicious behavior which in conjunction with other suspicious events could be an attacker trying to attack the application.

There is a drawback by having to implement these detection points, and that is the interference from other frameworks. The experiment did not use any framework and very few libraries, this was both to keep the application simple, but also to prevent these frameworks from filtering and sanitizing the input. Many web application frameworks already does a lot of sanitizing of input and other security features, this might make it hard to have the detection points actually getting their hands on the raw input to the web application since all that might be exposed to the programmer from these frameworks are the sanitized data which are already safe and therefore potential attack on the web application have already been averted and AppSensor never got any chance of detecting the attack. The same problem can happen if a WAF or IDPS is used which manipulates the traffic before sending it to the web application.

4.6 Drawbacks with the Scenario

There are several drawbacks with this scenario with regards to comparing IDPS, WAF and AppSensor. One of the biggest drawbacks is that the scenario will only have a few attacks, although these are some very common attacks, the scenario is not big enough to consider many other types of attacks, such as LDAP-injection, or tampering with data going into a Enterprise Service Bus and so on. The bank itself is also very basic and a realistic bank would contain much more functionality and therefore a bigger attack surface.

When it comes to the network, it will be simulated using a virtual machine and a host-based IDPS and host-based WAF and AppSensor implemented into the application itself. The WAF will be a reverse proxy which will be on the same host. The web application will therefore run on the same host but will use the WAF as a reverse proxy. This makes the network not a real

network, which would probably have a network-based IDPS and the WAF could be implemented as a reverse proxy which could be standalone from the web application server. The vulnerability that could be used by an attacker in a standalone WAF is that an attacker could attack the connection going from the WAF to the web application server. If the WAF filters the traffic to prevent an attack, the attacker could tamper with the data going from the WAF to the web application to enable the attack again. These kind of attacks is probably infeasible since all the different network appliances is running on the same machine.

The last drawback is that since the scenario requires an actual IDPS, WAF and AppSensor implementation, there can be errors in the products tested. As mentioned earlier, many IDPS can understand HTTP-protocol and therefore do much of the work which a WAF can do, and there are also different functionality in different IDPS and WAF products. This makes it difficult to find a clear distinction between the products and even though the intention is to test a general IDPS and WAF, there is no such thing as a general IDPS and WAF. The testing must therefore use an actual product which satisfies the criteria for an IDPS and a WAF. Another difficulty is the cost of the product, many products are huge and expensive and are made to be implemented in a huge network. Often these are integrated packages with both a network-based IDPS, a wireless IDPS and a SIEM solution that integrates all the information gathered from the different sensors. Since the scenario is quite simple it will be difficult to test these solutions. Another problem is the cost in money, these big solutions are quite expensive and this thesis project does not have the money to buy them just for this testing.

4.7 Scenario Summary

The scenario is a simple bank, which only have a web site as their customer interface. The web site is therefore critical in the operation of the bank itself. The scenario is a fairly common setup of a web application with both a IDPS and WAF inline in the network traffic. The web application has one proxy, the webserver with the WAF, which handles some of the load and some extra security. Behind all the layers of the security is the application server and database. This is where the crown jewels reside and is therefore hidden behind all the other layers of defense. The web application for the bank is also written to be insecure, so that it is vulnerable to all the different attacks presented in chapter 5.

Chapter 5

Attacks

This chapter goes through the different attacks which will be performed on the simplebank website. This chapter will show how the attacks are performed and contain all the details for the various attacks. It will explain the attacks in great detail so others can easily carry them out.

5.1 Injection Attacks

This is the injection attack from OWASP top ten guide. The experiment will only take the simplest one and perform them, because they are some of the most used ones according to OWASP top ten guide. They are also easy to perform by hackers. Some of the other types of injection attacks, like for example command injection or LDAP-injection will not be used here, both because they are a little less common but also because the application does not communicate with any LDAP-servers nor uses any OS-commands.

5.1.1 SQL-Injection

The injection attack that will be performed on the simplebank is a SQL-injection. This attack injects SQL code that can manipulate queries against the database. This attack is successful since the simplebank website passes the raw input data directly to the SQL-database. This means that something that is completely innocent in a HTML-context, is quite harmful and will be interpreted by the SQL-database.

Simplebank have performed their authentication of users by simply taking the input from the username and password field and sending it directly to a SQL query shown in the listing 5.1.

```
SELECT username , password
FROM users
WHERE username = ''' + username + '''
AND password = ''' + password + ''';
```

Listing 5.1: Shows the SQL-query executed when user logs in

The username and password being used in the query is the ones which are in the input fields from the simplebank web application. They are not filtered and therefore the database server will execute the SQL-commands contained in them. An attacker can exploit this by for example putting `' or '1' = '1` into the password field. This will cause the AND clause in the SQL query to always become true and the attacker can authenticate as any user without knowing their password.

This is the statement used in the experiment to perform a SQL-injection. It is injected in the password field in the login form for in the simplebank application. A valid username is entered in the username field and `' or '1' = '1` in entered in the password field.

5.2 Cross Site Scripting

Cross Site Scripting, often shortened XSS, is a very typical attack on web applications. Cross Site Scripting is about tricking a web application to let a users browser run javascript code inserted by the attacker [25]. This can be performed either reflected or stored [16]. A stored attack is stored in the web applications database or other storage and then served up to the user. A reflected attack cross site scripting is reflected of the target website, but runs in the websites context. The reflected attack often needs to send the victim a URL which is carefully crafted to exploit the cross site scripting vulnerability.

Cross site scripting is one the most popular attacks on the internet, both for attackers to prove that there is a cross site scripting vulnerability with a cross site scripting payload which is not harmful such as the one above. There are even several sites which collects cross site scripting attacks, for example this subreddit at Reddit [19] and another website which points out cross site scripting attacks [27]. These websites shows the popularity of the cross site scripting vulnerability.

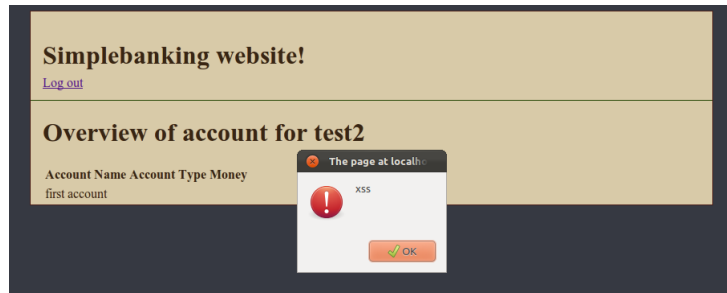


Figure 5.1: Showing Cross Site Scripting on the simplebank

The cross site scripting attack is performed through an input field in the web application there will be inserted a *script* tag which will contain Javascript code. This code will run when a user later requests the content for that page. The code used for the experiment is shown in listing 5.2.

```
<script>alert("XSS")</script>
```

Listing 5.2: Shows simple demonstration of Cross Site Scripting

This attack is carried out on the page where a user can create an account. The input field that lets a user create an account and assign a name to that account will be exploited and the above code is injected as the account name. This field allow arbitrary characters to be used as input and does not filter them either on the way in or the way out. A demonstration of the attack is shown in Figure 5.1. This attack can be used against the simplebank to for example letting an attacker utilize Javascript to read out the contents of the cookie being used in the session. By getting his hands on the session cookie an attacker can impersonate the user and perform a fraud by collecting money from the users account. The attack described above is the one being performed in the experiment.

5.3 Broken Authentication and Session Management

Broken authentication and session management is quite a broad area of vulnerability from the OWASP top ten [15]. It covers anything from the initial login, to the handling of a user session to how the username and passwords are stored.

5.3.1 Authentication

As seen in Section 5.1.1 an SQL-injection can be used to bypass the password check of the simplebank website, thus breaking the authentication scheme. The authentication is also weakly implemented and will allow as short a password from a user as the user wants, meaning that a password with one character is perfectly acceptable for the simplebank.

The authentication is also just one factor, while financial institutions in America is advised to use multifactor authentication [34]. The simplebank wanted it to be easy for its customers and therefore have single factor authentication. This makes it easier for an attacker to brute force the authentication scheme for the simplebank.

Another error the simplebank has made is that it is storing the passwords in plain text. This means that if an attacker is able to compromise the database, the attacker would get all the plain text passwords for the users at the simplebank website and can easily impersonate any of them.

Since there already is a way to bypass the authentication of the simplebank with the SQL-injection. The attack that is performed from this category from OWASP top ten is within the realm of session management and described below.

5.3.2 Session Management

The simplebank website uses a session cookie for tracking an authenticated users session across the website. The cookie is simple and has the username that is logged in as its value. This means that there are several ways an attacker can compromise this. One method is to change the value of the cookie to another username. This would make it trivially to impersonate another user by simply changing the session cookie value to be other users username. Another approach would be to forge a false session cookie, by simply creating a cookie with the correct name and have another users username as value. The attacker could then impersonate any user whose username was known, this is shown in Figure 5.2. This is the attack performed in the experiment.

In the experiment a cookie which is shown in Figure 5.2 was created and allowed an attacker to bypass the authentication mechanism of the simplebank website. This was performed by creating the cookie and thereafter access the accountoverview page which requires a valid session. These steps are the ones performed in the experiment.

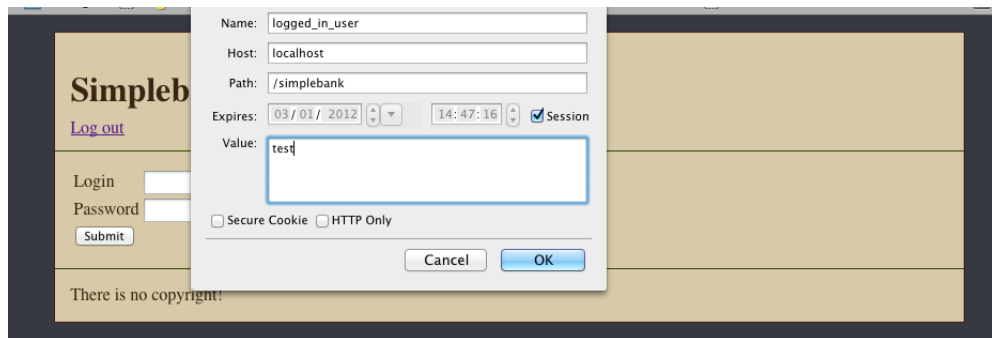


Figure 5.2: Creating a new session cookie

In addition even if the session cookie did not contain the usernames and would be infeasible to forge, it is still sent over normal HTTP and therefore not encrypted while in transit. This means that if an attacker could sniff and get the cookie he could use that cookie to impersonate that exact user. This works even if a correct implementation by session cookies is done because the cookie needs to be protected in transit.

5.4 Insecure Direct Object References

An insecure direct object reference is when the web application allows direct access to an object, which should normally be inaccessible for that user. This types of attacks can be of varying degree of impact on the business and really depends on which type of resource is accessed. The types of resources can be anything from files, to directories or other data. The simplebank have very few static files, but they have a readme file. This readme file contains some sensitive information about the setup of the simplebank website and should therefore not be exposed to an attacker. Unfortunately the simplebank only have access control for the get-requests to the pages that have servlets, since the README file is served directly by the application server it does not have any servlet and therefore lacks access control.

One of the attacks which is carried out is just directly downloading this readme file without have performed any authentication. When the attack is performed all the cookies are purged from the browser and the browser is restarted so that all session cookies are flushed out. Then the url where the readme file is accessed directly and the readme opens in the browser.

Another type of direct object reference can be access control errors. In the

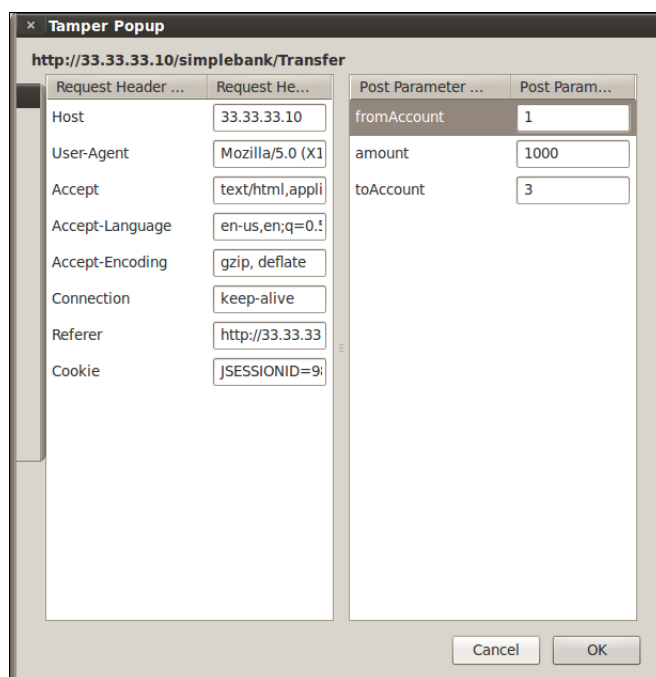


Figure 5.3: Tamperdata tampering with the transfer money request

simplebank there is a page which allows a user to transfer money from one of his account to another account in the system. However if the user submits another account instead of his own, the simplebank fails to check if that account belongs to the user and therefore happily performs the transaction for him. This means that a user can take money from another account and insert to his own. This attack works since the web application does not check that the data from the user is valid, it only assumes that since it sends our valid data, valid data will be returned.

The way this attack is carried out is by using the tamperdata plugin for firefox [24] and intercept the request going from the client to the server. The id of the bank accounts is changed to something that fits the attack and allows the data to be sent. This is shown in Figure 5.3. When the attack is carried out two users are created in the simplebank website, one with username test and another with username pentest. Both users create one account each, where the account ids become 1 for the pentest user and 2 for the test user. The pentest user then makes a request which he uses tamperdata with so that he transfers money from the account with id 2 to his own account which has id 1. This should not be allowed but since the simplebank does not check the data, which is sent back the user pentest is able to make the transfer.

5.5 Cross-site Request Forgery

Cross-site request forgery[42] happens when a user visit a malicious page which uses the users already established session with the simplebank website to send a post-request to the simplebank website, riding on the users session behind. This attack lets an attacker perform transactions behind the users back to his own account.

To perform a simple cross-site request forgery attack against the simplebank website, a simple form was created on another website, shown in Listing 5.3.

```
<form action ="http://10.0.2.2:8080/simplebank/Transfer"
name="transferMoney" method="POST">
  <input name="fromAccount" type="hidden" value="1">
  <input name="amount" type="hidden" value="100">
  <input name="toAccount" type="hidden" value="8">
</form>

<script type="text/javascript">
  window.onload=document.transferMoney.submit();
</script>
```

Listing 5.3: Cross Site Request Forgery site

The IP-address is the address to where the simplebank website is running and the path is to where the simplebank accepts the transfer money form. This form gets submitted when a user visits the site and transfer 100 money from account 1 to account 8, assuming that account 1 is the victim and account 8 is the attackers account. The code above would therefore be on a malicious webpage and when a user who is already logged in to the simplebank via another browser tab visits the malicious page, the form will be submitted and the users browser would send the session cookie along with it, thereby transferring money to the attacker.

Since cross-site request forgery is a general type of attack it can be used to send any valid form from a malicious page to the vulnerable page. For the simplebank the most obvious thing to do is the form shown above, which transfers money from one account to another. Another attack could be to submit another form, such as the register user form or the create account form. The register user form can be used to create a new user and the create account form can be used to create another account. None of these forms would do much harm to the simplebank, other than that customers might be upset if they suddenly get accounts they did not create themselves.

5.6 Security Misconfiguration

Security Misconfiguration is a broad point on the OWASP top ten, it spans from web server , application server to database configuration. In addition, it also spans the usage of frameworks and libraries. This makes it a very broad point which goes beyond the application itself. It is very important that every part of the web application stack is secured, because nothing is more secure than the weakest link. In the experiment the focus is on the web application itself and not so much upon the whole stack needed. This is one of the places where defense systems which are not only protecting the application, but all the things around like for example a network-based IDPS system or a WAF. These systems could be able to detect a cross site scripting attack on one of the underlying libraries that AppSensor have no way of reaching into to detect what is going on.

However since this is such a broad point and also often outside the web application itself, these types of vulnerabilities were omitted from the experiment. The point is included here because it is quite important, but it falls outside the scope of the experiment in the thesis, since there is little AppSensor can do outside the application.

5.7 Insecure Cryptographic Storage

Sensitive data is not always encrypted or stored securely. This can be anything from simple password hashes to other sensitive information that should be encrypted or stored in a hashed form. According to OWASP top ten the most typical mistakes made here are unsafe key generation and storage, not rotating keys and use a weak algorithm are common mistakes. Using a weak hash function or not salting is also common mistakes.

In the simplebank application there are two sensitive information fields stored in the database about a user. The first thing is the password, which is stored in plain text, used for login. The second field is the "something secret" field, this is to illustrate a piece of sensitive information, like a social security number, account number and so on which should be stored encrypted but is not. Both these fields are stored as plain text meaning that if an attacker can compromise the database, the attacker then has free access to this sensitive information for all the users. Since these types of attacks require an attacker to compromise the database, they are relative hard to execute, but if the

site is vulnerable to a SQL-injection, the attacker might be able to get the sensitive data.

This vulnerability itself is often not an exploit in itself, but often used in conjunction with other vulnerabilities for an attacker to get access to the data. It is therefore not tested in the scenario since there is no single attack that can be performed, but rather a consequence of an attack.

5.8 Failure to Restrict URL Access

By not properly restricting URL access an attacker could get access to an URL he is not authorized to access. This could happen if the URL contains no access control or wrong access control. For example wrong access control could be a page that only administrators of the application should be able to access suddenly is accessible to regular users. The severity of this type of attack is dependent upon the type of website the attacker is able to access. This can be anything from severely damaging the business goals to an embarrassing mistake that is no big deal.

The simplebank application have added access control to all their GET-requests, but they have forgotten to add access control to their POST-requests. This makes it easy for an attacker to post a form to any url on the simplebank website even though the user does not have access to it or is anonymous. This makes it possible for an attacker to perform a transaction without being logged in as the user where the money is transfered from; they only need to know their account number.

Since the simplebank does not have access control for their POST-request, a simple command with CURL can perform a POST-request which can transfer money from one account to another on the simplebank website. Listing 5.4 demonstrates the CURL command that will transfer 1000 money from account 1 to account 6. This is the attack performed in the experiment and the curl command is used.

```
curl -d 'fromAccount=6' -d 'amount=1000 '
-d 'toAccount=1' localhost:8080/simplebank/Transfer
```

Listing 5.4: Curl command for a POST-request

5.9 Insufficient Transport Layer Security

Insufficient transport layer security generally means that web sites do not use protection on the transport layer. Typically this means that a web site either do not use SSL/TLS or have improper handling of SSL/TLS. Improper handling would typically be a site which has only part of it protected by SSL and letting sensitive data such as session cookies or other information pass in the clear. The main difficulty for an attacker is to be able to sniff the traffic going to the website. To do this the attacker need to find a point between a client and the server in which the attacker can insert himself as a man in the middle. By placing himself as a man in the middle the attacker is able to sniff the sensitive traffic and can for example steal a session cookie to impersonate user.

Another often done mistake is that the connection to back end systems is not encrypted. So if the web application talks to a separate SQL-server it may be in the clear. If an attacker is able to sniff that traffic the attacker can manipulate the queries going to and from the database and thereby performing many types of mischief. In the case of the simplebank the attacker could steal money from a user or make a transaction go to his account instead of another valid users account, thereby performing a fraud.

However, since the simplebank website is only being a test scenario. For this setup there is transport layer security. This is acknowledged as a known issue but will not be considered when testing the different protection systems. The reason for not having transport layer security is that the testing will be further complicated. In some cases it can be very difficult to find out what is happening when the traffic is protected by an SSL/TLS connection. Therefore will this kind of vulnerabilities no be considered in the comparison and will not be considered an attack in the testing either.

5.10 Unvalidated Redirects and Forwards

This vulnerability lets an attacker redirect or forward a user to a malicious site of the attackers choosing. By doing this the attacker can infect the user with malware or perform a phishing attack. In terms of business risks this makes it hard for users to trust the website if they experience a phishing attack or get infected with malware by visiting a website they consider to be safe. The simplebank website makes use of both forwards and redirect none which are validated. The easiest to spot is the login page. The login page

can take one GET-parameter which tells the server where to redirect the user to after a successful login. An attacker could manipulate the GET-parameter and redirect the user to any site he wants. To get the user to click the URL could be done by sending the user the URL in an email, which looks legit since it points to the simplebank website but in truth it redirects to the attackers website which infects the user before it redirects back to the simplebank website.

To demonstrate this attack on the simplebank website, the following URL will redirect a user to the website *http://evilhacker.com* after they have authenticated with the simplebank website. The attack is carried out in the experiment by using that URL and perform the correct authentication and see if the browser is redirected to *http://evilhacker.com*.

```
http://localhost:8080/simplebank/Login?next=http://evilhacker.com
```

Listing 5.5: Unvalidated Forward URL

In the simplebank, the next parameter is meant to redirect the user back to the site they wanted to visit which needed authentication as shown in listing 5.5. For example, the user gets a direct link to his account overview page, the user clicks the link but in order to get an account overview the user need to authenticate first. The simplebank website then stores the link to the account overview page in the next parameter in the GET-parameter and uses it without validating that it only redirects to the internal simplebank page and is not able to redirect to an external page.

Chapter 6

Results

The chapter presents the results from the experiment. The attacks were all performed as described in chapter 5 and after each attack the logs for each defense mechanism was checked to see if it could detect the attack. Results from the attacks are found in the different sections below, with some discussion about both the attack and the performance of the defense mechanism.

6.1 Cross Site Scripting

Cross Site Scripting is one quite common attack which many webpages are vulnerable to. It is also quite easy to detect the common places where such an attack could happen, however due to web applications often being quite complex and having many ways of getting input, there are places where it is really hard to detect a Cross Site Scripting attack. For this experiment however a simple Cross Site Scripting attack was performed on an input field in the web application.

Snort Detection

Snort did not detect the Cross Site Scripting attack; even though Snort comes with a few rules that should be able to detect it. However even when a reflected Cross Site Scripting pattern in the URL was tried Snort did not detect that either.

ModSecurity Detection

ModSecurity is able to detect the Cross Site Scripting attack with no problems. It detects the attack both in the incoming request that have the payload, and in the outgoing requests which actually presents the script to the users browser so that it is executed. In the log ModSecurity have also logged the whole request and responses and also the IP-address from where the attack originated from.

AppSensor Detection

OWASP AppSensor contains a detection point in the application that is able to detect the incoming Cross Site Scripting attack. However right now the detection point only scans the incoming input fields in an incoming requests, and does not detect if a Cross Site Scripting attack could be performed elsewhere

6.2 SQL-Injection

The SQL-injection performed is also quite simple. It is one of the examples from wikipedias article on SQL-injection [26]. The SQL-injection is a ' or '1'='1 input string in the password field on the login form. This allows an attacker to log in as a user with only the username since the password check always evaluate to true in the password check because of the injected SQL statement. The reason this simple SQL-injection is used is to make it easy for the defense mechanisms to detect it. SQL-injections, just like Cross Site Scripting, can be used everywhere were input is used directly in a SQL-statement.

Snort Detection

Just as with the Cross Site Scripting attack, there is rules in Snort which mentions detection of SQL-injections, but it still does not detect this injection.

ModSecurity Detection

ModSecurity performs much like the case with the Cross Site Scripting attack. It detects the SQL-injection in the incoming requests and logs the whole request for further analysis. In addition it is able to check that both the Cross Site Scripting attack and the SQL-injection attack comes from the same user and therefore reports them both and gives a more serious combined warning than the individual warnings it produced for each one individually.

AppSensor

OWASP AppSensor has a detection point in its recommendations [32] which is to detect for SQL-injections. However the reference implementation of AppSensor does use a weak SQL-injection detection, which is able to find SQL-injections such as `'' SELECT * FROM users;''` but not the simple one used in the attack. Because of this the AppSensor detection point is not able to detect the SQL-injection, even though the detection point is scanning the string containing the SQL-injection, its rules does not detect it. This is a weakness in the reference implementation of OWASP AppSensor and not a weakness in the idea nor the concept of detecting SQL-injections in the input.

6.3 Broken Authentication and Session Management

Broken Authentication and Session Management can be many things. In this experiment the attack carried out was to manually create a session cookie. It allows an attacker to establish a session with the website without going through the usual authentication mechanisms. Since the session cookies does not contain any cryptographic nouns it can be created in order to establish a session with the website. The correct way would be to have a cryptographic noun and store all the information server-side and only have the cookie use the noun to identify itself, like the Java EE built in session cookie.

Snort Detection

Snort does not detect that the client has manually added a cookie. The attacker is able to establish a session without no problems. Also if a cookie is

tampered with Snort does not detect it.

ModSecurity Detection

ModSecurity does not detect if the client has created a new cookie, nor if an existing cookie is being tampered with.

AppSensor Detection

AppSensor does detect that the client for session management has manually added a cookie. It is able to detect if there are other cookies for that domain which is not used by that application, such as a PHP-session cookie set by another application will be reported by AppSensor. However the check to see if an attacker is tampering with the session cookie, the Java EE session cookie was used. It was used because a reliable way to detect this is needed, and the login cookie is not reliable since it is trivial to tamper with.

6.4 Insecure Direct Object Reference

Insecure Direct Object Reference can be exploited when web pages have static content which should be access control restricted, but it is simply served anonymously to everybody by the web server. The hard thing about these kinds of exploits is that no technology can know if a resource should be access restricted or not, since it depends on the context. Therefore there is no way for a third party device to know which resources should be put under access control and which should not. Because of this, both ModSecurity and Snort failed to detect the direct object reference. AppSensor on the other hand, can be made to detect such errors.

The second direct object reference were the attacker performed a tampering of the data in the transfer request had the same outcome. ModSecurity and Snort were unable to detect the attack but AppSensor were able to check the incoming data and check if it was a violation of the access controls or not. This is because the account ids only make sense for AppSensor and it is able to determine which account ids are allowed to be transferred from by the user performing the transaction and which ones are illegal. For ModSecurity and Snort all these ids are legal traffic and therefore not detected.

AppSensor Detection

Since the AppSensor detection points are built into the application, they can have the knowledge of the access control mechanisms in the application. An AppSensor detection points was built in to detect if a user tries to access the README.txt file on the web server. However one difficulty with using only AppSensor for detection is that the developers need to know beforehand which files should be restricted or not, and not make any mistakes, meaning forget some of them when implementing the access controls and associated detection points.

The other attack was also detected by AppSensor, since it is able to verify the results back from the client and check if there have been any tampering with the request. This is one of the strengths of AppSensor since it is able to detect access control errors that are specific to the application context.

6.5 Cross Site Request Forgery

This attack is quite common according to OWASP top ten [48], and the way it was performed on the simplebank was by having a third party website make the visitor submit a post-request to the simplebank website. This attack is hard to defend against without doing some modification of the simplebank website, namely sharing a secret between the website and the users browser. This is typically done by embedding a cryptographic nonce¹ into the forms on a webpage that is submitted to the client. When the form returns the server can check to see that the nonce is the same one as it sent out.

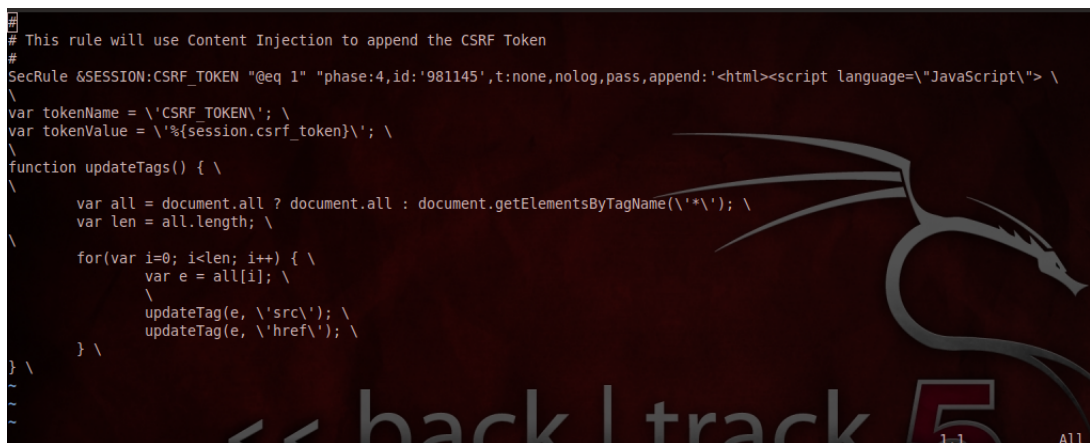
Snort Detection

Snort does not embed any cryptographic nonces to the forms which are outgoing, hence the only way it can detect a cross site request forgery attack is by looking at the referer header in the HTTP-traffic. But as shown in [42] this is not a reliable method for detecting cross site request forgery attacks. Because of this Snort did not find the cross site request forgery attack. Snort was therefore unable to detect the cross site request forgery attack.

¹The NIST SP-800-90 contains a definition for a Nonce [29]

ModSecurity Detection

ModSecurity with the optional rules enabled is able to detect the cross site request forgery attack. The optional rules are not enabled by default but can easily be enabled. These rules inject a cross site request forgery (CRSF) token or nonce to be precise into every form on the website. Part of the rule which does this is shown in Figure 6.1. The rule embed javascript code to the webpage that inserts the nonces into each form. When the form is submitted back through ModSecurity, it check to see both if the nonce are included, and if they match with the same nonce that where sent out. This made ModSecurity detect the cross site request forgery attack and reported that the nonces were missing from the form submission. One thing to note here is that this method does not work with asynchronous Javascript and XML which many modern websites use.

The image shows a screenshot of a code editor with a dark background. The code is a ModSecurity rule configuration. It starts with a comment: "# This rule will use Content Injection to append the CSRF Token". Below that is the rule definition: "SecRule &SESSION:CSRF_TOKEN "@eq 1" "phase:4,id:'981145',t:none,nolog,pass,append:'<html><script language=\"JavaScript\"> \\". The rule body contains JavaScript code: "var tokenName = \"CSRF_TOKEN\"; \\", "var tokenValue = \"%{session.csrf_token}\"; \\", and a function "updateTags()" that iterates over all elements in the document and updates their "src" and "href" attributes to include the CSRF token. The background of the code editor features a stylized dragon logo and the text "back | track 5" and "1,1 ALL".

```
# This rule will use Content Injection to append the CSRF Token
#
SecRule &SESSION:CSRF_TOKEN "@eq 1" "phase:4,id:'981145',t:none,nolog,pass,append:'<html><script language=\"JavaScript\"> \
\
var tokenName = \"CSRF_TOKEN\"; \
var tokenValue = \"%{session.csrf_token}\"; \
function updateTags() { \
\
    var all = document.all ? document.all : document.getElementsByTagName('*'); \
    var len = all.length; \
\
    for(var i=0; i<len; i++) { \
        var e = all[i]; \
        \
        updateTag(e, \"src\"); \
        updateTag(e, \"href\"); \
    } \
\
}
```

Figure 6.1: ModSecurity Rule with Javascript code it embed

AppSensor Detection

AppSensor currently has no way of detection cross site request forgery attacks. In order for AppSensor to detect it a proper cross site request forgery protection has to be built into the application which attaches a cryptographic nonce to each form and requires the POST-request to have that token upon submission. A detector can then be built in which checks if that nonce is missing or wrong and therefore detects the cross site scripting attack. This could have easily been implemented with AppSensor, but since the experiment focused on using the OWASP detection points, there was no detection point which would detect potential cross site request forgeries.

6.6 Failure to Restrict URL Access

Neither Snort nor ModSecurity is able to check if an attacker has bypassed the authorization or not. This is because they somehow have to have the knowledge both of how authorization is done in the application, but also which pages and resources should be protected. AppSensor detects this since it is inside the application and can know about which pages should be secured or not. The AppSensor guide has 4 detection points which cover access controls. By implementing these detection points AppSensor were able to detect if a user is accessing an URL that is restricted from that user. In the case of the simplebank, every GET-request which requires authentication is automatically redirect to the login page, but if a user tries to do a POST-request to the same page it will be detected and an AppSensor event will be generated. AppSensor therefore detected the attack.

6.7 Unvalidated Redirects and Forwards

In the simplebank web application this vulnerability is presented in an attribute that can be used to the login method. When a user visits a page that requires login he is redirected to the login page with a GET-parameter. It tells which site the user should be taken to after the login process is done. However since this is a GET-parameter, an attacker can change it to for example <http://evilhacker.com> that is a malicious web page. This domain was used to test the vulnerability in the experiment.

Snort Detection

Snort is unable to detect if an attacker is performing an unvalidated redirect or forward. This is because Snort would need the knowledge to know what are legal redirects and forwards and what are malicious.

ModSecurity Detection

ModSecurity is not able to detect the malicious redirect or forward. However in a recent edition of the Core Rule Set from OWASP that is used with ModSecurity, there is a feature which checks links against Google Safe Browsing API [5]. This feature could be used to check not only unvalidated redirects

and forwards but also all the links on a web site to check if they link to a known malicious site. Since this is a new feature it was not included in the experiment, but it could prove useful for these types of attacks. The feature could also be used to prevent malicious redirects.

AppSensor Detection

AppSensor was able to detect the unvalidated redirects and forwards attack. AppSensor had one detection point that checked the incoming argument to the next parameter and had an alert if it was not a subdirectory of the same domain.

6.8 Testing Other IDS Systems

Because of the bad Snort results, despite the fact that Snort did contain rules, which should have been able to detect the web application attacks, two other open source IDS systems were tested. OSSEC was one of them and it is an open source host-based intrusion detection system (hids) [12] developed by Trend Micro. Since OSSEC is a host-based IDS it contains some functionality that is not present in Snort, such as rootkit detection, integrated log analysis, Windows registry checking, file integrity checking and more. In addition to this it is able to monitor network traffic to that host, and just like Snort, it contains some rules for traffic analysis. It is therefore able to detect very simple web application attacks against the host it is protecting. When it was tested to see if it detected anything at all it was able to detect some simple cross site scripting patterns in the URL, but not in the body of the HTTP response.

The other IDS system that was tested was Suricata [11]. It is developed by the Open Information Security Foundation as the next generation IDS. Suricata can use the same rules as Snort, and just like Snort, is a network based IDS. Since Suricata can use the same rules as Snort, it was tested with both the rules from Snort, but also with the rules from Emerging Threats [3]. The rules from emerging threats were able to detect SQL-injection patterns when they were done in the URL but not in the body of the HTTP response.

Table 6.1: Suricata Results

Attack	Detected
Cross Site Scripting	Not detected
SQL-Injection	Partly Detected
Broken Authentication and Session Management	Not Detected
Insecure Direct Object Reference	Not Detected
Cross Site Request Forgery	Not Detected
Failure to Restrict URL Access	Not Detected
Unvalidated Redirects and Forwards	Not Detected

6.8.1 Suricata With Emerging Threat Rules

Since Suricata was able to detect some really simple SQL-injection patterns in the URL, it was decided to try and perform the attacks with Suricata as the IDS. The setup of Suricata was inline as a IDS. The version number of suricata was 1.1.1 from the ubuntu software repository. One thing to notice here is that Ubuntu version 12.04 was used in place of 11.10. Because Suricata is only available in 12.04 and an upgrade had been used previously to try a more recent version of Snort. However despite the operating system upgrade the software stayed the same, it was still Apache 2.2 and Tomcat 7 with Java 6 that was used. Mostly the standard configuration for Suricata was used but the IP-addresses was changed to fit the scenario. The rule set from Emerging Threats was also downloaded and the configuration was changed to use those rules and used the reference and classifications files from Emerging Threats ruleset. The full configuration file that was used is in the ZIP-file described in Appendix A. Both Snort and Modsecurity was turned off while Suricata was tested.

The results are shown below in a simple table:

The SQL-pattern which Suricata detected is shown in listing 6.1. The %20 is just a whitespace in the URL. This pattern is harmless against the simplebank website but it is a known SQL-patterns and therefore matches the rules in Suricata.

```
http://33.33.33.10/simplebank/Login?next=SELECT%20*%20FROM%20USERS;
```

Listing 6.1: Simple SQL-injection pattern

These results is almost the same as with Snort. The only difference is that in the SQL-injection the detection is listed partly. This is because Suricata was able to detect a very simple SQL-injection pattern in the URL-header.

Table 6.2: OSSEC results

Attack	Detected
Cross Site Scripting	Partly Detected
SQL-Injection	Partly Detected
Broken Authentication and Session Management	Not Detected
Insecure Direct Object Reference	Not Detected
Cross Site Request Forgery	Not Detected
Failure to Restrict URL Access	Not Detected
Unvalidated Redirects and Forwards	Not Detected

It was however unable to detect the actual attack in the testing or the same SQL-injection pattern when it was used in the body of the HTML and not in the URL itself. This is also not the best results but at least Suricata was able to detect something. Just as with Snort Suricata was also able to detect a simple portscan with NMAP.

Snort was also tested with the Emerging Threats rules for Snort. The result was actually identical to Suricata that was better than the detection with the Snort rules from Sourcefire, the creator of Snort. However this still does not detect the attacks performed in the experiment since they were delivered in the body of the HTTP-request and not in the URL.

6.8.2 OSSEC Detection

OSSEC was also tested to see if it was able to detect more of the attacks which both Snort and Suricata did not detect. The version of OSSEC which were used was 2.6 which had to be downloaded from the OSSEC website since there was no native Ubuntu package for it. The default configuration was used with OSSEC and it was able to detect on the correct ethernet interface with no changes in the default configuration.

OSSEC unfortunately had the same problems as Suricata, it was not able to find any attacks in the body of the HTML only in the URL. However OSSEC used other rules than Suricata and Snort, and was able to detect a simple cross site scripting in the URL as well as the simple SQL-injection pattern which Suricata also detected. The results from OSSEC in the scenario are shown in Table 6.2.

The results are only slightly better than Suricata, and unfortunately all three IDS systems did not perform very well in detecting web application attacks.

However Suricata and OSSEC was not tweaked as much as Snort and there might very well be a way for them to detect the web attacks if they have a different configuration. There was not enough time to try many different configurations of them.

6.9 Validity of the Results

For the sake of correctness. The simplebank, the attacks, the built in sensors and the performance of the experiment was all done by one researcher alone. Because of this, since the attacks and the AppSensor detection points both was implemented by the same person, there was bound to be some bias. The implementer already knew the attacks to detect against! With only one researcher available it is unavoidable and therefore this should be taken into consideration when reading these results. This was one of the reasons why the reference implementation of OWASP AppSensor, and the guidelines were used as much as possible, to make this bias as small as possible.

Chapter 7

Discussion

This chapter will discuss the results from the experiment. It highlights both the general trends that emerged in the results and discuss both the validity of the results, as well as other thoughts and findings some of which are out of the scope for this thesis but which is still worth mentioning.

7.1 General Trends from the Experiment

The Table 7.1 sums up the results from the experiment. The yellow cell on Unvalidated Redirects and Forwards, is yellow because in the newest release of ModSecurity, there is a feature which can check an URL against Googles Safe Browsing API[5]. Unfortunately the rules used in the experiment do not use this feature yet. The other yellow is the SQL-injection which AppSensor was unable to detect. This is not because AppSensor can not detect a SQL-injection but because the SQL-injection detection pattern included with the reference implementation does not match the attack from the experiment. By replacing the SQL-injection detection pattern this attack can be detected by AppSensor.

7.1.1 Unsatisfactory Snort Results

Snort did not detect any of the attacks in the experiment. However that does not make the conclusion that Snort is useless. It only says that in the way the experiment was set up Snort was unable to detect any of the web application attacks. However Snort is designed to detect much more than

Table 7.1: Which Attacks were Detected

Attack	Snort	ModSecurity	AppSensor
Cross Site Scripting	Not detected	Detected	Detected
SQL-Injection	Not Detected	Detected	Partly Detected
Broken Authentication and Session Management	Not Detected	Not Detected	Detected
Insecure Direct Object Reference	Not Detected	Not Detected	Detected
Cross Site Request Forgery	Not Detected	Detected	Not Detected
Failure to Restrict URL Access	Not Detected	Not Detected	Detected
Unvalidated Redirects and Forwards	Not Detected	Can possibly be detected	Detected

just web application attacks. When Snort was configured for the experiment, a simple portscan was performed which Snort detected and gave alerts to, none of the other security mechanisms would pick up this threat and Snort was the only one to detect the portscan.

There is also the possibility that Snort was not configured correctly for detecting the web application attacks and the rules used was not the latest subscribed rules which Sourcefire provide, but the rules that comes with the Ubuntu package of Snort. The setup of Snort could also be malconfigured for the detection of web application attacks, however as mentioned above it managed to detect the portscan so at least some part of Snort detection was working. There is unfortunately not any good explanation for the unsatisfactory Snort results. Even tough the rules said it should detect some attacks and Snort was able to detect other attacks, the attacks in the experiment were not detected. Subsection 7.1.2 highlights the effort to try and get Snort to detect the attacks, but unfortunately was unsuccessful.

7.1.2 Trying to Get Snort to Detect the Attacks

Snort both has rules and a `http_inspect` preprocessor which should be able to detect some web attacks. One rule specifically targeted cross site scripting for example, therefore this attack was specifically tested to get Snort to it since the current results probably does not do Snort justice.

One of the things which were tested was to upgrade the operating system and therefore also the Snort version to the latest. During the thesis a new release of ubuntu, 12.04 came out which contained new packages for Snort, version 2.9.2. After this upgrade the cross site scripting attacks were performed at the simplebank but Snort did still not detect them. After this the configuration was changed and many parameters for the `http_inspect` preprocessor were tried and also some changes to the rules which should have matched the cross site scripting. Still after many different tweaks of the configuration file Snort did not detect the cross site scripting attack.

Lastly the newest ruleset and configuration available on the Snort website where downloaded and tested. But still with this new configuration Snort was unable to detect the simple cross site scripting attack. This was quite unfortunate and the error is believed to be because of some unknown error in the setup of the experiment and not in Snort itself. As mentioned above, the results for Snort are therefore probably not accurate and many IDPS systems would probably have performed better in the experiment. But the results

are the results and while the potential error have been pointed out it still is there.

7.1.3 Very Satisfying AppSensor Results

AppSensor have some very good results from the experiment. This is mostly because AppSensor have some advantages by being built into the application itself. Some of these is highlighted below:

Access Control

Since AppSensor is built into the application itself, it have direct access to the access controls of the application. Often these can be quite complex. Let's consider a simple time registration system. When a user registers his working hours in the system only he is able to view them, this means that another coworker should not be able to view his work hours, but his manager should be able to view them. These kinds of access controls can often be quite complicated and it is impossible for something outside the application to understand them.

The biggest challenge with detecting access control attacks is that since AppSensor is built into the access control system and there is a bug in the access control system that grants a user illegal access to a resource, it is unlikely that a detection point can actually detect that particular bug. If the bug was known it probably would have been fixed in the code anyway. One advantage however is that AppSensor might be able to detect and block an attacker before the attacker discovers the bug.

Application Logic

Since AppSensor is built into the application, it can check for exploits that use the applications logic to make an exploit. These types of attacks come in many varieties and it is impossible to make it a general attack, it has to be specific for the application that is attacked. Since AppSensor is built into the application itself it is able to detect for example the access control violation of the direct object reference attack 5.4. This is because AppSensor can understand the context of the web application and detect these specific attacks. This is biggest difference between AppSensor and the other defense

mechanisms and much of the reason why AppSensor can detect attacks no other defense mechanisms can.

Validating Input

Since AppSensor is built into the application, it known exactly which character encoding the application is using and can perform data normalization in the correct character encoding. Any outside mechanisms can potentially use the wrong encoding and while it tries to sanitize the inputs for the web application, it might do so with a wrong encoding, making it possible to bypass the sanitation filter.

Another issue is that some pages often want to accept dangerous input to be able to write it in text, for example this thesis have SQL-injections inside it, but they are just text, nothing more. One problem with an external filter like a WAF is that it might be impossible without explicitly whitelist the input in order to be able to receive the dangerous input. The application would then properly escape the input itself. By being inside the application AppSensor can easily know which places this short of input should never occur and where it should. For example a blogpost should be able to write about SQL-injections, but a login field should probably not contain SQL-injection input.

More Granular Blocking

Since OWASP AppSensor is built into the application, it is able to do much more granular blocking of malicious activity. It can for example, just suspend a user account if a user is misbehaving while logged in, it could disable some part of the application if it is being exploited, or it could completely block both the user and his IP-address from the network. This kind of granular blocking makes AppSensor able to respond more appropriately to the threat. Not every threat is really serious and often it could be a curious technical guy just checking to see if there are any easy exploits on the site.

The OWASP AppSensor crosstalk [57] contains a list of many ways to block an attacker. These are given in Table 7.2.

Table 7.2: Response Actions

Response Type	Example
Logging Change	Log the incident, record the user performing it
Administrator Notification	Make an notification on the administrator console
Other Notification	Send a signal to the upstream firewall, IDPS or WAF
Proxy	Makes further requests go into a proxy system (honeypot)
User Status Change	Change data validation strictness of this user
User Notification	Send the user an email telling him he is being monitored
Timing Change	Deliberately slow down users requests
Process Terminated	Discard data and force user to start business process from start
Function Amended	Limit on feature usage rate imposed
Function Disabled	Part of the application is disabled
Account Logout	Session is terminated
Account Lockout	Account is locked until administrator resets it
Application Disabled	Application taken offline, temporary static page shows
Collect Data from User	Deploy a Java applet to collect remote IP address

The key point with these response mechanisms is that it is really up to the implementer of AppSensor to specify them and does not have to be restricted by the reference implementation done by OWASP. By restriction, it means that if their website is written in another language, they could implement the AppSensor framework themselves in that language. The implementer also could tune their AppSensor response mechanisms to be custom to their website. For example a webshop could have a malicious user not being able to checkout, if AppSensor has reported them for doing malicious activity. Therefore a full list of which detection points AppSensor would have been never ending.

7.1.4 Drawbacks with OWASP AppSensor

Even though OWASP AppSensor did get very good detection results in the experiment; there are several drawbacks with this kind of approach.

Implementation Specific

Detection points that are built into the application is depending on the implementation itself. This means that they are only as good as they are implemented and are only able to detect the things they are specifically meant to detect. What this means is that if the implementers have a vulnerability they do not know about, they cannot implement a detection point for it either. Since a WAF and a IDPS system often comes with a set of rules which can detect common attacks, it makes sense to use both a WAF/IDPS and at the same time build in AppSensor in the application. That way there is a possibility that if there are some vulnerabilities, which there are no detection points for, it might be caught by the IDPS/WAF.

Another concern here is the possibility for bugs and implementation errors in the implementation of AppSensor. There is always the possibility that an implementation error can make the detection either not working at all or give a false positives or false negatives. The same argument can be said for both an IDPS and a WAF, because they could also have bugs and errors making the detection fail. This is discussed further in Section 8.4.4.

AppSensor Lacks Anomaly Detection

AppSensor, in its reference implementation at least, lacks the anomaly detection that many IDPS and WAFs pride themselves in. Anomaly detection in an IDPS and WAF first learns the normal behavior of an application and then tries to detect when something out of the ordinary happens. The big argument for performing this is that attacks which are currently unknown might be detected, while signature based detection only can detect previously known attacks. While this might be true, the biggest drawback with anomaly-based detection is to actually detect attacks reliably and at the same time have a low false positive and false negative rates. While AppSensor does not have anomaly detection the same way an IDPS or WAF does, it can perform some of the same functionalities. For example the detection points of user trend and system trend from the OWASP AppSensor document [32] does indeed look for suspicious behavior in the application usage. Just as with anomaly

detection these can have false positives. For example with the system trend detection point, which detects if a large number of logins have happened in the last hour. This could happen if the website suddenly became very popular and generated a huge amount of legal traffic. The detection point would probably report false positives.

One advantage of the AppSensor anomaly detection is that like detection points they are often tuned for the specific application since they are built in. This means that the time needed to tune to the system is probably far less than a traditional IDPS or WAF would need. The need to tune the anomaly detection, or trending as OWASP calls them however is quite important, as shown above a false positive could easily occur with the AppSensor trend monitoring.

7.2 Response Mechanisms and Blocking

The different defense mechanisms have different ways of blocking attacks, each with their advantages and disadvantages. While this was not tested in the experiment, a short discussion is provided here to give the whole picture of the comparison because the detection is only part of the functionality of the different defense mechanisms.

7.2.1 Snort Blocking

There are several ways an IDPS system can perform blocking, the two main ways are listed on Sans intrusion detection FAQ [20]. The first method, which is *session disruption* is able to disrupt the connection used by an attacker to carry out the attack. It can send a TCP-reset package to end TCP connections or use ICMP-protocol to disrupt an attack carried out over UDP. As the FAQ points out, a sophisticated attacker with knowledge of TCP could circumvent his way around the session disruption. The other method, which is complete blockage of the IP-address should be used with care as the FAQ suggests. . This is because an attacker could use the blocking features to his advantage by having the IDPS system block valid traffic, which in practice makes a denial of service attack.

These two types of blocking are certainly useful, but in a web application context they mostly work at a lower level than the web application itself. However an attack carried against a web application is usually performed

Table 7.3: ModSecurity Actions

Disruptive Actions	Causes ModSecurity to do something. Often means blocking, but not always
Non-disruptive actions	Do something, but that something does not and cannot affect the rule processing flow.
Flow actions	These actions affect the rule flow.
Meta-data actions	Meta-data actions are used to provide more information about rules.
Data actions	Not really actions, these are mere containers that hold data used by other actions.

over HTTP, which is transmitted over TCP. So the response with either TCP session disruption or complete blocking would prevent the web application attack. However as mentioned above this could also be user by an attacker to perform a denial of service attack.

The Snort manual [49] also lists a third option, which is mainly a variant of the session disruption action. This action is called React and it sends a HTML-page on a session and then resets it. The HTML-page can be custom and control of the HTTP-headers is also allowed, so a custom HTTP response code can be used.

7.2.2 ModSecurity Blocking

ModSecurity have several actions it can take according to the ModSecurity manual [56]. These actions have been divided into five categories.

The different actions that ModSecurity can perform are many, but since they are divided into different groups it is easier to get a grasp of them. The disruptive actions are actions, which deny or block. One example of a disruptive action is to send the request to a proxy or send a redirect to a page informing that such attacks is not tolerated. The disruptive actions is similar to the session disruption and blocking which Snort can do, but it also have some additional options such as the proxy action which Snort does not have.

The non-disruptive actions have some interesting features, such as sanitizing the request before handling it to the web application. This means that if an attack is launched ModSecurity can thwart it, but it does not disrupt or

block the connection. This makes it impossible for an attacker to misuse the feature for denial of service attacks as can be done with the blocking options. There are many other options here, such as settings a environmental variable for Apache or pause the connection for a certain period of time.

The last three categories of actions is mostly about how ModSecurity parses and handles the rules it matches against. These actions can be used to manipulate or do other actions with the data that is being processed by ModSecurity. Examples of these involves, setting the severity rating of the rule, skip rules on successful match or log some extra information when this rule is matched. These actions are not detailed here since they are mostly relevant for the internal detection which ModSecurity performs.

7.2.3 AppSensor Response Mechanisms

The subsection 7.1.3 contains most of the details regarding AppSensor response mechanisms. The main point is that it is possible to write custom actions that are part of the application and can manipulate the application behavior in case of an attack. AppSensor can also take many of the ideas that ModSecurity and Snort uses and apply them. However it could be difficult for AppSensor to utilize the low-level session disruption that Snort and ModSecurity uses since the application generally does not have this low level access to the webserver.

Chapter 8

Comparison

This chapter will compare a IDPS, a WAF and an AppSensor based application. First a general overview of the technologies and their strengths will be given and afterward a list of attacks and how these different technologies can protect against it, if they can. The comparison will focus on web application vulnerabilities, meaning that other types like email, LDAP and other services will not be considered. The comparison starts with a discussion of general IDPS and WAF systems and thereafter discusses the concrete products used in this thesis, snort and ModSecurity.

Both other literature and the experiment will be used as a basis for the comparison of the different products.

8.1 IDPS Overview

An IDPS is mostly defending at the network layer in the infrastructure and not so much on the application layer. This makes the IDPS a very useful addition to any network, but in this comparison it will fall short since it is not meant to be purely about web application vulnerabilities. The main advantage of the IDPS is that it is able to understand low-level traffic and protocols [50]. The other detection points higher up in the ISO OSI models layers won't be able to detect for example a syn-flood attack, because it operates on a level too low for a WAF or the application itself to detect. This low-level detection might be attacks that can have an effect on a web application, because for example an attack on the DNS-server would render the web application useless since no one can get to it by DNS name. Another

example is a simple syn-flood attack or a sockstress attack[22] which would make a denial of service attack and also prevent users from accessing the web application.

In a network topology the IDPS sensors is used to protect almost every aspect of the network and not just the web application. This means that an IDPS might protect both the email-server, the web-server as well as users desktops or laptops. Since it operates at this low-level it means it spans many more services which relies on the low-level services, such as TCP which is used by both the web application and by the email SMTP server, while an IDPS would not necessarily understand the SMTP or HTTP protocols. IDPS systems also comes in many different flavors, from the ones that monitors the big incoming connections to a corporate network to a small IDPS system running on every host computer. Since they come in so many different flavors, they can be used many places in the infrastructure as detailed in Section 2.1.

Since an IDPS can also understand up to layer 7 traffic, it is completely possible that an IDPS can do much of the work a WAF can do in addition to all the low-level protocols it is able to handle. For a business, if an IDPS system is able to detect many common web application exploits, like for example injection attacks by analyzing the HTTP-portion of a network packet. It can in many ways replace a WAF. A WAF may also perform analysis of the cookies and state of the HTTP-traffic. However there is nothing preventing an IDPS system from being able to also understand the states and session created using cookies if it is sophisticated enough. All this means the lines between an IDPS and a WAF can often be quite blurry and hard to make a real distinguishing between them.

8.2 WAF Overview

A WAF is only able to protect a web application. It usually sit in between the traffic flow to the web application or even on the same server and monitors and blocks attacks on the web application. A WAF unlike a IDPS does only understand the HTTP(S) protocols and perhaps other related protocols such as Javascript, SQL, XML, cookies etc. It is specialized in detection at this level. This mean that it can easily detect known attacks such as XSS or SQL-injections since they are easy to recognize. The biggest difference between a WAF and an IDPS is that the WAF only understands the HTTP(S) layers, while the IDPS understand and detects attacks in the lower level protocols too. This oftem makes the WAF perform more accurate in detecting web

application attacks compared to a IDPS. However since both understand HTTP they can largely overlap. By having both an IDPS and a WAF that overlaps can be an advantage, this means that if one of them has an error there is a chance that the other one catches it.

In addition to filter and scan the traffic that goes to a web application, a WAF can be configured to have some intelligence about the web application itself. Such as restriction for certain web pages by IP-address or other types of restrictions. For example it can prevent the usage of other HTTP methods than GET for a specific page.

A WAF can produce something called a virtual patch, meaning that if a vulnerability is found in a web application, instead of patching the application itself a patch can be written as a virtual patch for the WAF. That way an exploit against the vulnerability is blocked by the WAF until a proper patch in the application itself can be issued. The advantage is that the vulnerability is blocked early, the disadvantage is that if the virtual patch need to be carefully written so it does not malfunction or creates false positives.

The main thing that a WAF can not protect against is application specific vulnerabilities, such as an attacker bypassing the authorization by accessing a resource directly with URL guessing. A WAF can detect generic attacks such as XSS and SQL-injections that have the same characteristics across different web applications. This does not make a WAF not useful, quite the contrary. Many attacks today are generic attacks that works on many different web applications, at least for finding the vulnerability. To craft a SQL-injection to actually do harm requires some manual work to tailor the SQL-query to the specific web application, but the detection can still be done in a generic manner.

8.2.1 Anomaly Detection

One way both an IDPS and a WAF uses to overcome the limitation rule-based detection is with anomaly detection. Anomaly detection is covered in Subsection 2.2.2, but it let an IDPS and a WAF detect previously unknown threats. The big benefit with anomaly detection is that it can learn and be tuned to the normal traffic of an application and then be able to detect anomalies, which often can be attacks. The weakness is that anomaly detection has a high degree of both false positives and false negatives. The counter argument AppSensor makes is that since it understands the application better, since it is tailored to it, it can do just as good a job with detecting these

attacks but with a lower false positive rate.

8.2.2 A Note About Stateful Packet Inspection

Stateful Packet Inspection is a feature many IDPS systems boast. It is essentially that the IDPS inspects the whole packet, meaning the contents of the packet and not just the headers. This often makes an IDPS able to detect many of the same vulnerabilities inside packets the same way a WAF does. Even though a WAF tends to be better at understanding web traffic, meaning cookies, Javascript and so on, there is nothing preventing an IDPS system from implementing the same features. Because of this it is often difficult to draw a hard line between a WAF and an IDPS system. There are however a few keypoints which most of the time make them different.

- WAF is often able to decrypt and inspect HTTPS traffic
- WAF is often better at learning the web application traffic (if a positive security model is used)

However all these points can be proven wrong, there are no rules that say that a IDPS should never be able to decrypt HTTPS traffic, nor that it could not learn the behavior of a web application just as well as a WAF. Because of these points there is often hard to draw a line between an IDPS and a WAF.

8.3 AppSensor Overview

AppSensor is able to detect many generic attacks such as SQL-injection and XSS just as a WAF is able to, but since it is built into the application itself, it is also able to block business logic attacks and similar context sensitive attacks. If we look at the example from the WAF where an attacker accesses a resource directly. In an AppSensor powered application it can detect that an unauthorized user is trying to access the resource and if the user tries many other resources by URL-guessing, it might block the users IP-address from the application entirely. The big weakness is that since AppSensor has to be built in, the developers building the application needs to think about and handle such incidents or else the application will be vulnerable. This happens despite the presence of AppSensor detectors other places in the application.

This makes AppSensor potentially as good as a WAF for detecting general threats against a web application, however since it has to be implemented

for every new application there is a higher risk that there could either be a mistake in the implementation or some spot of the application which the application developers have forgotten.

The biggest drawback and advantage with using AppSensor is that it has to be built into the application itself and can not be a drop in solution like a WAF or an IDPS. This means that while potentially AppSensor can detect and prevent all attacks on a web application, it is not practical to make that assumption. Since AppSensor is built into the application and can only detect what the developers make it detect, it suffers the same consequences as a regular web application, meaning that if the web application have a vulnerability, the chances are high that there is no detection point which can detect and prevent the vulnerability from being exploited. With this argument one can assume that an application with AppSensor is probably not more secure than an application without AppSensor. However, AppSensor lets the application detect even potential attacks. So if the application is attacked with for example a simple SQL-injection, which fails because the application properly sanitizes its input, with AppSensor the attack still gets detected. If the same attacker then proceeds to try other simple attacks which the application also blocks but still detected AppSensor is able to identify the attacker and take preventative measures. AppSensor can then take appropriate action against that user to prevent the user from trying further exploits. This means that while the application might have a vulnerability that it can not detect, it can still thwart the attacker before he finds it.

Since AppSensor is built into the application, it is also closer to the data it wants to protect. This can be a benefit since it knows how the data is handled inside the application and can therefore be built with that in mind. For example, in the simplebank application, AppSensor can detect if a large transaction is taking place and thereby alerting the administrators of the simplebank that something is happening. Another benefit by being closer to the data is that it can prevent an attacker from circumventing the entire security layer. If the network setup would allow an attacker, which had gained access to the network where the application was running. The attacker could make requests directly to the web application server and thereby circumventing the entire WAF which would normally be placed in front. The attacker is not able to circumvent AppSensor since it is part of the application itself. The only way would be to gain direct database access and read and manipulate the data stored there. Thus by being closer to the data it is trying to protect, AppSensor has some benefit compared to the other tools. However, this assumption can be wrong since there are Host-based IDPS systems and a WAF can very well be running on the same host as the web application server.

8.4 Recommended Use of the Security Mechanisms

Some recommendations for using different security mechanisms in web applications will be given in this section. The recommendations will be given for both an existing project as well as for new projects. The projects, which are discussed is assumed to be web application projects as this is the focus of the thesis.

8.4.1 Defense in Depth

Defense in depth is a principle in security that means that security should be implemented in depth. Looking at the information centric defense in depth from [43]. The web application with its data would be the entity that need most protection. This is illustrated in Figure 8.1 which contains the data in the middle, then the application, the WAF and the IDPS on the outside. This figure could have been larger with many more layers, but this will suffice for the point illustrated here.

The idea with having these defense mechanisms in layers is that if the outer layer can not catch an attack, there is still another layer which might catch the attack. Even tough these layers can be redundant there is still the possibility of errors in each of the layers and there are also many ways an attacker can get around some of these defenses. For example an attacker can use a special encoding to trick the IDPS to not catch the attack, but the same special encoding might not work on the WAF which is behind.

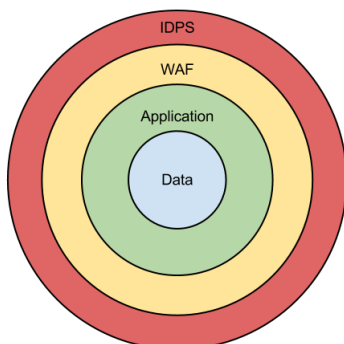


Figure 8.1: Defense in Depth Illustrated

8.4.2 Why Not Take All of Them

If the project can afford to have all three of the security mechanisms, then off course, all three should be applied. One of the benefits to having all three is that they excel at different levels of detection as shown in Figure 8.2. However having all three can lead to high costs in terms of initial setup and configurations, as well as a higher chance of false positives since there are now three sources that can get a false positive alert and not one or two. There are also higher costs for doing the initial setup and later on tune and maintain all three. If the application changes both the IDPS, the WAF and AppSensor would need to be configured for that change and as usual the cost is higher when the number of devices are higher.

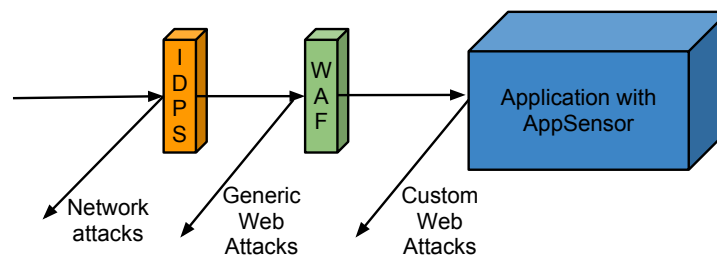


Figure 8.2: Showing the main differences between IDPS, WAF and AppSensor

In addition to these costs, the possibility that all these mechanisms can provide unexpected side effects is also a possibility. This can happen when for example the WAF sanitizes the input when the application needs to have input which are considered harmful. For example in order to post about cross site scripting on a forum, the forum need to accept cross site scripting in its input and properly escape them by itself. If a WAF sanitizes the input it could be double encoded resulting in the original characters being lost in the process. This might make the AppSensor detection not accurate enough, since many attacks would be blocked before they reach AppSensor. This might make it hard for AppSensor to give a proper analysis of the attacks it detects and the severity of them. For example, if an attacker first tries many simple SQL-injections, which all are blocked by a WAF. AppSensor therefore does not take any actions against the user, so when the user finally finds a vulnerability, it might be too late because from AppSensors view this is just one attack attempt and will not trigger the proper response.

8.4.3 IDPS or WAF

If a project wants to have only one external defense mechanism in front of the web application, what should it choose? An IDPS or a WAF? From the results of the experiment, ModSecurity outperformed Snort when it came to web application defense. However one should consider that the experiment was only done with one type of IDPS and one type of WAF. If other IDPS and WAFs had been tested the results might have been different. Another consideration is that the test only considered web application exploits. So the choice between either a WAF or IDPS will come down to if it is only web application protection that is needed or if a whole network needs protection and the web application is only part of that network. In the latter case then an IDPS system would be preferably, since it can protect against a wide range of attacks not only web application based attacks. If there is only the need for web application security, then a WAF would be the best choice.

8.4.4 AppSensor or WAF

One difficult decision is to choose between a WAF and AppSensor as a means to identify attacks and protect a web application. This is not to say that the application should not be written without security in mind, but it is not always enough and mistakes are made. Because of this it is often valuable information to know if somebody is trying to attack the application or not. If a project has to make a decision between a WAF or implement AppSensor there are some though choices that needs to be made. As with everything the answer depend but some considerations are listed below.

- Is the goal to protect an existing web application or is the application being developed from scratch?
- Is the project able to use the reference implementation of AppSensor or does it need to implement a framework from scratch themselves?
- Are there many frameworks or other network equipment that make the detection difficult?
- Does the architecture support many of the recommended response mechanisms?
- Is there a significant enough security requirements to invest in building in AppSensor?
- The need to change the application in the future

- Is there much security knowledge on the development team

The questions asked above are all important to consider if a project is trying to consider between a WAF and AppSensor. It is also wise to consider that AppSensor, since it is built in, often have a lower false positive rate than a WAF and that the WAF would need to be configured and tuned for the application it is protecting. So while AppSensor might take development time, a WAF also takes time to deploy, tune and get it to work correctly, it is not something that can easily just be added. This needs to be taken into consideration if a project is choosing between a WAF or AppSensor.

If these points are taken into consideration there is according to the experiment and the theoretical comparison a benefit to choosing AppSensor over a WAF. In the experiment AppSensor was able to detect more attacks than the WAF. AppSensor is also able to perform a much more granular responses to attack than what a WAF is able to do. So while a WAF might be good protection, according to both the experiment and the literature AppSensor would perform better. However having both would not be a bad thing according to the defense in depth principle, but as mentioned above one need to be careful since many of these technologies can manipulate the traffic and hence making it harder for the other to detect attacks.

If the application is of such a nature that it will often be changed, AppSensor might have an advantage. Since AppSensor is built into the application, it is easier to also update the detection points and response mechanisms when changing the application. This change is also something that AppSensor might get for free, since it uses the application itself. For example, if the application have a whitelist of characters it accepts and there is a detection point which detects every time something is not on the whitelist. If the whitelist changes, AppSensor will automatically use the new changed version. If a WAF was doing the same sanitation against the whitelist, the WAFs copy of the whitelist also needs to be updated. Since the WAF is further away from the application change, it is easy to forget to update the WAF.

One big advantage that a WAF has, is that since it comes with built in rules, it comes with great security knowledge. If the developers developing a web application have little knowledge about security, this might be a huge benefit since the WAF implicitly contains expert security knowledge. AppSensor on the other hand only has its manual from OWASP, which contains some expert knowledge. While this is certainly useful, it still requires much more knowledge about security from the developers than a WAF would require. On the other hand, a WAF needs tuning to the application and should probably not be seen as a appliance which can be deployed without any

Table 8.1: Comparison of Strengths and Weaknesses

	Strengths	Weaknesses
IDPS	<ul style="list-style-type: none"> • Low-level protocols • Protects more than web application • Can perform blocking • Can detect anomalies 	<ul style="list-style-type: none"> • Limited understanding of HTTP(S) • No understanding of application logic • Furthest away from the application data
WAF	<ul style="list-style-type: none"> • Understands HTTP(S) • Can detect anomalies • Can perform blocking 	<ul style="list-style-type: none"> • Only protects Web application • No understanding of application logic • Away from the application data
AppSensor	<ul style="list-style-type: none"> • Understands HTTP(S) • Understands application logic • Can perform some blocking • Close to the application data 	<ul style="list-style-type: none"> • Only protects Web application • Have to be customized for every application

security knowledge.

8.5 Comparison Summary

Below are some of the main points from the comparison collected in Table 8.1. This table provides an easy overview for the different strengths and weaknesses of the three security tools and summarizes the comparison of the products. The different tools each have their strengths and weaknesses and the comparison have tried to highlight the differences as well as the commonalities.

Chapter 9

Conclusion

The conclusion repeats the answers to all the research questions and summarizes the results from the experiment. In addition it will discuss some topics for further research.

9.1 Research Questions

The research questions were developed to help answering the overall goal of the thesis, which was to compare the detection part of different web application security products. These products were divided into three categories, namely, an IDPS system, a WAF and at last OWASP AppSensor. The research questions which were developed where the following:

- RQ1: What is the current state of application-based intrusion detection and prevention systems?
- RQ2: How does OWASP AppSensor compare to other IDPS technologies?
- RQ3: In the given scenario, what are the benefits of using AppSensors compared with trying to stop the attacks in a IDPS or web-application firewall?
- RQ4: How hard is it to built AppSensor into an application?

The first and second research questions are answered by doing a simple literature review of the latest research on web application intrusion detection and prevention, spanning from both IDPS systems, to OWASP AppSensor

with many other mentions also. The second question is partly answered with this literature study and partly answered with the experiment performed.

The third and fourth research questions are answered with the experiment. The experiment is to make a scenario of a bank, which only has a website which its customers interact with. Then a set of attacks were developed which could exploit the bank and three different security mechanisms where each fitted one of the three categories were placed in front of the web application and the experiment observed which kind of security mechanisms were able to block what.

9.1.1 Research Question 1

Chapter 2 goes into depth of what the current state of attack detection systems are. In essence there were three big types of attack detection systems for web applications. The first one is intrusion detection and prevention systems. These systems are more broad than the others, because it does not only protect the web application, but also protects the operating system and other applications. IDPS systems come in many different sizes and types, but the two main types for this thesis is network-based and host-based. For this thesis, the main difference between them is their placement in the network. In the thesis, both of these are in front of the web application and inspect all the traffic that goes to the web application.

The second type of security mechanisms was the Web Application Firewall (WAF). The WAF is similar to an IDPS system in that it inspects the traffic that comes into a web application. The main difference is that a WAF is only focused on HTTP-traffic and other web application related traffic. This means that often a WAF is better than an IDPS system at detecting web application attacks. Often these types of systems can be configured in a positive security model, meaning that they only allow traffic that seems secure into the web application. Since a WAF is only interested in the web application traffic, it is generally argued that a WAF is better at creating a realistic model of what is legal and allowed traffic to a web application.

The last type of web application security mechanisms which were found, are OWASP AppSensor. OWASP AppSensor is a project, which takes the idea of custom-built sensors or detection points in a web application to be used to detect attack on the web application. This is a pretty new approach because it lies in the application itself and also that it is not a simple general drop-in solution that can be placed in front of any application like an IDPS or a WAF

can be. It is these properties that make AppSensor interesting to compare with the other solutions, both because it requires a significant higher cost for a project to implement, but it can detect attacks which are almost impossible for the other solutions to detect, at least without custom rules.

The first research question therefore found the three biggest alternatives in the field of application-based intrusion detection and prevention systems:

- IDPS
- WAF
- OWASP AppSensor

9.1.2 Research Question 2

The second research question is answered by part literature review and part from the experiment. Since the literature had few concrete comparisons of the different products that was published openly and available to the public, the experiment was developed to see if the claims from OWASP was actually true. The other part of the experiment was to see how difficult it is to implement AppSensor into a simple application and how much work is required in order to use AppSensor.

To answer the second research question, the literature most of the time where consistent with the results from the experiment. One exception was that Snort was unable to detect any of the web application attacks, however as discussed earlier this might be a configuration error or some other errors in the setup. The biggest weakness of the experiment is that some concrete IDPS and WAF had to be chosen to perform the experiment and the results could be quite different if any other IDPS or WAF is being used. However based on the experiment it seems that a WAF is well suited to defend against well known web application attacks, which can be easy to detect across applications because they use the same technique. AppSensor shines in that in addition to detect the well known web application attacks it is also able to detect attack which exploits the internal workings of an application, such as failure in access controls mechanisms.

9.1.3 Research Question 3

Research question 3 is quite interesting because there are both benefits and drawbacks to detect and respond to attacks by using AppSensor. This question is answered in the discussion chapter 7. The main benefit with using AppSensor to detect attacks is that it is inside the application and close to the data. Because of this it is able to understand the applications logic, business logic, authorization logic and other parts of the application. When reacting to attacks AppSensor is able to have the application itself respond to the attacks by blocking a user, logging him out or disable the entire application. These granular response mechanisms is much more powerful and more flexible than the more simple block or interrupt which an IDPS or a WAF is able to do. However in the testing it was discovered that the WAF is also able to disable the application or reroute the traffic to a honeypot system. However it is still not as flexible as AppSensor.

One drawback with AppSensor is that since it is so close to the application, it is the last layer of defense. Defense in depth principle says that the defenses should be put i layers around the application such that if one layer either can not detect an attack, the next layer may be able to detect the attack. The other drawback with AppSensor is that it must be built into an application, this might not be a problem if an application is written from scratch but for an existing application it would be more difficult, especially if the application have no way of supporting the response mechanisms such as a forced logout, account lockdown or disable part of the application.

9.1.4 Research Question 4

The last question is a hard one to give a concrete answer to and the answer from the little experiment is that, it depends. However if the implementation is broken into four different scenarios it might be easier to give an answer.

- New project using the reference implementation
- Existing web application using the reference implementation
- New project making its own implementation
- Existing web application making its own implementation

The first type is the easiest to make use of AppSensor, since the web application being developed can be developed with AppSensor in mind and since the business logic must be written from scratch it is easy to include

detection points. The response mechanisms can also easily be tuned to the application and opposite. So a new application being built might include ways of dynamically disable features or take the application offline and other such response mechanisms that might be hard to add to an existing application. In the experiment there was not many difficulties with implementing the detection points, however since the response mechanisms were omitted from the implementation it is hard to say how difficult it would be to implement those.

The next type is an existing application wanting to build in AppSensor by using the reference implementation. Since the reference implementation is in Java, it is assumed that the existing application is also written in Java. While some of the detection points could be implemented outside the application, with for example filters in Java EE. Some of the detection points are directly attached to the business logic and these would be more difficult to implement. The most difficult part would be the response mechanisms to implement. However since it is really up to the implementer to decide which response mechanisms should be implemented, these could be chosen carefully with regards to existing functionality and architecture of the application.

To make a new implementation of AppSensor for a new application would not be too hard. When doing this the AppSensor implementation could be customized to exactly that application and could be implemented in whatever fashion wanted. However while some work would have been duplicate of the reference implementation there are many reasons to make their own implementation. One could be that the application is being written in another language, another could be that the reference implementation does not meet certain requirements, such as test-coverage, performance or other requirements.

The last type is to custom build AppSensor for an existing application. How hard this is is very depended on the existing application, but since AppSensor is being built from scratch it could take advantage of the architecture and design of the application. For example if the application supports filters which can filter traffic before reaching the application, many detection points could be implemented this way just as with the reference implementation. However the existing application could also be a huge challenge and make the implementation very hard, but it is really dependent on the existing application.

9.2 Further Work

Since AppSensor and application based intrusion detection seems to be a new idea and not such a mature field yet, there is still much work to be done.

9.2.1 Larger Experiment

One type of further work could be to perform a larger experiment, with more attacks and another bigger web application. This would really help to compare AppSensor to other intrusion detection technologies and perhaps reveal some new findings. Another benefit be performing a bigger experiment is that a more fair comparison would have been given,

9.2.2 Study of AppSensor in Practice

A study of how AppSensor was used in the industry and also some numbers and experiences from the different places AppSensor are used and how effective it is. With such a study questions such as, how hard it is to implement AppSensor, would be answered more correctly. Other questions that can be answered is how common it is to integrate AppSensor with other applications, such as a SIEM system or a WAF. It would also answer how many actually use the reference implementation and if it is common to implement their own implementation of AppSensor.

9.2.3 More Comprehensive IDPS Experiment

An experiment that covered IDPS-systems more comprehensively and tried to get for example Snort or Suricata to detect attacks which are inside the HTTP-requests and not just in the URL. The reason to why for example Snort was unable to detect the different attacks when even tough it reported that is processed even POST-requests is currently unknown. A new experiment which focused on either getting these IDPS systems to work with web application attacks, or compared other commercial products would be beneficial and perhaps get better results for IDPS systems with regards to web application security.

9.2.4 Another Reference Implementation

One topic for further research would have been to create another reference implementation, or extend the current one with more detection points and perhaps with some built in ways for interacting with other systems. Other benefits with having another reference implementation are that the concept of application-based intrusion detection could be further explored and another implementation could help both implementations become better.

9.2.5 Testing AppSensor on a Production System

One of the last and perhaps hardest pieces of further work is to implement and test the AppSensor concept on a real website. By having AppSensor on a real website, it would be able to find and tune the different thresholds for AppSensor. The false positive rate and false negative rates could also be measured and guidelines for what the different thresholds should be, could be collected.

Other benefits would be to see if how effective the different response mechanisms would be against real world attackers. For example the account lockout could easily be worked around by an attacker. It might also scare the attacker away from trying to attack the website anymore or it might encourage the attacker into trying harder and get around the AppSensor protection system. These questions and many more could be answered by testing AppSensor on a real production system.

9.2.6 Make AppSensor and a WAF Cooperate

In chapter 8 the point was made that with more than one security appliance sanitizing the input or blocking the attack, knowledge of the attack was lost for the next layer of defense. One idea for further work would be to look at the possibility of getting for example AppSensor and a WAF to cooperate. This would solve the problem of information being lost when several defense systems are used. One way to do this is to have AppSensor read the logs of the WAF and track all the incidents it reports. This would need to be true the other way around and AppSensor could be able to trigger an event in the WAF. A research project like this would be very interesting and useful, since it will increase the security according to the defense in depth but at the same time get accurate detection in all the systems.

9.2.7 Test Anomaly Detection Against AppSensor

In this experiment, only the rule-based detection of both the IDPS and the WAF was tested. An experiment, which looked at how anomaly detection in both an IDPS and a WAF would compare against AppSensor, would be very interesting. However this would be very hard to perform since anomaly detection needs both tuning and learning time, which are many sources for uncertainty. But the claims from OWASP AppSensor team that AppSensor is able to detect attacks with a lower false positive rate compared to an IDPS or a WAF would be interesting to test.

References

- [1] Apache - http server project. <http://httpd.apache.org/>, Checked Online May 2012.
- [2] Apparmor. http://wiki.apparmor.net/index.php/Main_Page, Checked Online April 2012.
- [3] Emerging threats. <http://www.emergingthreats.net/>, Checked Online May 2012.
- [4] Google code appsensor. <http://code.google.com/p/appsensor/>, Checked Online May 2012.
- [5] Google safe browsing api. <https://developers.google.com/safe-browsing/>, Checked Online May 2012.
- [6] Jetty. <http://jetty.codehaus.org/jetty/>, Checked Online May 2012.
- [7] Jwall auditviewer. <http://jwall.org/web/audit/viewer.jsp>, Checked Online April 2012.
- [8] Mandatory access control bsd. <http://www.trustedbsd.org/mac.html>, Checked Online April 2012.
- [9] Modsecurity - open source web application firewall. <http://www.modsecurity.org/>, Checked Online May 2012.
- [10] Object-relational mapper. <http://www.hibernate.org/about/orm>, Checked Online April 2012.
- [11] Open information security foundation - suricata. <http://www.openinfosecfoundation.org/>, Checked Online May 2012.
- [12] Ossec. <http://www.ossec.net/>, Checked Online May 2012.

- [13] Owasp - testing for business logic. [https://www.owasp.org/index.php/Testing_for_business_logic_\(OWASP-BL-001\)](https://www.owasp.org/index.php/Testing_for_business_logic_(OWASP-BL-001)), Checked Online May 2012.
- [14] Owasp enterprise security api. https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API, Checked Online April 2012.
- [15] Owasp top ten: Broken authentication and session management. https://www.owasp.org/index.php/Top_10_2010-A3-Broken_Authentication_and_Session_Management, Checked Online May 2012.
- [16] Owasp wiki: Cross site scripting. <https://www.owasp.org/index.php/XSS>, Checked Online May 2012.
- [17] Perl programming language. <http://www.perl.org/>, Checked Online April 2012.
- [18] Perl taint mode. <http://perldoc.perl.org/perlsec.html#Taint-mode>, Checked Online April 2012.
- [19] Reddit cross site scripting. <http://www.reddit.com/r/xss/>, Checked Online May 2012.
- [20] Sans intrusion detection faq: What is active response? <http://www.sans.org/security-resources/idfaq/active.php>, Checked Online May 2012.
- [21] Security enhanced linux. http://selinuxproject.org/page/Main_Page, Checked Online April 2012.
- [22] Slashdot article - new denial-of-service attack is a killer. <http://it.slashdot.org/story/08/10/01/0127245/new-denial-of-service-attack-is-a-killer>, June 2012.
- [23] Snort. <http://www.snort.org/>, Checked Online May 2012.
- [24] Tamperdata. <https://addons.mozilla.org/en-US/firefox/addon/tamper-data/>, Checked Online May 2012.
- [25] Web application security consortium: Cross site scripting. <http://projects.webappsec.org/w/page/13246920/Cross%20Site%20Scripting>, Checked Online May 2012.
- [26] Wikipedia sql-injection. http://en.wikipedia.org/wiki/SQL_injection, Checked Online May 2012.

- [27] Xssed website. <http://www.xssed.com/>, Checked Online May 2012.
- [28] National Security Agency. Defense in depth a practical strategy for achieving information assurance in today's highly networked environments. http://www.nsa.gov/ia/_files/support/defenseindepth.pdf.
- [29] E.B. Barker and J.M. Kelsey. *NIST SP-800-90A: Recommendation for random number generation using deterministic random bit generators*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, January 2012.
- [30] Jim Beechey. Web application firewalls: Defense in depth for your web infrastructure. <http://www.sans.edu/resources/student-projects/200904-01.doc>, March 2009.
- [31] F.P. Brooks Jr. The computer scientist as toolsmith ii. *Communications of the ACM*, 39(3):61–68, 1996.
- [32] Michael Coates. Owasp appsensor. https://www.owasp.org/images/2/2f/OWASP_AppSensor_Beta_1.1.pdf, Checked Online June 2012, Published 2009.
- [33] Web Application Security Consortium. Web application firewall evaluation criteria. <http://www.webappsec.org>, January 2006.
- [34] Federal Financial Institutions Examination Council. Authentication in an internet banking environment. http://www.ffiec.gov/pdf/authentication_guidance.pdf.
- [35] Maximilian Dermann, Mirko Dziadzka, Boris Hemkemeier, Achim Hoffmann, Alexander Meisel, Matthias Rohr, and Thomas Schreiber. Best practices: Use of web application firewalls. https://www.owasp.org/index.php/Category:OWASP_Best_Practices:_Use_of_Web_Application_Firewalls, July 2008.
- [36] G. Dodig-Crnkovic. Scientific methods in computer science. In *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*, pages 126–130, 2002.
- [37] D.G. Feitelson. Experimental computer science: The need for a cultural change. *Internet version: http://www.cs.huji.ac.il/~feit/papers/exp05.pdf*, 2006.

- [38] Paul. Feyerabend. *Against method (1975)*. Published by Verso, 1993.
- [39] Marc Fossil, Gerry Egan, Kevin Haley, Trevor Mack, Teo Adams, Joseph Blackbird, Mo King Low, Deble Mazurek, David Mckinney, and Paul Wood. Symantec internet security threat report trends for 2010, 2010.
- [40] Jeremiah Grossman. Seven business logic flaws that put your website at risk. *WhiteHat Security, October, 2007*.
- [41] Jeremiah Grossman and Trey Ford. Get rich or die trying "making money on the web, the black hat way". https://www.whitehatsec.com/assets/presentations/PPT_BlackHat080708.pdf, 2008.
- [42] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Securecomm and Workshops, 2006*, pages 1–10. IEEE, 2006.
- [43] Peter Leight and Richard Hammer. Defense-in-depth - what is it? *Sans Technology Institute Student Projects, 2006*.
- [44] Symantec Corporation Matthew Conover, Principal Security Researcher. Analysis of the windows vista security model. Technical report, Symantec Advanced Threat Research.
- [45] J.H. Morris Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.
- [46] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of internet background radiation. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 27–40. ACM, 2004.
- [47] T. Pietraszek and C. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, pages 124–145. Springer, 2006.
- [48] The Open Web Application Security Project. Owasp top 10 - 2010. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, January 2010.
- [49] The Snort Project. *SNORT Users Manual, 2.9.2*, December 7, 2011.
- [50] Karen Scarfone and Peter Mell. Guide to Intrusion Detection and Prevention Systems (IDPS) Recommendations of the National Institute of Standards and Technology. *Nist Special Publication, 2007*.

- [51] Bruce Schneier. The process of security. <http://www.schneier.com/essay-062.html>, Checked Online April 2000.
- [52] R. Sekar. An efficient black-box technique for defeating web application attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, pages 8–11, 2009.
- [53] Ofer Shezaf. "modsecurity core rule set": An open source rule set for generic detection of attacks against web applications. *OWASP AppSec Conference - Italy*, 2007.
- [54] Payment Card Industry Data Security Standard. Information supplement: Requirement 6.6 code reviews and application firewalls clarified. https://www.pcisecuritystandards.org/documents/information_supplement_6.6.pdf, Checked Online October 2008.
- [55] W.F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.
- [56] Inc. Trustwave Holdings. *ModSecurity Reference Manual*. http://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Reference_Manual.
- [57] Colin Watson, Michael Coates, John Melton, and Dennis Groves. Creating attack-aware software applications with real-time defences. *Crosstalk - September/October 2011*.

Appendix A

Zip-file Contents

There is a zip-file with the thesis which contains the sourcecode, the configuration files and the rules used in the thesis. These are all included so the results are easy to reproduce and possible errors can be found.

Simplebank

The sourcecode is in the **simplebank** folder. The simplebank application is using Maven to handle the dependencies. In order to run the simplebank application with Jetty, the following command in listing A.1 will run the simplebank application with the embedded jetty application server.

```
mvn jetty:run
```

Listing A.1: Run simplebank with Jetty

Security Systems

In the Snort folder is both the Snort configuration. The folder rules, contains the rules which was supplied with Ubuntu 11.10 and the Snort package. The folder newrules contains the rules which was downloaded from the Sourcefire website and tested with the latest version of snort as mentioned in Section 7.1.2. In the folder newrules is also the new configuration file which was used with the newest rule set, this configuration file was also downloaded from sourcefire.

The folder named ModSecurity contains both the ModSecurity configuration as well as the rules used by ModSecurity.

The Suricata folder contains both the configuration file which is a yaml-style file. It also contains a subfolder with the emerging threat rules which were used with Suricata.

The last folder is named ossec and contains the OSSEC configuration and rules. The folder named etc contains the configuration for OSSEC, and the folder named rules contains the rules.

Appendix B

Screenshots of the Simplebank Application

Pictures of the different screens in the simplebank is included in this appendix.

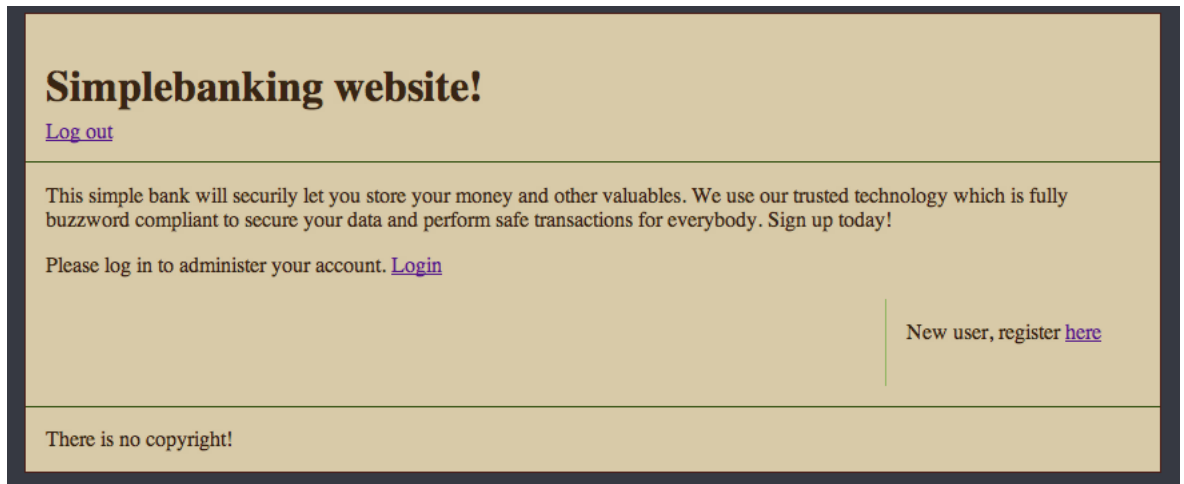


Figure B.1: Frontpage for the Simplebank

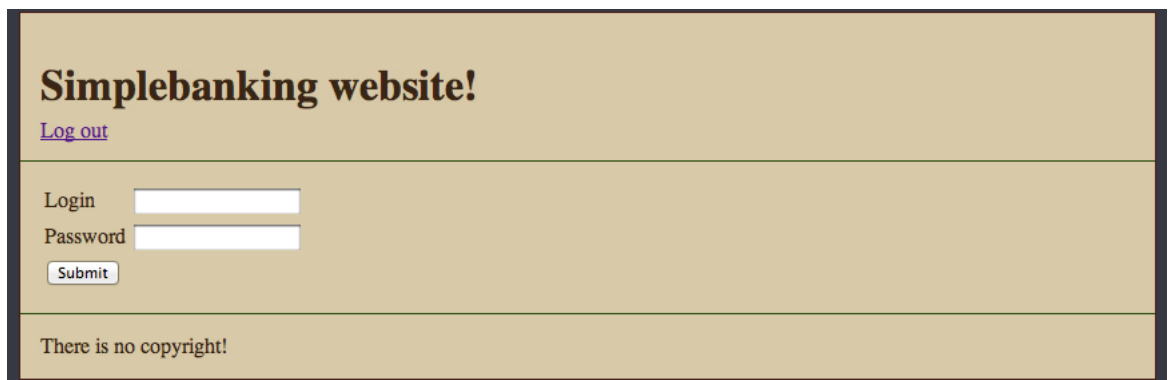


Figure B.2: Loginpage for the Simplebank

Simplebanking website!
[Log out](#)

Username
Password
First Name
Surname
SomethingSecret

There is no copyright!

Figure B.3: Register new user page

Simplebanking website!
[Log out](#)

Overview of account for test

Account Name	Account Type	Money
testAccount	Normal	355.0
superaccount	Credit	10000.0

[Create a new Account](#)
[Transfer money](#)

There is no copyright!

Figure B.4: Account overview page

Simplebanking website!
[Log out](#)

Create a new account

Account Name

Account Type Normal Credit

Money amount

[Accountoverview](#)

There is no copyright!

Figure B.5: Create new account page

Simplebanking website!
[Log out](#)

From:

Accounts: ▾

Amount:

To:

To Accountnumber:

There is no copyright!

Figure B.6: Transfer money page