DEGREE PROJECT

Deception strategies for web application security: application-layer approaches and a testing platform

Mikel Izagirre

Information Security, master's level (120 credits)
2017

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

LULEÅ UNIVERSITY OF TECHNOLOGY

Master Thesis Project

Deception strategies for web application security: application-layer approaches and a testing platform

Author: Mikel Izagirre
E-mail: mikiza-5@student.ltu.se

Supervisor: Dr. Ali Ismail Awad

June 2017
Master of Science in Information Security

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

Abstract

The popularity of the internet has made the use of web applications ubiquitous and essential to the daily lives of people, businesses and governments. Web servers and web applications are commonly used to handle tasks and data that can be critical and highly valuable, making them a very attractive target for attackers and a vector for successful attacks that are aimed at the application layer. Existing misuse and anomaly-based detection and prevention techniques fail to cope with the volume and sophistication of new attacks that are continuously appearing, which suggests that there is a need to provide new additional layers of protection.

This work aims to design a new layer of defense based on deception that is employed in the context of web application-layer traffic with the purpose of detecting and preventing attacks. The proposed design is composed of five deception strategies: Deceptive Comments, Deceptive Request Parameters, Deceptive Session Cookies, Deceptive Status Codes and Deceptive JavaScript.

The strategies were implemented as a software artifact and their performance evaluated in a testing environment using a custom test script, the OWASP ZAP penetration testing tool and two vulnerable web applications.

Deceptive Parameter strategy obtained the best security performance results, followed by Deceptive Comments and Deceptive Status Codes. Deceptive Cookies and Deceptive JavaScript got the poorest security performance results since OWASP ZAP was unable to detect and use deceptive elements generated by these strategies.

Operational performance results showed that the deception artifact could successfully be implemented and integrated with existing web applications without changing their source code and adding a low operational overhead.

Acknowledgements

I would like to thank everyone who has contributed in some way to this work.

Firstly, I would like to thank LTU and its staff for making this learning experience possible. I would like to specially acknowledge all the lecturers and assistants who have participated in different courses of the MSc. Information Security Programme, including the students I've had the opportunity to work with. The diversity of backgrounds and the participation of both on-campus and distance students have made this learning journey very enrichening.

Secondly, I would like to express my sincere gratitude to Dr. Ali Ismail Awad for being the supervisor of this work and also to my thesis opponents Marcus Hufvudsson and Peter Ken Bediako for their useful comments and suggestions provided during the seminars.

Lastly, I would like to thank my family for their support and especially recognize my sister Miren Izagirre and my sister-in-law Itsaso Noya for their interesting comments and discussions.

Table of Contents

List of Tables

List of Figures

Abbreviations

ANN	Artificial Neural Networks
API	Application Programming Interface
APT	Advanced Persistent Threat
CAPTCHA	Completely Automated Public Turing test to tell Computers and Humans Apart
CSRF	Cross-Site Request Forgery
DSR	Design Science Research
DTK	Deception Toolkit
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
LFI	Local File Inclusion
OS	Operating System
OSI	Open Systems Interoperability
OWASP	Open Web Application Security Project
PCA	Principal Component Analysis
REST	Representational State Transfer
RFI	Remote File Inclusion
RQ	Research Question
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
SVM	Support Vector Machine
TCP	Transmission Control Protocol
TLS	Transport Layer Security

URL	Uniform Resource Locator
WAF	Web Application Firewall
XML	Extensible Markup Language
XSS	Cross-Site Scripting

CHAPTER ONE

1. INTRODUCTION

In recent years, the popularity of the internet and web applications has drastically increased. Web applications delivered over the HTTP protocol are the primary way of providing services on the internet [1] and these services play a significant role in our everyday life in a variety of fields such as finance, education, industry, commerce, healthcare and even critical infrastructures. In consequence, web servers and web applications have become an attractive target for attackers and an important vector for successful attacks. Many of those attacks rely on techniques aimed at the application layer (e.g.: SQL injection, Cross Site Scripting, parameter tampering, etc. [2] [3]) and they are not detected by detection mechanisms operating on the transport or network layers such as network firewalls or network Intrusion Detection Systems (IDSs) [4] [5].

In order to detect and prevent attacks targeted towards web applications, specific Web Application Firewalls (WAFs) are commonly used. Although IDS and WAF systems have been studied and employed for decades, they have great difficulty in keeping up with the amount of new attacks that are constantly appearing. The use of zero-day exploits, sophisticated Advanced Persistent Threats (APTs) and other forms of attacks are on the rise and they are able to bypass the protection layer of IDS and WAF systems that usually rely on signature or anomaly detection techniques [6] [7] [8]. The shortcomings of signature-based detection techniques are well known because they can't detect new unknown attacks. The research community has focused its attention to anomaly-based detection techniques aiming to detect new unknown attacks, but the effectiveness of those techniques has also been challenged. Some researchers even state that anomaly detection is flawed in its basic assumptions [8], as machine learning techniques being used are good to find similar events to ones previously seen, but they are not effective to find new unknown events that are not present in the training datasets.

This situation suggests that there is a need to provide extra layers of protection using additional techniques, as existing protection mechanisms don't seem to be sufficient to address current threats.

Deception provides an alternative approach for security that can deliver a useful additional layer of protection. The goal of deception is to deceive and manipulate adversaries inducing them to take actions that are advantageous and aid computer security defenses by using different mechanisms to manufacture, alter or hide the reality they perceive.

Well-known examples of deception and decoy-based tools are honeypots and honeytokens [6] [7]. Honeypots are fake computer systems that look and behave like real systems and can be used to detect attackers and analyze their attacking procedures. Honeytokens are digital entities (e.g.: fake files, fake user accounts, fake database records, fake passwords, fake URLs, etc.) that should never be accessed by anyone and are created for the purpose of triggering an intrusion alarm whenever someone uses them.

The use of deception for computer security is not a new concept but its use has typically been limited to separated or isolated systems such as honeypots. However, any layer from a computer

system that is susceptible of attacks or abuse could potentially benefit from the use of deception for security defense, including application-layer protocols such as HTTP.

The purpose of this work is to study the possible usages of deception for web application security defense by designing application-layer deception strategies and building a platform to apply and test those strategies in the context of web application-layer traffic. The designed strategies will operate at the application layer by creating or modifying deceptive application-layer elements into the HTTP traffic (e.g.: methods, status codes, header fields, cookies, body content element, JavaScript, etc.) with the purpose of detecting or preventing attacks.

1.1. Problem statement

The popularity of the internet has made the use of web applications ubiquitous and essential to the daily lives of people, businesses and governments. Web servers and web applications are used to handle tasks and data that can be critical and highly valuable, making them a very attractive target for attackers and a vector for successful attacks that are aimed at the application layer. Existing misuse and anomaly-based detection and prevention techniques fail to cope with the volume and sophistication of new attacks that are continuously appearing, which suggests that there is a need to provide new additional layers of protection.

Deception provides an alternative approach to defend systems that can deliver an advantageous additional layer of protection. The use of deception is not a new concept, but it has traditionally been limited to ad-hoc approaches realized as single tools or to repackaged entire solutions deployed as isolated honeypot machines. However, deception provides a useful additional layer of protection that could potentially be integrated into any production system at any layer that is susceptible of suffering attacks, including application-layer protocols such as HTTP.

Current deceptive solutions tend to be agnostic to application-layer protocols and they don't really study how deception can be applied at this level. Consequently, those solutions can't properly be used to defend against application-layer attacks that are inherently based on elements from the application-layer itself.

This work aims to study possible usages of deception strategies that could be integrated in the context of application-layer traffic of web applications with the purpose of detecting or preventing application-layer attacks.

1.2. Research questions

RQ1: What is the performance of different deception strategies in detecting or preventing web application attacks?

RQ2: How can a testing platform be developed in order to evaluate the performance of those deception strategies?

1.3. Proposed solution and research goals

These are the goals that this research aims to achieve:

- To study different types of possible application layer (OSI layer 7) attacks when using the HTTP protocol.
- To study and design suitable deception strategies that could be used to detect or prevent those attacks in a protocol-aware fashion based on the features, syntax and semantics of HTTP protocol elements (e.g.: methods, status codes, header fields, cookies, body content element, JavaScript, etc.).
- To design and implement a testing environment that will be used to evaluate the performance of the deception strategies through a set of experiments.

Web application scanners or penetration testing tools will be used to launch controlled attacks against vulnerable web applications in order to measure the performance of the deception strategies.

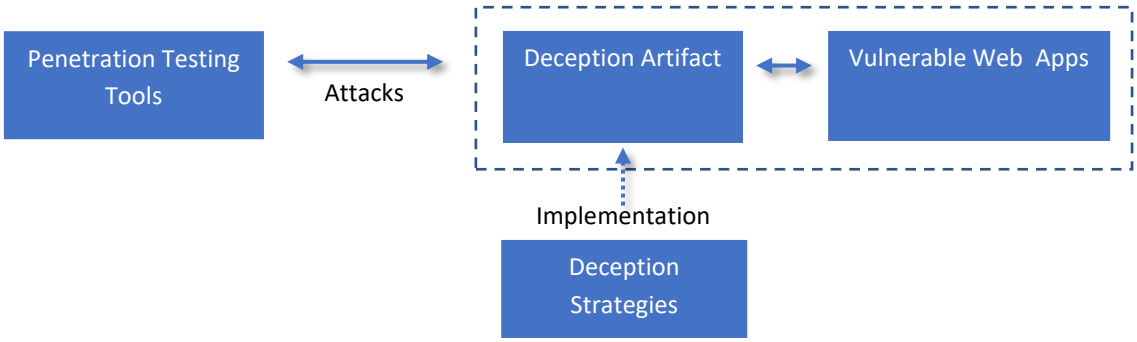


Figure 1 Testing environment overview

Deception strategies will be implemented as a software instantiation. This software instantiation will be capable of generating and injecting deception data automatically into HTTP traffic (e.g.: using a reverse-proxy design or similar [9]) as well as being able to monitor the feedback channels and reacting accordingly (e.g.: modifying responses, triggering alerts for certain events, providing a way of getting information about attacks attempts that have been detected, honeytokens that have been activated, etc.).

1.4. Research contributions

The main contribution of this work lies in the design and performance evaluation of five new application-layer deception strategies that could be applied to detect or prevent web applications attacks.

The use of deception for computer security is not a new concept, but its use still seems to be limited to ad-hoc implementations and isolated repackaged solutions in the form of honeypots. There are theoretical deception models and frameworks to integrate deception into computer systems, however, there are few practical implementations or real uses that are actually applying those models to integrate deception into different layers of production systems and applications.

Taking all this into consideration, the design and performance evaluation presented in this thesis aim to be a contribution by itself, as there is a lack of works following similar approaches of integrating deception strategies into the application layer of production web applications following the planning steps and the guidance provided by a theoretical framework [6].

This work also contemplates the creation of a software artifact implementing the deception strategies to perform the evaluation experiments. Therefore, the engineering problem that is solved with the implementation of the artifact is also intended to be a contribution, since it shows an example of how the engineering problem can successfully be solved.

The accomplishment of this work is expected to be valuable to illustrate examples of deception strategies that could motivate other researchers to use similar approaches or help them think of new applications of deception in computer security defense.

1.5. Delimitation

This research is limited to the use of deception for application layer (OSI layer 7) web application traffic (HTTP protocol). This work includes the creation of a software instantiation implementing the designed deception strategies and this instantiation is built as a prototype for the proof of the suitability of the deception strategies. Although operational performance aspects of the implementation are measured, the actual performance of the artifact itself to generate deceptive traffic or deception tokens is not the goal of this work.

1.6. Thesis outline

The remainder of this work is organized as follows. Chapter 2 provides background information about web application security, Intrusion Detection Systems and computer deception. Chapter 3 presents a literature review to reveal the current state-of-the-art of the use of deception for web application security defense and web application security testing, followed by Chapter 4 which explains the research method used to conduct this work. In Chapter 5 different deception strategies are designed, which are then tested in the experiments covered in Chapter 6. Results are discussed in Chapter 7 and finally Chapter 8 provides some final conclusions and directions for future work.

CHAPTER TWO

2. BACKGROUND

2.1. Web application vulnerabilities

Web applications are client-server applications based on the HTTP [10] protocol that usually manage HTML, JavaScript, JSON and XML content data.

There is a large amount of literature available covering web application vulnerabilities and associated common attacks [1] [2] [3] [4] [5] [11] [12] [13]. Although the types of vulnerabilities and attacks are known since a long time, there is no single solution to mitigate all of them and the limited security support offered by web application frameworks and the popularity and growing complexity of web applications have made them more prone to attacks than ever [5].

For example, injection and session management vulnerabilities have occupied the top 3 issues of the OWASP Top 10 2010 and 2013 editions [2] and they are still present in the updated version of the document to be released in July 2017 [3].

Common web application vulnerabilities can be classified into three types [5]: Injection Vulnerabilities, Business Logic Vulnerabilities and Session Management Vulnerabilities. The following diagram shows a classification of several types of attacks that are commonly used to exploit each type of vulnerability.

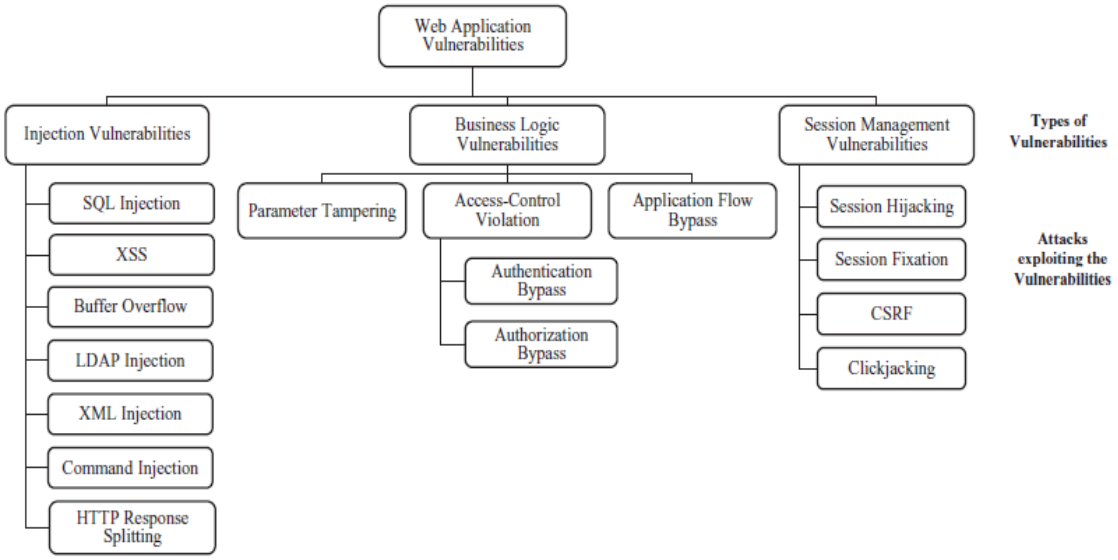


Figure 2 Web application vulnerabilities classification [5]

Injection Vulnerabilities: An injection occurs when an attacker sends untrusted data as part of an apparently legitimate command or query in order to trick the interpreter of the application and execute unintended commands. Most common types of injections are SQL injection, Cross Site Scripting (XSS) and LDAP injection [2].

Business Logic Vulnerabilities: Business Logic Vulnerabilities allow malicious attackers to manipulate the legitimate logic of the application and execute illegitimate transactions. These vulnerabilities are usually exploited by modifying parameters, bypassing authentication and authorization constraints and violating the normal flow of an application.

Session Management Vulnerabilities: Session Management Vulnerabilities allow attackers to read or manipulate session variables that are used to keep the state of web applications (e.g. state that a user is logged into the application and has certain authorization rights). Session hijacking and fixation attacks target on the session ID of the user whereas Cross Site Request Forgery (CSRF) and clickjacking aim the client's browser to submit requests that the legitimate user would not want to submit.

The OWASP Top 10 document [2] [3] is commonly referenced when mentioning the most common web application security flaws. The document describes the 10 most common flaws based on data collected from hundreds of organizations and over 50,000 applications and APIs, and it classifies them based on their prevalence and consensus estimates of exploitability, detectability and impact. The OWASP Top 10 document also gives mitigation recommendations aiming to help developers and organizations to better protect their applications. The latest version of the OWASP Top 10 2017 is going to be released in July or August 2017, and it will update the previous OWASP Top 10 2013 document. However, the Release Candidate 1 version of the 2017 document has already been published. The OWASP Top 10 2013 and 2017 documents classify the top most common flaws into 10 categories that are summarized below.

Table 1 OWASP Top 10 2017 security flaws

OWASP Top 10 2013	OWASP Top 10 2017 (RC1)
A1 – Injection	A1 – Injection
A2 - Broken Authentication and Session Management	A2 - Broken Authentication and Session Management
A3 - Cross Site Scripting (XSS)	A3 - Cross Site Scripting (XSS)
A4 – Insecure Direct Object References	A4 – Broken Access Control
A5 - Security Misconfiguration	A5 - Security Misconfiguration
A6 - Sensitive Data Exposure	A6 - Sensitive Data Exposure
A7 – Missing Function Level Access Control	A7 - Insufficient Attack Protection
A8 - Cross-Site Request Forgery (CSRF)	A8 - Cross-Site Request Forgery (CSRF)
A9 - Using Known Vulnerable Components	A9 - Using Known Vulnerable Components
A10 – Unvalidated Redirects and Forwards	A10 - Underprotected APIs

The 2017 version merges previous A4 and A7 categories into a renamed "A4 – Broken Access Control" category, drops the "Unvalidated Redirects and Forwards" category and introduces 2 new categories: "A7- Insufficient Attack Protection" and "A10- Underprotected APIs". It is significant to note that the top three categories remain unchanged (Injection, Broken Authentication and Session Management and Cross Site Scripting (XSS) and they keep being considered as the three most important web application security flaws according to the OWASP nonprofit organization.

Information about the OWASP web application flaws categories is summarized below.

Injection: An injection occurs when an attacker sends untrusted data as part of an apparently legitimate command or query in order to trick the interpreter of the application and execute unintended commands or access unauthorized data. Most common types of injections are OS command injection, SQL injection and LDAP injection.

Example: An attacker sends an HTTP POST request whose username attribute has the following value:

```
Username = ;)'DELETE * FROM USERS
```

If the interpreter does not properly treat quotes and scape characters, the malicious query could be executed on the server. If there is a table called USERS, all records will be deleted.

Mitigation: The application must avoid input interpretation and must use a safe parameterized API instead.

Broken Authentication and Session Management: Poor authentication and session management implementations could allow attackers to steal passwords, session tokens or exploit other flaws to assume different user identities.

Example: An attacker gets access to the password file or database of the system. If passwords are not properly hashed, he or she will know all passwords of all users.

Prevention: Application should meet proper authentication and session management requirements.

Cross-Site Scripting (XSS): A XSS attack takes place when an attacker can inject client-side scripts on web pages that will be delivered to users. The victim's browser will execute those scripts that might be able to hijack sessions, deface websites or redirect the user to malicious sites.

Example: If a web page uses untrusted data as part of its HTML code, e.g.:

```
<input name='creditcard' type='TEXT' value='"request.getParameter("CC")+ "'>"
```

An attacker might modify the 'CC' parameter to inject a custom script. In this example, the victim's session id will be sent to the attacker's website and sessions might be hijacked:

```
<script>document.location='http://www.attacker.com/cgi-bin/cookie.cgi?
foo='+document.cookie</script>
```

Prevention: One of the most important recommendations consists on properly escaping all untrusted data on the HTML contexts of the document.

Broken Access Control (merges Insecure Direct Object References and Missing Function Level Access Control): This category includes flaws related with insufficient or incorrect access control mechanisms of applications that allow authenticated users to bypass the restrictions on what they are allowed and not allowed to do in the application.

Insecure Direct Object References refer to programs that use direct references to internal implementation objects (files, directories, database rows) that are exposed and an attacker can manipulate to access unauthorized data.

Example: The source code of an application running on the server includes code like:

```
String query = "SELECT * FROM accts WHERE account = ?param1";
param1 = request.getParameter('account')
```

If the attacker modifies the acct number (656565) on the request URL: e.g.:

```
http://example.com/app/accountInfo?account=656565
```

He or she will be allowed to access other user's accounts just by changing the account number.

Prevention: Enforce additional access checks or use per user/per session indirect object references (e.g.: random IDs generated per session for a user that map to the real referenced IDs on runtime).

Missing Function Level Access control refers to the verification of access level rights only just before making functionality visible in the UI can be dangerous if further server-side access control checks are not performed. An attacker might be able to forge the UI components and requests in order to access unauthorized functionality.

Example: An application generates a webpage showing a disabled button that the user is not authorized to click on. However, the URL of that functionality is accessible and can be accessed from the web browser.

Prevention: Perform server-side authorization checks using fine-grained authorization role schemes.

Security Misconfiguration: Server, database and development frameworks should be properly configured and maintained. Insecure default settings and outdated software versions are one of the most typical sources of vulnerabilities.

Example: Using default passwords on database servers, keeping administrator interfaces intended for development when the application is on production, and relying on outdated libraries of frameworks that attackers can easily exploit.

Prevention: Carefully plan production software platform's hardening, perform periodic security audits and apply upgrades.

Sensitive Data Exposure: Sensitive data such as credit card numbers or passwords can be unintentionally exposed by applications.

Example: A website that is not using SSL/TLS has a login form for authentication. The credentials are sent in cleartext and can be easily intercepted.

Prevention: Always use encryption for sensitive data in transit. Encryption for data at rest can also be necessary when dealing it sensitive data such as passwords or credit card numbers.

Cross-Site Request Forgery (CSRF): A CSRF attack can happen when an attacker finds a reproducible link that executes a specific action on the server while the victim is logged in there.

The attacker might embed such link as image references or spam email links and trick the victim into opening it, forcing to execute specific actions without the user's consent.

Example: The attacker finds the following link that transfers funds of a user account when the user is correctly logged in

```
http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243
```

The attacker can embed the link in malicious sites or spam emails and trick the legitimate user to access those links while the victim keeps logged in. The malicious action will be executed without the user's consent:

```
<img
src="http://example.com/app/transferFunds?amount=1500&destinationAccount=attac
kersAccount#" width="0" height="0" />
```

Prevention: Use random unique IDs in hidden fields with each request or use CAPTCHA puzzles to reauthenticate users.

Using Components with Known Vulnerabilities: Application components such as libraries and frameworks might contain known vulnerabilities that are easily exploitable.

Example: Apache CXF webservice library and authentication bypass. Vulnerable Apache CXF library versions allowed the execution of any webservice if the authentication token was absent.

Prevention: Used components need to be identified. Their vulnerabilities must be periodically checked and updated.

Unvalidated Redirects and Forwards: Web applications usually redirect and forward users to other pages, and can use untrusted data to calculate the destination page. An attacker can redirect victims to malicious sites or use forwards to access unauthorized pages.

Example: An application has a "redirect" page that uses a URL as a parameter. An attacker could craft malicious URLs to redirect users to malicious sites:

```
http://www.example.com/redirect.jsp?url=evil.com
```

Prevention: Avoid redirects based on destination parameters and use server-side validation.

Underprotected APIs: The architecture of some web applications involves the creation of rich clients front-ends (e.g.: using JavaScript running on mobile phones or browsers) that are connected to remote back-end servers using APIs based on HTTP (e.g.: SOAP/XML, REST/JSON over HTTP, etc.). Those APIs are designed to be accessed by programs instead of humans, and they are often under protected.

Example: An API is designed to get requests from a mobile phone app using REST/JSON. An attacker can easily reverse engineer the mobile phone app and intercept the JSON data structures and manipulate them to send crafted requests with malicious intentions.

Prevention: API security should be comprehensively tested, making sure that communications are encrypted between client and server, strong authentication schemes are being used and that data format elements used in requests are parsed by hardened functions implementing protection against injection and unauthorized data and function references.

Summary

The following table provides a summary of the web application vulnerabilities mentioned in this chapter including some of their characteristics and associated attacks.

Table 2 Summary of web application vulnerabilities and related attacks

Vulnerability Type	Exploitability	Detectability	Impact	Attacks
Injection	Easy	Average	Severe	SQL Injection, XSS, LDAP Injection, XML Injection, Command Injection, HTTP Response Splitting
Broken Authentication and Session Management	Average	Average	Severe	Parameter Tampering, Authentication Bypass, Authorization Bypass, Session Hijacking, Session Fixation, Clickjacking
Cross Site Scripting (XSS)	Average	Average	Moderate	XSS
Broken Access Control (Insecure Direct Object References and Missing Function Level Access Control)	Easy	Easy	Moderate	Parameter Tampering, Authentication Bypass, Authorization Bypass
Security Misconfiguration	Easy	Easy	Moderate	(Any attack)
Sensitive Data Exposure	Difficult	Average	Severe	(Any attack)
Insufficient Attack Protection	Easy	Average	Moderate	(Any attack)
Cross-Site Request Forgery (CSRF)	Average	Easy	Moderate	CSRF, Clickjacking
Using Known Vulnerable Components	Average	Average	Moderate	(Any attack)
Underprotected APIs	Average	Average	Moderate	SQL Injection, XSS, LDAP Injection, XML Injection, Command Injection, HTTP Response Splitting

2.2. Intrusion Detection Systems (IDS)

The Intrusion Detection System concept was first mentioned by Denning in 1986 when he presented a novel intrusion detection model that could identify abnormal network behavior [14]. Despite the amount of years that have passed since then, intrusion detection still continues to be an important research topic and keeps evolving due to the continuous evolution of technology, networks and techniques used by attackers to break into systems.

An intrusion can be defined as "the set of actions that make attempts to challenge the integrity, discretion or accessibility of a resource" [14] or "any set of actions that attempt to compromise the integrity, confidentiality or availability of a resource" [15].

An Intrusion Detection System (IDS) can be defined as "the collection of tools, methods and resources to help identify, assess and report those intrusions" [15]. A more comprehensive definition of IDS is stated in [14] as "the collection of practices and mechanisms used to detect errors that may lead to security failure with the use of anomaly and misuse detection and diagnosing intrusions and attacks". Although IDSs don't prevent the malicious activity from happening, they are able to detect it. The detection of attacks is a key aspect because it can provide a way to analyze and study attacks, determine their sources and obtain enough information to help repair damaged systems or prevent future intrusions. Intrusion Prevention Systems (IPS) are similar systems with the additional ability to react whenever an intrusion is detected (e.g.: by blocking the request), thus preventing the intrusion.

Typically, IDSs have been classified by its location (host-based, network-based) and by the type of detection technique they employ (misuse/signature-based or anomaly-based) [14] [15]:

Host-based: They are based on individual computers and they are able to inspect features such as applications, users, login attempts, unusual memory allocations, file system activity or any other relevant resource at host level.

Network-based: They are located on the network and are able to collect and inspect traffic of the network segment they belong to.

Misuse-based detection: This type of detection consists of detecting patterns of already recognized attacks or known critical points in the system based on rule or signature databases. These techniques offer a high degree of accuracy but they lack the ability of detecting new unknown attacks.

Anomaly-based detection: Anomaly detection is based on using statistical measures on system features and comparing them against an established baseline that is considered to be "normal". Any significant deviation from the baseline could be considered as a possible attack. The main advantage of this type of detection is that it can detect previously unknown attacks. However, they can deliver a high rate of false positives and be computationally expensive. Most widely used techniques for anomaly detection fall under Data Mining and Machine Learning (ML) domains such as Bayesian networks, Markov models, fuzzy logic, genetic algorithms, Artificial Neural Networks (ANN), Principal Component Analysis (PCA) for dimensionality reduction, Support Vector Machines (SVM) matrices, decision trees and clustering algorithms [11] [12] [16] [17].

Intrusion Detection is a field that still faces many challenges. There isn't any detection technique that can universally be applied and current rule or signature based techniques are not able to cope with the amount of new attacks that are constantly appearing. The use of anomaly detection techniques is also challenging because they suffer from an excessively high rate of false positives, have performance problems [13] and the datasets used to test them (e.g.: DARPA and KDD) are considered by many to be outdated and not useful anymore as a reliable tool to

measure the validity of an IDS [17] [14]. Some researchers also state that anomaly detection is flawed in its basic assumptions [8] because machine learning techniques being employed are good to find similar events to ones previously seen, but they are not effective to find new unknown events that are not present in the training datasets.

Table 3 Summary of Intrusion Detection techniques

Detection Technique	**Location**	**Type of Attacks**	**Algorithms**	**Advantages**	**Disadvantages**
Misuse based detection	Host and Network	Known	Rule-based techniques, pattern matching	- High accuracy - Low false alarm rate - Computationally efficient	-Inability to detect new unknown attacks
Anomaly based detection	Host and Network	Unknown	Bayesian networks, Markov models, fuzzy logic, genetic algorithms, Artificial Neural Networks (ANN), Principal Component Analysis (PCA) for dimensionality reduction, Support Vector Machines (SVM) matrices, decision trees, clustering algorithms	- Ability to detect new unknown attacks	- High false alarm rate - Computationally expensive - Lack of suitable training datasets

2.3. Web Application Firewalls (WAF)

Web Application Firewalls are a specific type of intrusion detection and prevention system designed to protect web applications [1]. They are able to filter and monitor traffic on the application layer (OSI layer 7) and detect or block application-layer attacks such as SQL Injection or Cross Site Scripting. They also deal with common web server security misconfigurations and vulnerabilities that could be exploited by malicious attackers. An example of a popular open source WAF is ModSecurity [18].

2.4. Computer Deception

Computer Deception can be defined as "Planned actions taken to mislead attackers and to thereby cause them to take (or not take) specific actions that aid computer-security defenses" [6].

Deception provides an alternative approach to protection that focuses on hindering and deceiving adversaries, manipulating them and inducing them to take actions that are advantageous for computer security defense. Instead of focusing on attackers' actions by

detecting or blocking them, deception is focused on attackers' perceptions and it tries to mislead or confuse attackers for the benefit of the computer system defense.

Well-known examples of deception tools are honeypots, honey tokens and other honey-* techniques [6]. Honeypots are fake systems designed to look like real systems and whose aim is to attract adversaries in order to record their actions and analyze their attack goals and procedures. Honeytokens and other honey-* techniques (e.g.: honey accounts, honey files, honey words, etc.) refer to deception techniques that are implemented in the form of special fake files, fake user accounts, fake database records or fake passwords that are artificially created and are not meant to be accessed by legitimate users or applications. If those fake items were accessed, they would trigger an intrusion alarm.

Deception approaches have several advantages over traditional security controls because they can detect attacks and attack attempts based on unknown methods and they can also mislead or prevent attackers from succeeding regardless of the method being used (known or unknown). Consequently, deception techniques can be employed as an additional security layer of security control to detect or hinder attacks and protect systems.

One of the limitations of deception tools such as honeypots relies on the fact that they are implemented as repackaged isolated systems and can be fingerprinted and recognized by attackers using several tools [6]. In order to overcome this limitation, deception techniques can be integrated into real production systems; however, performing these integrations into existing systems must be done carefully not to interfere with the legitimate functions of the system [7]. The following diagram shows an example of a classification of different components of Computer Systems where deception could be integrated with.

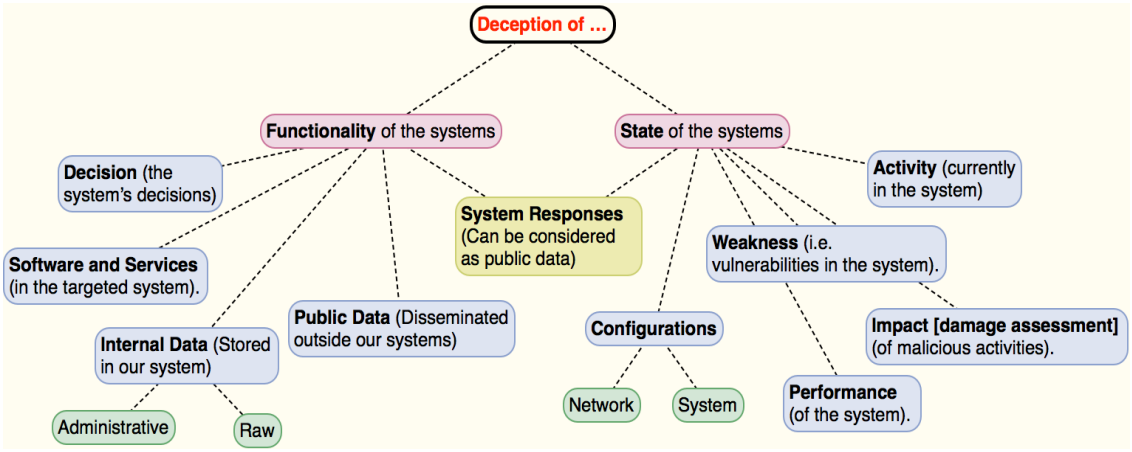


Figure 3 Integration of deception into computer system components [6]

The elements from the classification above could be applied to different layers of a computer system. If we consider the OSI (Open Systems Interconnection) reference model, deception could be integrated starting from layer 2 (data-link layer) and layer 3 (network layer) for example when redirecting attackers' traffic to controlled environments such as honeynets. Deception could also be employed at layer 4 (transport layer) for example by using fake port numbers and deceptive TCP handshake responses. Finally, deception could also be integrated at the upper layers (layer 5,6,7) when employing fake application data and responses.

CHAPTER THREE

3. LITERATURE REVIEW

A literature review has been conducted to reveal the current state-of-the art of the use of deception for web application security defense, aiming to grasp the central issues of the subject and summarize the most important concepts and current challenges.

The literature search has been carried out using the following databases and search engines that include top journals and conference papers: Scopus, EBSCOHost Academic Search Premier, ACM, arXiv.org, IEEE Xplore, ScienceDirect, SpringerLink, Web of Science and Google Scholar.

The range of years used for the search was 2000-2017 and the papers were selected by the number of citations, novelty and relevance to the subject by reading the titles, abstracts and some of the contents. Used keywords include combinations of: "Deception", "Cyberdeception", "Security", "Intrusion Detection", "Intrusion Prevention" "Intrusion Deception", "Attacks", "Web", "Web Applications", "HTTP", "Web Application Firewalls", "Active Defense", "Honeypots", "Honeytokens", "Decoy", "Automatic", "Generation", "Framework", "Model", "Web application vulnerabilities", "Web application scanners", "Web vulnerability scanners", "Penetration testing" and "Security testing".

Backward and forward searches based on references and authors have also been conducted in order to broaden the amount of literature to review [19].

The articles covered by this review are classified in two main areas: deception and its use for computer security defenses and web application security testing.

3.1. Deception and its use for computer security defense

The literature review shows that the use of deception for computer security is not a new concept [6] [20] [21]. Multiple references have been found in literature mentioning two works from the early 90s: "The cuckoo's egg" by Clifford Stoll [22] and "An Evening with Berferd" [23], in which authors explain how they interacted with attackers using fake or manipulated responses. These works influenced the first ideas of honeypots as fake computer systems created to deceive attackers and learn about their goals and tactics. In 1997 The Deception Toolkit (DTK) was released [21]. DTK was able to configure and simulate Unix-based deceptive services and it is considered as one of the first tools created to apply deception techniques for cyber defense. The "Honeytoken" term was coined later in 2003 by Spitzner [24] in order to characterize the use of the honeypot concept in any digital entity that was not a computer (e.g.: database records matching certain criteria, files, users, URLs, etc.).

One of the main concepts related with the use of deception is that deception must be believable for the attacker to be effective. Several works have addressed the problem of automatic generation of high quality believable data using different approaches.

In [20], authors present an automated honeytokens generator called HoneyGen. HoneyGen focuses on the creation of high-quality data honeytokens that are believable for an attacker and

indistinguishable from a real data entity. To accomplish this, the authors propose a three-step process: A rule mining phase (accessing the production database data in order to extract the structure, constraints and logic out of it), the honeytoken generation phase (using obfuscation of real data and also generating data from scratch based on the extracted rules from phase 1) and finally likelihood rating (scoring the similarity and ranking data tokens that have been generated from phase 2).

A system for generating and injecting indistinguishable network decoys is presented in [25].This paper also focuses into the generation of quality "believable" tokens in a network environment with the purpose of detecting silent passive attackers who are eavesdropping on enterprise networks. The system can generate different types of tokens (monitored passwords, credit card numbers, authentication cookies and documents containing beacons to alarm when opened). The system defines three phases (record, modify and replay) and provides templates to record and broadcast the decoys over the network using different protocols.

Another important concept of the use of deception is related with its modeling and planning. Multiple works underline the importance of the planning steps and provide theoretical deception models and frameworks.

One of the first models for using deception in computer defense was developed by Cohen et al. [26] after the creation of DTK, with the purpose of providing a general concept of human and computer deception. Game theory has also been used to model and study deception for computer security using honeypots [27] and honeynets [28]. Rowe [29] provided a model concentrating on how to apply "resource denial" responses effectively aiming to waste time and resources of adversaries while triggering alarms of potential attacks.

According to [6], most of the attempts to incorporate deception techniques to computer systems have been designed in an ad-hoc fashion. In order to overcome the limitations of ad-hoc approaches, they present a general framework that can be used to plan and integrate deception for computer security defenses. It is composed of different phases and eight steps that cover the planning, implementation and integration, and finally the monitoring and evaluating aspects of the use of deception for computer security defense.

The framework stablishes three main phases:

Phase 1. Planning Deception (Steps 1 -6). The planning of deception takes into account the goals, the expected behavior and reactions of the attacker, the elements that will be used for deception, the feedback channels and the risks that the strategy might introduce.

Phase 2. Implementing and Integrating Deception (Step 7). This step integrates the planned strategies into the system.

Phase 3. Monitoring and Evaluating Deception (Step 8). This step has to do with the monitoring of the deception using the specified feedback channels. The results obtained must be evaluated to check if the expectations have been met. If the deception strategies does not obtain the expected results, the process can begin again by setting a new goal or by improving and repeating steps 1 to 7 again.

The following picture summarizes the steps and the process proposed by the Framework to Incorporate Deception in Computer Security Defenses:

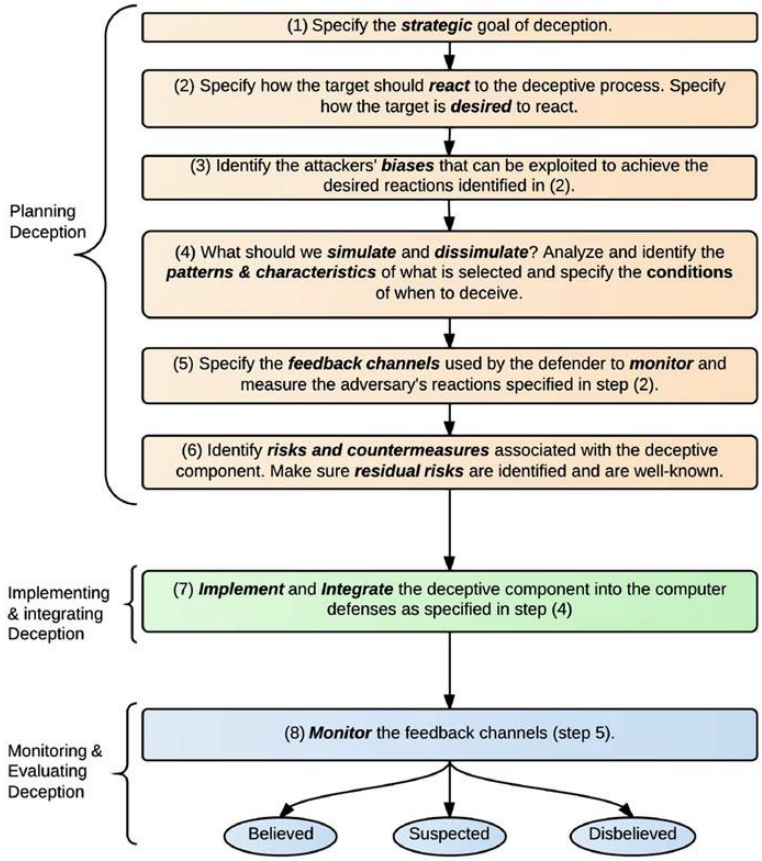


Figure 4. Framework to Incorporate Deception in Computer Security Defenses [6]

These are the details about each one of the steps that the framework proposes:

Step 1. Specify the strategic goals of deception. The goal of deception-based mechanisms must always be comprehensively detailed. Just installing honeypots or putting honeytoken elements on the system without detailing the exact goals of deception can give a false sense of deceiving adversaries.

Step 2. **Specify how the target should react.** The expected reaction of the attacker must be considered, as well as how we desire him or her to react, in order to influence his perception.

Step 3. **Understand the attacker's biases.** Successful deception should be designed to exploit precise adversaries' biases. Biases can be divided in four groups: personal biases, cultural biases, organizational biases and cognitive biases.

Step 4. **Create the deception story.** Decide what components to simulate/dissimulate (functionality or state of the system) and what techniques to apply (masking, repackaging, dazzling, mimicking, inventing or decoying).

Step 5. **Specify feedback channels.** Deception cannot be considered as a one-time defense mechanism and it is fundamental to monitor adversaries by using feedback channels to be able to measure their perceptions and actions.

Step 6. **Identify risks that deception may introduce.** Potential risks associated with the introduction of deception must be identified and accepted (e.g.: effects on normal user's activities).

Step 7. **Integration and implementation.** Separate isolated systems (e.g.: honeypots) can be fingerprinted and detected by attackers, therefore, successful deception must be integrated into real production systems and operations.

Step 8. **Monitoring and evaluation.** Deception must be monitored using the feedback channels identified in step 5. The information obtained from the feedback channels must be analyzed and evaluated in order to review the attackers' biases of step 3 and iteratively improve the design of the deception strategy over time.

The importance of planning and integrating deception for computer security is also discussed in [21]. This work also underlines the importance of the planning steps and states that using ad-hoc approaches usually result in developing single tools or repackaging solutions (e.g.: honeypots) that lead to poor results. They present a model that share similarities with the framework provided by [6].

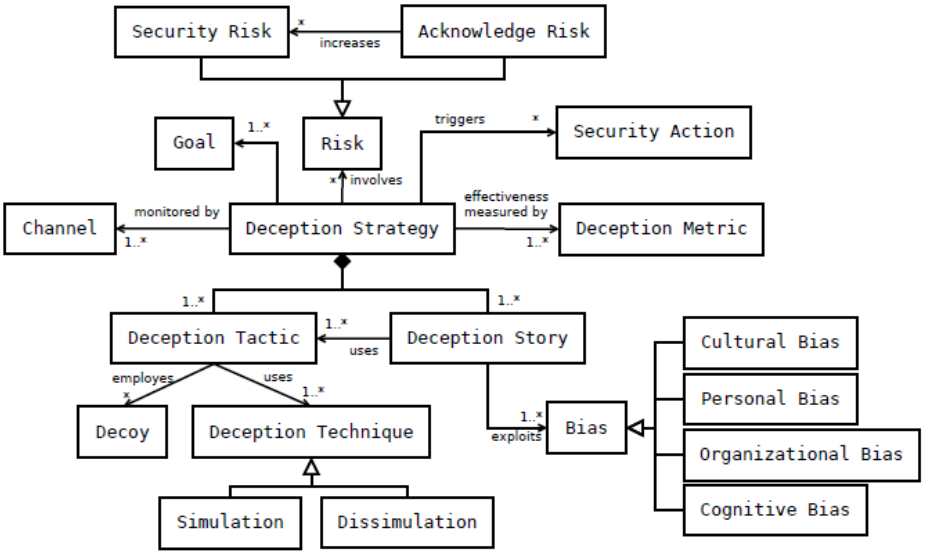


Figure 5. Deception Strategy Model [21]

The specific use of deception for web application security defense is addressed by some articles from a conceptual level.

In [30], authors present a framework for Intrusion Deception on web servers. This paper proposes a framework that underlines the importance of the planning steps that should be considered when designing any deception-based intrusion detection system and it also proposes a theoretical high-level architecture of components that should take part in an intrusion deception system, composed of intrusion detection systems, decoys, virtual honeypots and a deceptive content generator that could be used in the context of a web server. This work only names the high-level architecture of the elements and does not mention the types or ways in which decoys will be created or how deceptive content generator components will generate and integrate deceptive data.

In [31], a centralized deceptive server is designed to be hooked to multiple public-facing servers of an organization (e.g.: web servers, FTP servers, etc.) is presented. This server aims to create on-the-fly deceptive responses to suspicious clients, based on a centralized database of deception generation rules. This work focuses on the concept of having a centralized element for deceptive data generation and does not detail the actual rules for the generation of those deceptive elements.

The literature search that has been conducted hasn't been able to find a variety of articles that tackle the concept of the actual design of deception strategies that are integrated into application-layer protocols aiming to detect or prevent web application attacks. Only one article has been found [32], where authors use a web application deception proxy to detect and prevent parameter manipulation attacks. This recently published paper from 2017 proposes the use of deception to increase the economic cost of attacks based on parameter manipulation in web applications. The authors present an example of a web application proxy with the ability to detect and prevent parameter tampering attacks using two examples of parameter injection and parameter obfuscation mechanisms. The effectiveness of these techniques is supported by building a vulnerable web application, launching automated attacks and verifying that the injection of additional and obfuscated parameters makes the penetration tools spend more time in performing their scans.

3.2. Web application security testing

The experimentation platform that needs to be designed during this research will be built using vulnerable web applications and web application scanners or penetration testing tools.

Web application security testing is an active research field and there are multiple works available related with the analyses, comparatives or categorizations of web application security scanning and penetration testing tools [33] [34] [35] [36] [37] [38] that are tested using web applications with known vulnerabilities. The aim of the review has been to select the articles that offered the best representative coverage of this specific topic.

Vulnerable web applications are designed on purpose with a number of exploitable vulnerabilities that are well documented. These vulnerable applications can be used to test the performance of vulnerability scanner tools by comparing the actual number of documented vulnerabilities of the applications against the number of vulnerabilities that have been detected by the tools.

In [34], OWASP ZAP and Skipfish scanners are compared against DVWA and WAVSEP, concluding that OWASP ZAP has a higher accuracy for detecting vulnerabilities and a low rate of false positives. In [39] a comparison of 32 free analysis tools is conducted using the WASSEC [40] evaluation criteria. The study concludes that W3AF, OWASP ZAP, Arachni and IronWASP are the best tools. The same author also releases a similar study in [41] comparing 13 commercial web application scanners using the same WASSEC evaluation criteria and stating that Acunetix Web Vulnerability Scanner is the best performing commercial tool.

[35] compares Nessus, Acunetix and OWASP ZAP scanners against two custom developed web applications stating that Acunetix is the best vulnerability scanning tool for web applications.

[36] evaluates six free and open source Web Vulnerability Scanners: NetSparker, N-Stalker Free, OWASP ZAP, W3AF, Iron WASP, and Vega Vulnerability Scanner. They are tested against the WackoPicko vulnerable web application. IronWASP is the best scoring tool with the highest number of found vulnerabilities and lowest rate of false negatives.

[37] evaluates Acunetix, Netsparker and Burp Suite against a custom vulnerable web application. It scores Acunetix as the best tool for all evaluated categories (except reporting of security misconfiguration vulnerabilities).

[38] analyzes the traffic generated by different penetration testing tools (Acunetix, OWASP ZAP, HP WebInspect and Arachni Scanner) against vulnerable web applications (DWVA and WackoPicko), combining it with an IDS (Snort) in order to find out which vulnerabilities the tools are trying to detect. Arachni is one of the best performing tools according to this study.

The following table shows a summary of web application vulnerability scanners that are mentioned in literature:

Table 4 Vulnerability scanners

Vulnerability Scanner Tool	Free / Open source	Referenced by
Acunetix [42]	NO	[41] [35] [37] [38]
Arachni [43]	YES	[39] [38]
Burp Suite [44]	YES	[37] [41]
Google Skipfish [45]	YES	[34] [39]
HP WebInspect [46]	NO	[41] [38]
Iron WASP [47]	YES	[39] [36]
Nessus [48]	NO	[41] [35]
Netsparker [49]	NO	[41] [36] [37]
N-Stalker Free [50]	YES	[36]
OWASP ZAP [51]	YES	[34] [39] [35] [36] [38]
Vega Vulnerability Scanner [52]	YES	[39] [36]
W3AF [53]	YES	[39] [36]
Sqlmap, Watobo, Andiparos, ProxyStrikev, Wapiti, Paros Proxy, Grendel Scan, PowerFuzzer, Oedipus, UWSS, Grabber, WebSacarb, MiniSqlat0r, WSTool, Crawlfish, Gamja, iScan, DSSS, Secubat, SQID, SQLiX, Xcobra, XSSploit, XSSS, XSSer, aidSQL	YES	[39]
Ammonite, IBM AppScan, JSky, NTOSpider, ParosPro, QualysGuard WAS, Syhunt Dynamic, WebCruiser Enterprise Edition	NO	[41]

The following table shows a summary of all publicly available vulnerable web applications that have been found in literature:

Table 5 Vulnerable web applications

Vulnerable web application	Free / Open source	Referenced by
Damn Vulnerable Web Application (DVWA) [54]	YES	[34] [38]
WackoPicko [55]	YES	[36] [38]
The Web Application Vulnerability Scanner Evaluation Project (WAVSEP) [56]	YES	[34]
Btslab, BadStore, BodgeIt Store, Bricks Butterfly Security Project, bWAPP, Cyclone Transfers, Damn Vulnerable Node Application – DVNA, Damn Vulnerable Web Application – DVWA, Damn Vulnerable Web Service – DVWS, Damn Vulnerable Web Services – DVWS, Damn Vulnerable Thick Client App – DVTA, Gruyere Hackademic Challenges Project, Hackazon Hacme Bank – Android, Hacme Bank, Hacme Books, Hacme Casino, Hacme Shipping, Hacme Travel, hackxor, Juice Shop, LampSecurity, Mutillidae, .NET Goat, NodeGoat, Peruggia, Puzzlemall, Rails Goat, SecuriBench, SecuriBench Micro, Security Shepherd, SQL injection test environment, SQLI-labs, VulnApp, Vulnerable Web App, WackoPicko, WAVSEP - Web Application Vulnerability Scanner Evaluation Project, WebGoat, WebGoatPHP, WIVET - Web Input Vector Extractor Teaser, Xtreme Vulnerable Web Application (XVWA)	YES	[57]

3.3. Research gap

The use of deception for computer security defense has been studied for a long time since the popularization of honeypots in the early 90s. Several works have studied numerous aspects of deception, mostly focusing on the automatic generation of quality "believable" data tokens and creating models or frameworks for the use of deception.

There are works mentioning the use of deception in the context of web servers providing theoretical architectures of components and presenting platforms with a centralized server to provide deceptive data to multiple servers of an organization (including web servers). However, those works do not state how the actual deception strategies are created and applied; hence, they do not detail the actual deception stories and rules for the generation of deceptive elements in the context of web application-layer traffic.

[32] is the only work that presents an example of an actual deception strategy integrated into the application-layer of web applications, employing two mechanisms based on injection of additional and obfuscated parameters into web application traffic aiming to detect and hinder parameter tampering attacks. The effectiveness of these techniques is supported by implementing an HTTP proxy, using a vulnerable web application and launching attacks with automated penetration testing tools.

In [32], authors state that deception hasn't been employed in the application-layer and they mention it is a field where new techniques could be developed in the future. The existence of [32] is helpful to confirm the validity and relevance of the research problem that this thesis aims to address. The way and the approach to conduct this thesis shares similarities with the mentioned work. Firstly, because they design a deception strategy for the application-layer and implement it into a proxy that aims to detect and hinder web application attacks, and secondly because they use a vulnerable web application and penetration testing tools to test their deception proxy.

However, the authors of [32] do not rely on any theoretical deception model or framework for deception, thus they lack a careful planning of the goals of their strategies. Their work is also limited in the sense that they only take parameter tampering attacks into account.

Taking everything into consideration, this literature review reveals that there is a lack of works addressing the specific use of deception strategies that are integrated into the application-layer protocol aiming to detect or prevent web application attacks. The work of [32] confirms the interest and validity of the problem but there is still a gap to fill when it comes to the development of new additional strategies for diverse types of web application attacks. The importance of carefully planning deception is underlined by several authors and some deception models and frameworks have been proposed in literature. Nevertheless, no works have been found that are actually applying those theoretical models and frameworks to develop deceptive strategies. This thesis will follow the Framework to Incorporate Deception in Computer Security Defenses [6] as a model to guide the design of deception strategies in a rigorous way.

CHAPTER FOUR

4. RESEARCH METHODOLOGY

Design Science Research (DSR) is the selected methodology to conduct this work. This methodology has been chosen because it is adequate to develop and evaluate solutions to problems that involve the creation of new artifacts, that is, things that are artificial or constructed by humans. The two fundamental questions that DSR addresses are: "what utility does the new artifact provide?" and "what demonstrates that utility"? [58].

DSR can be summarized in six steps: problem identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation, and communication [59].

The following picture shows a summary of the DSR steps and the sequence of the process.

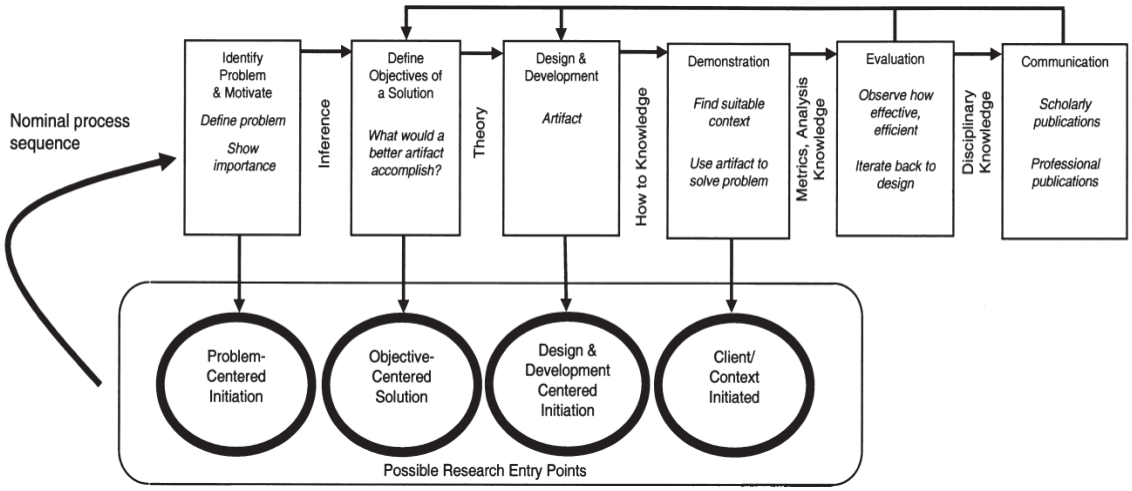


Figure 6 Design Science Research Process [59].

1. **Problem identification and motivation**: This step defines a specific research problem and justifies the value of a solution.
2. **Definition of the objectives for a solution**: This step infers the objectives for a solution from the previous step.
3. **Design and development**: This step takes the creation of the artifacts into account.
4. **Demonstration**: This step demonstrates that the artifacts can solve one or more instances of the problem.
5. **Evaluation**: This step observes and measures how well the artifacts support the solution to the problem and it is one of the key steps of DSR. If the evaluation step does not provide a satisfactory result, it is necessary to go back to activity 3 to refine and improve the design until a satisfactory solution is found. The evaluation step can be carried out using observational, analytical, experimental, testing or descriptive methods [58].
6. **Communication**: This step has to do with the communication of the problem and the motivation of the research, the artifact designed, its utility, novelty, the rigor of its design and its effectiveness to researchers or other interested audiences.

DSR defines four main types of artifacts: constructs, models, methods and instantiations [59] [58]. The main research question of this work consists on the design of the deception strategies and they can be described as a "method" type of artifact. The secondary research question involves the creation of an additional "instantiation" type of artifact to create a testing environment. This environment will contain the implementation of the strategies in a software artifact needed to conduct experiments.

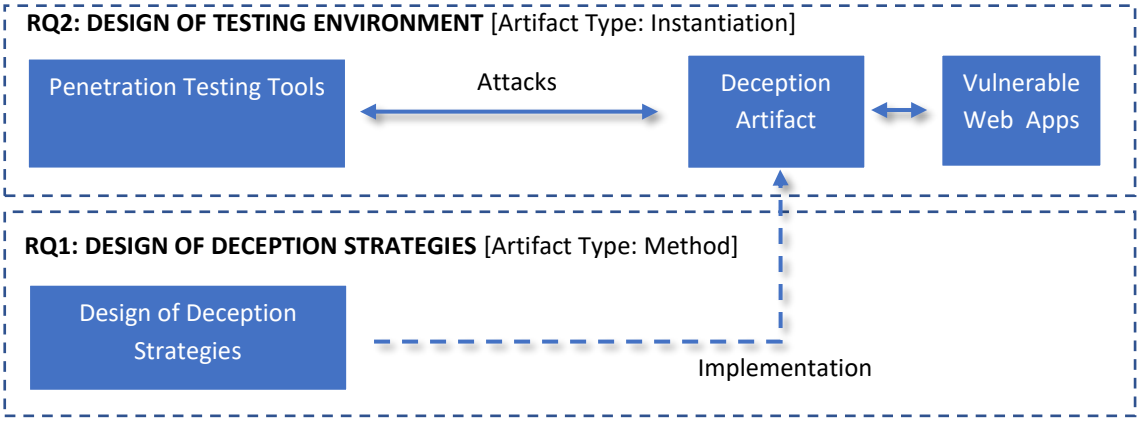


Figure 7 Research questions and testing environment architecture

This is how the DSR steps will be mapped for each one of our two research questions:

RQ1. What is the performance of different deception strategies in detecting or preventing web application attacks?

Problem identification and motivation: The problem is identified with the design of application-layer deception strategies and their performance to secure web applications. The motivation is driven by the high amount and high impact of web application attacks that are currently taking place and the need of creating new alternative protection approaches to complement current techniques (e.g.: IDS, IPS, WAF, etc.).

Definition of the objectives for a solution: The objectives of the deception strategies are to detect or prevent application-layer attacks targeting web applications.

Design and development: Deception strategies will be designed following the Framework to Incorporate Deception in Computer Security Defenses [6].

Demonstration and evaluation: Demonstration and evaluation will be performed using the experimental method, executing "design experiments" [60]. The difference between experimentation and pure observation is that experimentation requires the researcher to actively manipulate and control the causes of the phenomenon. To accomplish this, the experiments need to be properly planned by setting up reproducible experiments, identifying dependent and independent variables, evaluating the effects that changes of independent variables have on the dependent variables by predefining certain metrics, and finally analyzing the empirical results obtained.

Experiments will be set up launching attacks with web application scanners or penetration testing tools (e.g.: OWASP ZAP) against vulnerable web applications enabling and disabling an instantiation of the deception artifact.

The independent variables that will be controlled in our context are: testing environment system architecture, deception strategy parameters, software versions and the configuration values of the penetration testing tools. The dependent variables to be analyzed will be the number of attacks that can be detected or prevented by the deception artifact, the number of vulnerabilities found by the attacking tools and the time needed to perform those attacks. The instantiation of the testing environment is covered in RQ2.

The criteria for evaluation will be based on the ability of the artifact to detect or prevent attacks originated by the penetration testing tools. The performance will be measured by the number of vulnerabilities found by the attacking tools, the alerts generated by the deception artifact and the time needed by the penetration testing tools to perform their attacks.

The evaluation criteria do not predefine any fixed performance result requirements that the strategies must meet in order to consider them satisfactory. The designed strategies will be considered valid as long as they are able to detect or prevent at least one kind of attack each. The design process is expected to include several iterations that will aim to improve and refine the design of the strategies over multiple evaluation and redesign cycles.

RQ2. How can a testing environment be created in order to test the suitability of the deception strategies?

An "instantiation" type of artifact will be used to create a testing environment to test the deception strategies.

Problem identification and motivation: The identified problem is the design and implementation of a testing environment. The need to complete the evaluation phase of the method artifact covered in RQ1 is the motivation of this problem.

Definition of the objectives for a solution: The testing environment must be able to execute reproducible tests with controlled variables to apply deception strategies to real HTTP traffic against a vulnerable web application.

Design and development: The testing environment will contain a software implementation of the deception strategies that will be hooked to an existing vulnerable web application. A set of penetration testing tools will be used to launch attacks.

Demonstration and evaluation: Demonstration and evaluation will be performed based on functional testing. Different requirements will be defined for each component that takes part in the platform. Tests will be executed trying to discover failures and defects on the platform and its components.

The criteria for evaluation will be based on the ability of the artifact to apply all the use cases of the designed deception strategies into real HTTP traffic and will be evaluated executing functional test cases for each strategy implementation. The implemented strategies will have to trigger the expected alarms when the requests match the criteria stablished by each strategy.

Operational aspects such as CPU and memory use will also be measured. The evaluation criteria do not predefine any minimum or maximum operational result requirements that the implementation must meet, but the goal will be to get the minimum CPU and memory overhead possible. The implementation process is expected to include several iterations that will aim to improve and minimize operational overhead aspects.

CHAPTER FIVE

5. DESIGN OF DECEPTION STRATEGIES FOR HTTP

The design of the deception strategies is composed of five independent strategies. Each one of the strategies takes distinct types of attacks into account and uses different elements of the HTTP protocol to apply deception.

The aim has been to produce as many strategies as possible using different elements of the HTTP protocol that could have the possibility to include deceptive elements into it. The strategies have been created from scratch and in the end five of them have been shaped, which are summarized in the table below.

Table 6. List of deception strategies

Deception Strategy		
	Name	Description
1	Deceptive Comments	Generates HTML comments with deceptive content
2	Deceptive Parameters	Adds fake parameters into existing application forms and parametrized links
3	Deceptive Cookies	Injects fake session cookies simulating predictable session identification patterns
4	Deceptive Status Codes	Modifies HTTP error 404 status codes with HTTP 200 OK codes when a certain threshold is passed
5	Deceptive JavaScript	Injects fake JavaScript functions with deceptive variables and URLs

The design of the strategies has been guided using the framework to incorporate deception into computer security defense [6]. The design has been guided following the steps detailed in the selected framework, which are summarized in the following figure.

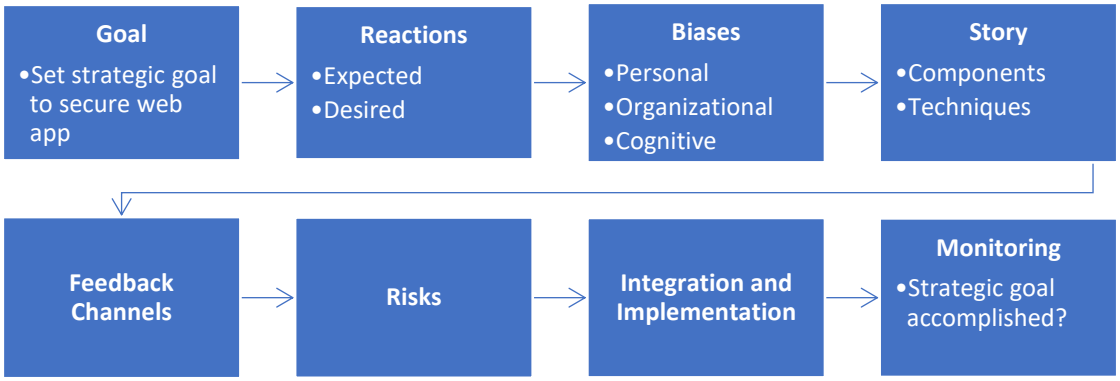


Figure 8 Strategy design procedure

For each one of the strategies, a table has been created containing sections that correspond to the guidelines established by the design framework, aiming to include all the information that is necessary for the design.

5.1. Deceptive Comments

The aim of this strategy is to detect intrusion attempts by injecting deceptive content inside HTML comments that can be attractive for an attacker (e.g.: fake secret links or fake leaked credentials). The main advantage of this strategy lies on its simplicity and on the lack of risks to interfere with the regular use of the application by legitimate users.

Table 7. Deception Strategy 1. Deceptive Comments

<table>
<tr><th colspan="3">Deception Strategy 1: Deceptive Comments</th></tr>
<tr><td colspan="3">Strategic goal</td></tr>
<tr><td colspan="3">The goal of this strategy is the detection of intrusion attempts by an attacker and the monitoring of its activities and intentions. The goal will be accomplished by injecting fake links inside HTML comments into the HTML output of server responses.</td></tr>
<tr><td colspan="3">Attacker reactions</td></tr>
<tr><td colspan="2">How the attacker should react</td><td>How the attacker is desired to react</td></tr>
<tr><td colspan="2">The attacker should believe that a fake link is genuine and has accidentally been leaked inside HTML comments. The attacker should believe that the link can be used to access high privileged hidden content and should send a request to it.</td><td>The attacker should be monitored as long as possible. In order to accomplish that, two scenarios are considered:
• Continue: Requests to fake links will return a valid HTTP response containing a new different fake link to keep the attacker interested and interacting with the system while being monitored.
• Block: When a request to a fake link is detected, the attacker will be flagged and the original response will be blocked. Instead, a special page will be returned simulating that the website is down for maintenance for a given period. The attacker is expected to stop the attacks in that moment and try to reconnect after the specified period.</td></tr>
<tr><td colspan="3">Attackers' biases</td></tr>
<tr><td>Personal</td><td>Organizational & Cultural</td><td>Cognitive</td></tr>
<tr><td>Not considered because there is no information about the attacker's personal background.</td><td>Not considered because there is no information about the attacker's</td><td>Conjunction fallacy bias will be exploited. Fake HTML comments and responses will contain explicit details that will be associated among</td></tr>
</table>

	organizational/cultural background	them, as humans are usually more prone to believe detailed stories or events that are composed of several disjoint components rather than single or compact ones [6]. In the case of the fake page simulating that the system is down for maintenance, an explanatory message and the expected system recovery date and time will be shown to the user.

Conditions and characteristics of deception

Deception will be integrated in system responses by manufacturing new HTML content containing fake links that will act as decoys. Access to those links will mimic valid content and incorporate additional new fake links.

Feedback channels

The monitoring of the URL of HTTP requests will constitute the feedback channel of this strategy.

Risks and countermeasures

This strategy has no associated risks because deceptive content is injected inside HTML comments that are ignored by web browsers, causing no interferences with the normal flow of the application.

Implementation and integration

The deception strategy will be integrated into the application-layer HTTP traffic. The artifact implementing the strategy will have to complete the following steps:

1. Intercept HTTP responses of the original application and inject a fake HTML comment before delivering the response to the client. The comment will contain an attractive phrase followed by a newly generated random URL. e.g.: `{//TODO: Remove this link for production. Autolink, //Test link for direct backend access, //remove this!, //devel only}`
2. Intercept HTTP requests and check if the request is targeting a fake URL that has been injected inside comments in a previous request. If a request to a fake URL is detected, the system will have to:
3. Trigger an intrusion detection alarm, increase the alert counter and log the request details to a log file.
4. Return a response depending on the configured mode:
    a. Continue mode: Return a valid HTTP response with 200 status code containing at least a new additional fake URL that the attacker could access.
    b. Block mode: Return a valid HTTP response providing a special response page stating that the system is down for maintenance and giving some specific detailed information

	about the recovery time. The blocking mode flag will be automatically reset after a configurable period (default period=5 minutes).

Example

[Client → Server] Initial request from client to server:

HTTP GET http://localhost:8080/BodgeIt

[Client ← Server] Original response intercepted by the deception artifact

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
<title>The BodgeIt Store</title>
<link href="style.css" rel="stylesheet" type="text/css">
…
```

[Client ← *Server] Response modified by the deception artifact and delivered to the client

A fake HTML comment will be injected containing a new random URL

```
<!-- TODO: Remove for production. Autolink:<a
href="1r18r7m9q73cm0hfchkc2o8ns3"></a>/ -->
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
<title>The BodgeIt Store</title>
<link href="style.css" rel="stylesheet" type="text/css">
…
```

[Client → Server] Request from client to server:

If the client accesses the generated URL inside the comments, the deception artifact will intercept it:

HTTP GET http://localhost:8080/BodgeIt/1r18r7m9q73cm0hfchkc2o8ns3

The deceptive comments intrusion counter will be incremented in one unit and the request details will be logged. The server will respond with a HTTP 200 status code and the body of the response will contain a new deceptive comment. If the blocking mode is enabled, a special response page will be returned stating that the system is down for maintenance, including specific detailed information about the recovery time. The blocking mode flag will be automatically reset after a configurable time (default period=5 minutes).

5.2. Deceptive Request Parameters

Parameter tampering attacks attempt to maliciously manipulate HTTP request parameters aiming to change the original behavior of the application and bypass its legitimate logic. This strategy aims to detect parameter tampering attempts by injecting fake parameters and monitoring their changes.

Table 8. Deception Strategy 2. Deceptive Request Parameters

Deception Strategy 2: Deceptive Request Parameters		
Strategic goal		
The main goal of this strategy is the detection of parameter tampering attempts by an attacker and the monitoring of its activities and intentions. The goal will be accomplished by injecting fake parameters into existing HTTP application-layer traffic and reacting whenever they suffer any modification.		
Attacker reactions		
How the attacker should react	**How the attacker is desired to react**	
The attacker should believe that fake parameters are genuine and really belong to the real application and should try to modify them in a malicious way.	The attacker should be monitored as long as possible. In order to accomplish that, changes into fake parameters will seem to have an effect in the HTTP response, aiming to keep the attacker interested and interacting with the system while being monitored. Two scenarios are considered: • **Continue**: Requests with modified fake parameters will trigger an intrusion alarm but the original server response will continue to be returned to the user. A comment with random information will be injected into the response in order to keep the attacker interested and interacting with the system while being monitored. • **Block**: Requests with modified fake parameters will trigger an intrusion alarm but instead of returning the original server response, a special page will be returned simulating that the website is down for maintenance for a limited period. The attacker is expected to stop the parameter tampering attacks in that moment and try to reconnect after the specified period.	
Attackers' biases		
Personal	**Organizational & Cultural**	**Cognitive**
Not considered because there is no information about the attacker's personal background.	Not considered because there is no information about the attacker's	Conjunction fallacy bias will be exploited. Fake parameters and responses will contain explicit details that will be associated

	organizational/cultural background	among them, as humans are usually more prone to believe detailed stories or events that are composed of several disjoint components rather than single or compact ones [6]. In the case of the fake page simulating that the system is down for maintenance, the expected system recovery date and time will be shown to the user.

Conditions and characteristics of deception

Deception will be integrated in system responses by manufacturing new parameters that will act as decoys and mimicking the effects of changes on those parameters.

Feedback channels

The monitoring of the HTTP request parameters of the attacker will constitute the feedback channel of this strategy.

Risks and countermeasures

Fabricated deceptive parameters must always be distinct from the real parameters that the legitimate application uses for its real use cases, not to cause interferences with the normal flow of the application.

Implementation and integration

The deception strategy will be integrated into the application-layer HTTP traffic. The artifact implementing the strategy will have to complete the following steps:

1. Intercept HTTP responses of the original application and inject fake parameters before delivering the response to the client. The possible fake parameter values will try to have attractive names and values. The candidate list will be composed of `{"admin=false", "development=false", "authorization=no", "fullView=no", "query=none"}`.
    a. For GET requests, the parameters will be added to existing `<a href>` link values.
    b. For POST requests, the parameters will be added as hidden input type parameters of existing forms.
2. Intercept HTTP requests and check if the injected parameters from the previous steps have been modified. If a modification is detected, the system will have to:
3. Trigger an intrusion detection alarm, increase the alert counter and log the request details to a log file.
4. Return a response depending on the configured mode:
    a. Continue mode: Modify the HTTP response injecting a comment with random content and changing the fake parameters to a different value.

b. Block mode: Return a HTTP response providing a special response page stating that the system is down for maintenance and giving specific detailed information about the recovery time. The blocking mode flag will be automatically reset after a configurable time (default period=5 minutes).

Example

Fake parameter candidates:

```
{admin=false, development=false, authorization=no, fullView=no, query=none}
```

[Client → Server] Initial request from client to server:

HTTP GET http://localhost:8080/BodgeIt/basket.jsp

[Client ← Server] Original response intercepted by the deception artifact

```
…
<a href="product.jsp?typeid=2">Product</a>

…
<form action="basket.jsp" method="post">
        <input type="hidden" name="productid" value="6">
        <input type="hidden" name="price" value="3.3">
        <input type="submit" id="submit" value="Add to Basket">
…

</form>
```

[Client ← *Server] Response modified by the deception artifact and delivered to the client

A random parameter will be picked from the candidate list. e.g.: admin=false

```
…
<a href="product.jsp?typeid=2&admin=false">Product</a>

…
<form action="basket.jsp" method="post">
        <input type="hidden" name="productid" value="6">
        <input type="hidden" name="price" value="3.3">
        <input type="hidden" name="admin" value="false">
        <input type="submit" id="submit" value="Add to Basket">
…

</form>
```

[Client → Server] Request from client to server:

If the client modifies the admin parameter via an HTTP GET request or submitting the HTTP POST form, the deception artifact will intercept it:

HTTP GET http://localhost:8080/BodgeIt/product.jsp?typeid=2&admin=true

The parameter tampering intrusion counter will be incremented in one unit and the request details will be logged. A comment with random information will be injected into the response in order to keep the attacker and the name of the fake parameter will be modified with the next candidate. If the blocking

mode is enabled, a special response page will be returned stating that the system is down for maintenance, including specific detailed information about the recovery time. The blocking mode flag will be automatically reset after a configurable time (default period=5 minutes).

5.3. Deceptive Session Cookies

Many common session management attacking methods are based on the manipulation or theft of HTTP session identification tokens that can maliciously be employed to bypass the authentication mechanisms of a web application. These tokens are usually set using HTTP cookies, which are essentially small pieces of data sent by web servers and stored on client computers or web browsers.

Session ID cookies can be compromised in diverse ways. One approach is to eavesdrop the content of cookies if they are being transmitted using a non-secure transport (HTTP instead of HTTPS) or obtaining them running malicious scripts on victims' browsers that can read cookie data and transmit it towards an attacker.

Another approach used by attackers is based on the analysis and understanding of the session ID generation process of web applications. If an attacker can understand the session ID generation process, it will be feasible to manipulate, predict or fabricate valid session IDs to hijack existing sessions and bypass the authentication process of an application.

The design of the following deception strategy is focused on those attacks where attackers try to understand and reproduce the session ID creation process of a web application. The deceptive strategy will mimic weak session ID creation patterns by the server, aiming to draw the attention of the attacker and persuade him or her to perform session id manipulation attacks that will be detected and monitored.

Table 9. Deception Strategy 3. Deceptive Session Cookies

Deception Strategy 3: Deceptive Session Cookies	
Strategic goal	
The goal of this strategy is to detect session manipulation attempts by an attacker and to know about the level of skills that the attacker has. This goal will be accomplished by injecting fake session ID cookies into existing HTTP application-layer traffic and reacting whenever they suffer any modification.	
Attacker reactions	
How the attacker should react	How the attacker is desired to react
The attacker should believe that fake session id cookies are genuine and think they have been created using a certain pattern that can be forged or manipulated. The attacker should be tempted to modify those cookies in a malicious	Two scenarios are considered: • Continue: Requests with manipulated cookies will trigger an intrusion alert but the requests will continue towards the application and will return the corresponding original response and

way in order to get access to restricted resources or impersonate existing valid session ids.	a random content inside comments in order to keep the attacker interested as long as possible. • **Block**: If a cookie manipulation attempt is detected, the attacker will be flagged and the original response will be blocked. Instead, a special page will be returned simulating that the website is down for maintenance for a given period. The attacker should give up the attempts because of not being able to connect to the web application in that moment and should try to reconnect after the specified period.

Attackers' biases		
Personal	Organizational & Cultural	Cognitive
Not considered because there is no information about the attacker's personal background.	Not considered because there is no information about the attacker's organizational/cultural background	Conjunction fallacy bias will be exploited [6]. Fake session cookies will be created using patterns that could be guessed by the attacker and will contain different explicit details about session numbers or authorization levels.

Conditions and characteristics of deception
Deception will be integrated in system responses by manufacturing new session id cookies that will act as decoys. Manipulation of those session ids will mimic the effects of changes on the behavior of the system.
Feedback channels
The monitoring of the HTTP request headers of the attacker will constitute the feedback channel of this strategy.
Risks and countermeasures
Fabricated deceptive cookies must always be distinct from the real cookies that the legitimate application uses for its real use cases, not to cause interferences with the normal flow of the application.
Implementation and integration
The deception strategy will be integrated into the application-layer HTTP traffic. The artifact implementing the strategy will have to complete the following steps:

1. Intercept server HTTP response headers of the original application traffic. If a Set-Cookie header is found, fake session id cookies will be injected before delivering the response to the client. The association between the fake cookie and the original cookie needs to be stored.

   Fake cookie identifiers must have attractive names related with sessions and identifiers. The candidate cookie name list will be: `{"JESSIONID", "SID", "ID", "SESSID", "AUTHID", "AUTHLEVEL", "JSERVERSESSION", "SESSIONID", "CFID", "CFTOKEN", "SIDTOKEN"}`.

   Values of fake cookies must follow a certain pattern that will be guessable by the attacker. The difficulty of guessing the pattern will be configurable in three levels. The values will consist on integer values ranging from 0000000000000000 to 9999999999999999 that will be incremented in one unit with every new stablished session. Depending on the difficulty level that wants to be applied, the values will be shown in a different format:

   a. Easy. Simple integer session ids (e.g.: The session id integer will be shown directly. `SID=0000000000000001`)
   b. Medium. Base64 obfuscated values (e.g.: The Base64 encoding function will be applied. Base64(0000000000000001)=M4M4M4M5. `SID=M4M4M4M5`).
   c. Hard. MD5 hashed values (e.g.: The MD5 hash function will be applied. MD5(0000000000000001)=0e7799e0a7b89ff4dda7fbf6a125f60b. `SID=0e7799e0a7b89ff4dda7fbf6a125f60b`).

   Base64 obfuscation can easily be identified by the attacker and can be decoded with a minimum effort. MD5 hash values can also be easily reverse engineered using available online dictionaries that contain common MD5 values such as those belonging to integer sequences and common strings. Using a higher difficulty level will likely make the session ids more believable for the attacker and will allow classifying attackers as having more skills and more advanced attacking powers.

2. Intercept client HTTP request headers and check if the session id cookie has an associated fake cookie. If there is an association and the fake cookie value has been modified, the system will have to:
3. Trigger an intrusion detection alarm, increase the alert counter and log the request details to a log file.
4. Return a response depending on the configured mode:
   a. Continue mode: Return the server response normally and inject a random comment in the response.
   b. Block mode: Return a special response page stating that the system is down for maintenance, including specific detailed information about the recovery time. The blocking mode flag will be automatically reset after a configurable time (default period=5 minutes).

Example

Fake parameter candidates:

```
{"JESSIONID", "SID", "ID", "SESSID", "AUTHID", "AUTHLEVEL",
"JSERVERSESSION", "SESSIONID", "CFID", "CFTOKEN", "SIDTOKEN"}
```

[Client → Server] Initial request from client to server:

HTTP GET http://localhost:8080/BodgeIt/basket.jsp

[Client ← Server] Original response header intercepted by the deception artifact
`Set-Cookie:JSESSIONID=0f212441490546a5d5a0833871c0;`

[Client ← *Server] Response modified by the deception artifact and delivered to the client

The first unused fake session id name will be picked from the candidate using the next session id counter value. This id will be associated with the original session id using the configured difficulty level (easy/medium/hard)
`Set-Cookie:SID=00000000000000001;`

[Client → Server] Request from client to server:

If the client modifies the values of the fake cookie, the deception artifact will detect it. The intrusion counter will be incremented in one unit and the request details will be logged. If the blocking mode is enabled, a special response page will be returned stating that the system is down for maintenance for a period of time (default period=5 minutes).

5.4. Deceptive HTTP Status Codes

Knowing the structure of a web application provides highly valuable information for the attacker, such as the available URLs, the way the URLS are linked and how the flow of the application is organized. This information can be obtained by crawling a website either manually or using the help of automated spiders to parse the pages of an application and to check and follow all possible links and different combinations of URLs that are found.

The objective of this strategy is to detect and prevent the crawling of web applications by replacing the real status codes of HTTP error requests with deceptive status codes trying to confuse adversaries with false information and therefore making them waste their time and resources.

Table 10. Deception Strategy 4. Deceptive Status Codes

Deception Strategy 4: Deceptive HTTP Status Codes	
Strategic goal	
The goal of this strategy is to detect and prevent website crawling attempts by attackers monitoring their activities and wasting their resources. This goal will be accomplished by replacing HTTP 4xx and 5xx (errors) with deceptive HTTP 200 (OK) fake status codes whenever a specific client exceeds a certain amount of error code responses returning from the server in a defined period of time.	
Attacker reactions	
How the attacker should react	How the attacker is desired to react

The attacker should try to make numerous crawling requests that would return 4xx or 5xx HTTP status codes, but after a certain threshold, 200 HTTP status codes will be returned. The attacker should believe that those responses are valid and that the requested URLs exist.	The attacker should not be able to obtain the real status code of the crawling requests and should continue with its crawling attempts as a long as possible in order to exhaust its time or resources.

Attackers' biases

Personal	Organizational & Cultural	Cognitive
Not considered because there is no information about the attacker's personal background.	Not considered because there is no information about the attacker's organizational/cultural background.	Conjunction fallacy bias will be exploited [6]. Instead of simply blocking the detected crawling requests, a longer story will be provided returning HTTP 200 OK status code responses with a page saying the network is down for a limited time due to maintenance or technical problems.

Conditions and characteristics of deception

Deception will be integrated in system responses by masking real HTTP response status codes with fake HTTP 200 OK and inventing a new response page.

Feedback channels

The monitoring of the HTTP response status codes will constitute the feedback channel of this strategy.

Risks and countermeasures

Only 4xx-5xx error responses will be replaced. There is no risk of breaking the normal flow of the application if only error messages are replaced.

Implementation and integration

The deception strategy will be integrated into the application-layer HTTP traffic. The artifact implementing the strategy will have to complete the following steps:

1. Intercept server HTTP response status codes of the original application traffic
2. If the server responds with 4xx or 5xx status codes, the error counter will be incremented in one unit.
3. If the error counter exceeds a specified threshold (default threshold=5), the deception artifact will consider that the server is receiving crawling requests and a crawling detection flag will be enabled.
4. If the crawling detection flag is enabled, all the HTTP 4xx-5xx responses from the server will be replaced with HTTP 200 OK responses providing a special response page stating that the system is

down for maintenance and giving some specific detailed information about the reason or the recovery time. The crawling detection flag will be automatically reset after a configurable time period (default period=60 seconds).
Example
[Client → Server] Multiple Requests from client to server returning 404 status codes HTTP GET http://localhost:8080/BodgeIt/admin HTTP GET http://localhost:8080/BodgeIt/setup HTTP GET http://localhost:8080/BodgeIt/checkout HTTP GET http://localhost:8080/BodgeIt/config HTTP GET http://localhost:8080/BodgeIt/cmd … The error counter will count 5 error attempts, reaching the limit threshold. The crawling detection flag will be enabled for 60 seconds. [Client ← *Server] Following 404 response codes will be replaced with HTTP 200 OK responses This behavior will last until the configured period expires without returning any error code (60 seconds).

5.5. Deceptive JavaScript

This strategy uses JavaScript as a decoy to attract the attention of an attacker with fake variables and URLs. The goal of this strategy is to detect attackers with high attacking capabilities and be able to detect and block them.

Table 11. Deception Strategy 5. Deceptive JavaScript

Deception Strategy 5: Deceptive JavaScript	
Strategic goal	
The goal of this strategy is to detect attackers with advanced attacking skills in order to monitor their activities or block their access to the web application.	
Attacker reactions	
How the attacker should react	How the attacker is desired to react
The attacker should analyze the JavaScript code, find a variable with an attractive fake URL and make a request to that URL.	The attacker should be monitored or blocked, depending on the configured option. Two options are considered: • Continue: Requests to URLs extracted from deceptive JavaScript will return a valid HTTP

	response containing a new different fake JavaScript content to keep the attacker interested and interacting with the system while being monitored. • **Block**: When a request to a fake link extracted from JavaScript is detected, the attacker will be flagged and a special page will be returned simulating that the website is down for maintenance for a limited period. The attacker should give up the attempts because of not being able to connect to the web application

Attackers' biases		
Personal	**Organizational & Cultural**	**Cognitive**
Not considered because there is no information about the attacker's personal background.	Not considered because there is no information about the attacker's organizational/cultural background.	Conjunction fallacy bias will be exploited [6] to make the JavaScript code believable.

Conditions and characteristics of deception
Deception will be integrated in system responses by injecting a fake JavaScript function into the real HTTP response.
Feedback channels
The monitoring of the HTTP request URLs will constitute the feedback channel of this strategy.
Risks and countermeasures
Fabricated deceptive JavaScript function name must always be distinct from existing JavaScript functions that the application uses for its real use cases, not to cause interferences with the normal flow of the application.
Implementation and integration
HTTP traffic. The artifact implementing the strategy will have to complete the following steps: 1. Intercept HTTP responses of the original application and inject a fake JavaScript function before delivering the response to the client. The function will contain an attractive variable with a newly generated fake URL. 2. Intercept HTTP requests and check if the request is targeting a fake URL that has been injected inside a JavaScript variable in previous requests. If a request to a fake URL is detected, the system will have to:

3. Trigger an intrusion detection alarm, increase the alert counter and log the request details to a log file.
4. Return a response depending on the configured mode:
    a. Continue mode: Return a valid HTTP response with 200 status code containing at least a new additional fake URL that the attacker could access.
    b. Block mode: Return a valid HTTP response providing a special response page stating that the system is down for maintenance and giving some specific detailed information.

Example

[Client → Server] Initial request from client to server:

HTTP GET http://localhost:8080/BodgeIt

[Client ← Server] Original response intercepted by the deception artifact

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
<title>The BodgeIt Store</title>
<link href="style.css" rel="stylesheet" type="text/css">
…
</head>
<body>
…
</body>
```

[Client ← *Server] Response modified by the deception artifact and delivered to the client

A fake JavaScript function will be injected containing a new random URL

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
<title>The BodgeIt Store</title>
<link href="style.css" rel="stylesheet" type="text/css">
…
<script type="text/javascript">
<!--
// TODO: Always remove this for production
function getDevelSiteAuthToken() {

       var url = "http://localhost:8080/0a18a1f2bb315e123c1ad72314f"
       var xhr = new XMLHttpRequest();
       var tokenElement = document.getElementById('token');
       var resultElement = document.getElementById('result');
       xhr.open('GET', url, true);
       xhr.setRequestHeader("Authorization", "JWT "+ tokenElement.innerHTML);
       xhr.addEventListener('load', function() {
       var responseObject = JSON.parse(this.response);
       console.log(responseObject);
       resultElement.innerHTML = this.responseText;
       });
```

```
        xhr.send(null);
//-->
</script>

</head>
…
```

[Client → Server] Request from client to server:

If the client accesses the generated URL contained in the JavaScript function, the deception artifact will intercept it:

HTTP GET http://localhost:8080/BodgeIt/0a18a1f2bb315e123c1ad72314f

The deceptive JavaScript intrusion counter will be incremented in one unit and the request details will be logged. Depending on the configured mode, the following action will be performed.

- Continue mode: Return the original response coming from the application injecting a new additional fake JavaScript content that the attacker could access.
- Block mode: Return a valid HTTP response providing a special response page stating that the system is down for maintenance and giving some specific detailed information about the recovery time. The blocking mode flag will be automatically reset after a configurable time (default period=5 minutes).

CHAPTER SIX

6. IMPLEMENTATION AND EVALUATION

This chapter covers the design and implementation of the testing environment that is needed to evaluate the performance of the five deception strategies specified in the previous chapter. The evaluation or testing environment is composed of three main elements: a web application scanner (OWASP ZAP [51]), two vulnerable applications (BodgeIt Store [61] and WAVSEP [56]) and the software artifact in charge of intercepting traffic and implementing the logic of our five deception strategies that needs to be designed and implemented.

The applications and the artifact have been deployed on a Java EE 6 compliant web server (Payara 4.1.171) on top of a 1.8.0_121 Oracle Java Virtual Machine.

The following figure shows the basic composition of the testing environment:

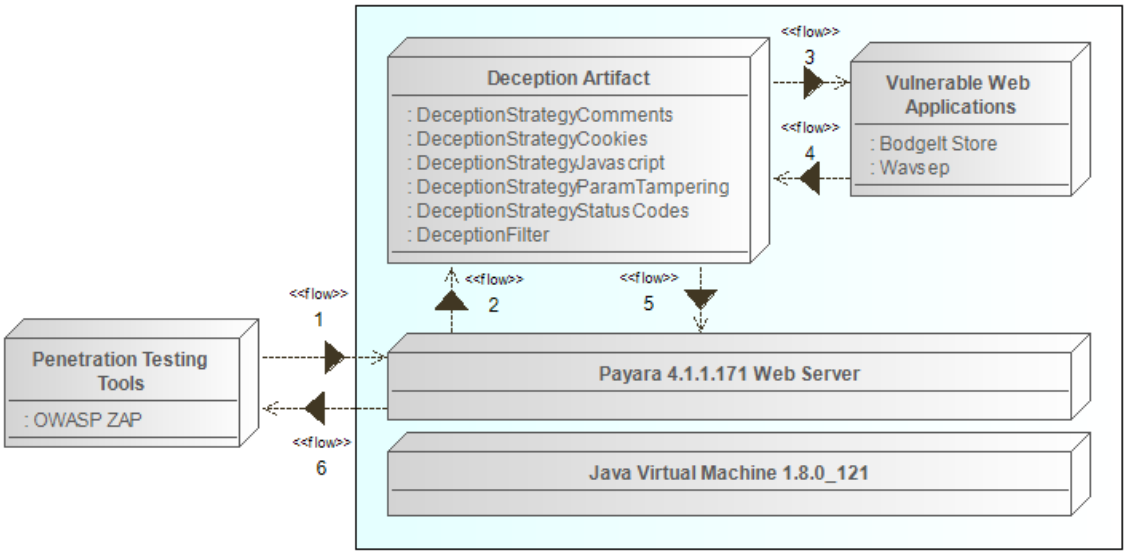


Figure 9 Testing environment overview

These are the main steps of the deception process:

Step 1: Requests originating from the penetration testing tool arrive to the web server's HTTP request listener.

Step 2: Requests are intercepted by the deception module instead of following the original application request/response lifecycle. Requests are processed according to the defined deception strategies.

Step 3: Requests continue its way and are processed by the destination web application.

Step 4: Responses from the application are intercepted again by the deception artifact module and are processed according to the defined deception strategies.

Step 5: The final response is ready to be delivered to the client and it is handled by the web server

Step 6: The response is delivered to the client.

The following tables show additional details about the components of the testing environment.

Table 12. Testing environment web server

Web server			
Name	Version	License	Platform
Payara (Glassfish 4 Fork)	4.1.1.171	CDDL	Java

Table 13. Deception artifact dependencies

Deception Artifact			
Dependencies	Version	License	Platform
Oracle Java Development Kit	1.8.0_121	OBCLA	Windows, Linux, Mac OS/X
JSOUP Library	1.10.2	MIT	Java

Table 14. Testing environment penetration testing tools

Web Application Scanners / Penetration Testing Tools			
Name	Version	License	Operating System
OWASP ZAP	2.6.0	ASF2	Windows, Linux, Mac OS/X

Table 15. Testing environment web applications

Vulnerable Web Applications			
Name	Version	License	Language
Bodgeit Store	1.4.0	ASF2	Java
WAVSEP	1.5.0	GPL	Java

6.1. Testing environment design and implementation

6.1.1. Deception artifact implementation

The software artifact implementing the five proposed deception strategies has been developed using the Java Development Kit and the Java programming language. The main goal of the implementation has been to provide a simple but performant implementation of the strategies that are easy to hook to existing web applications while providing a low operational overhead.

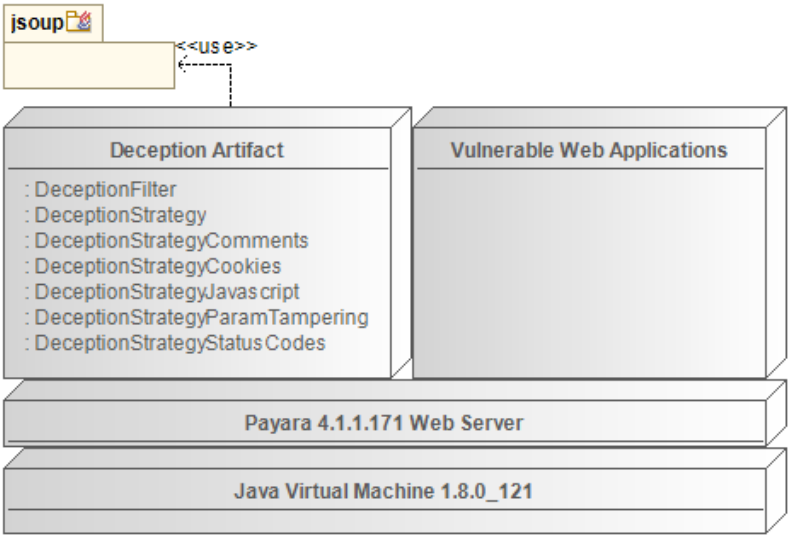


Figure 10 Deception artifact implementation overview

In order to meet those requirements Java Servlet Filters have been used. Java Servlet Filters are designed to hook additional preprocessing and postprocessing logic to existing applications running on any web server that supports the Java Servlet Specification 2.3 or higher. Filters allow the interception and modification of HTTP requests and responses in an efficient and straightforward way without requiring changing the source code of existing applications. The deployment is easily done just by changing a configuration file and adding a reference to the deception filter.

Initially, the deception artifact was intended to be implemented as a web application proxy, but the Java Servlet Filter option has been chosen in the end. The main advantage of the use of Servlet Filters compared to an independent reverse proxy is that requests and responses are processed very efficiently without creating any additional TCP connections thus minimizing the processing overhead. An external library has also been used (JSOUP 1.10.2) to parse and modify HTML content of the requests and responses.

The following class diagram shows an overview of the main classes that are part of the implementation of the deception artifact.

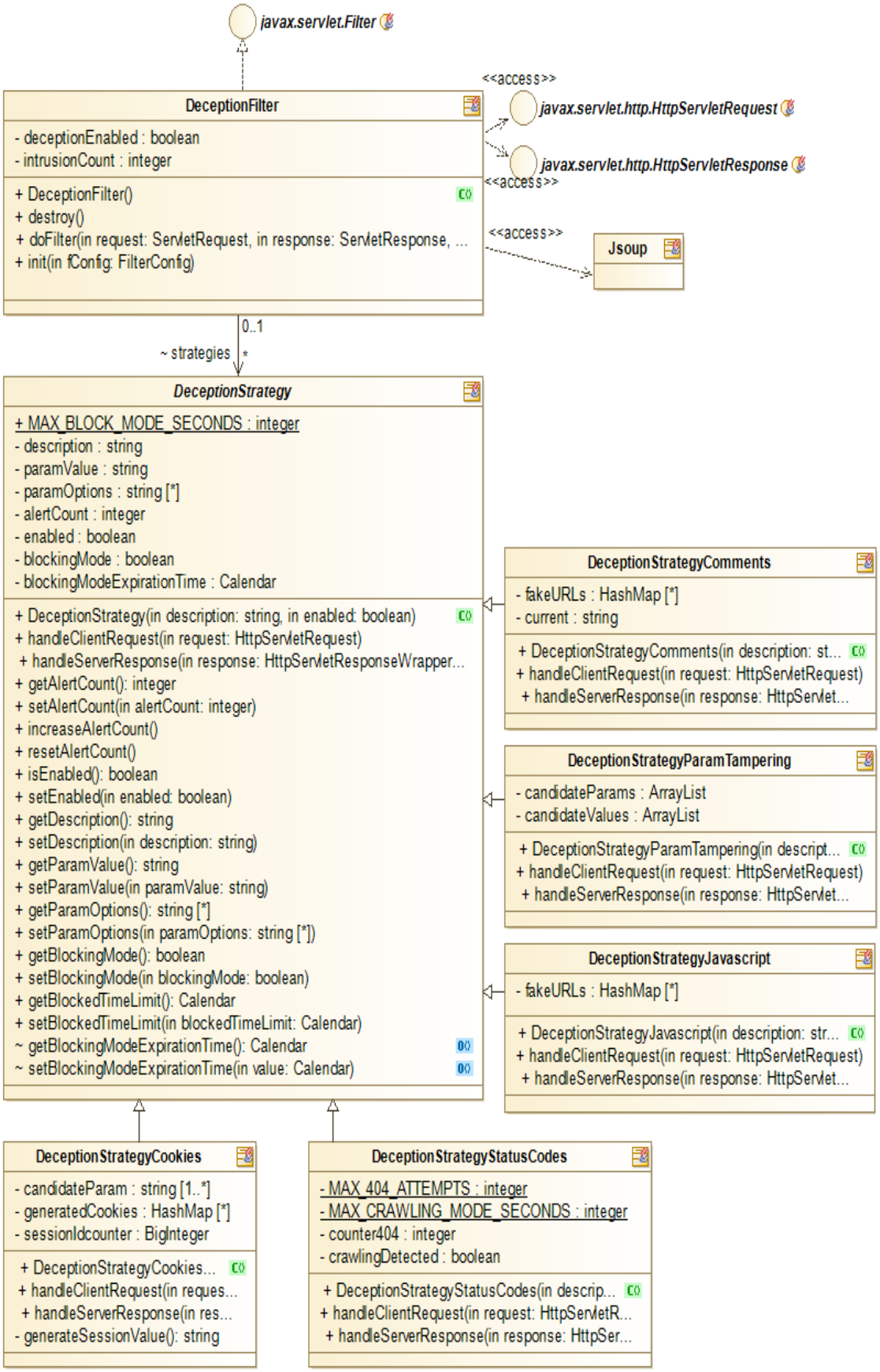


Figure 11 Deception artifact class diagram

The `DeceptionFilter` class implementing the `javax.servlet.Filter` interface is in charge of intercepting requests and responses of web applications running on the web server. A `DeceptionFilter` object maintains a list of `DeceptionStrategy` objects that receive the request and responses to be processed sequentially by all the strategy objects that are enabled in the list. `DeceptionStrategy` is an abstract class containing common functionalities that all strategies have in common. Different specializations of this abstract class have been implemented to override the specific logic needed to apply by each deception strategy: `DeceptionStrategyComments`, `DeceptionStrategyParamTampering`, `DeceptionStrategyCookies`, `DeceptionStrategyStatusCodes` and `DeceptionStrategyJavascript`.

When a request is intercepted by the `DeceptionFilter`, it is processed sequentially by all the enabled strategy instances invoking the `handleClientRequest` method that all the strategies override and afterwards the request is sent to the web application. The response follows a similar approach and it is intercepted and processed sequentially by all the enabled strategy instances invoking their `handleRequestRequest` methods before returning the request to the client.

The following sequence diagram shows a high-level overview of the sequence of the main elements that take part in the process.

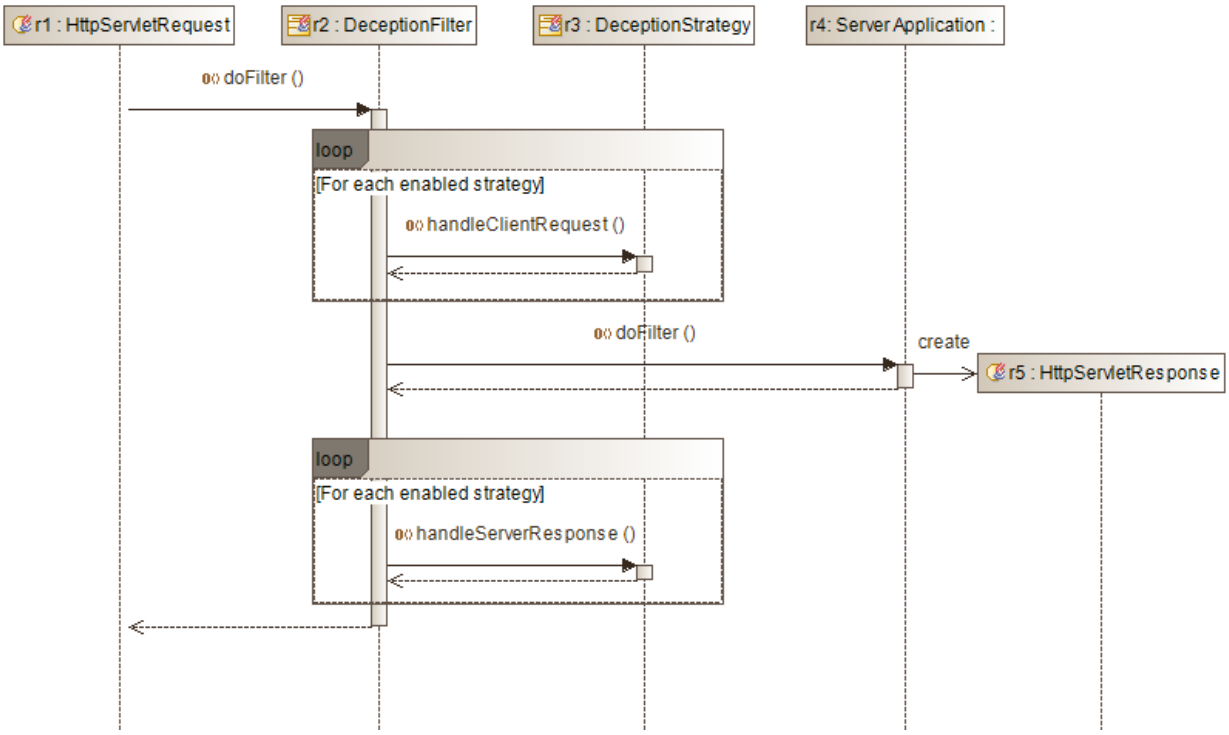


Figure 12 Deception artifact sequence diagram

The implementation of the deception artifact includes a simple GUI that is useful to configure the parameters of the strategies without modifying files or restarting the server. The GUI is accessed from a web browser using a special URL with the `/DECEPTION_CONFIG` prefix and it allows to control basic aspects of the deception process, such as the intrusion alerts generated

by each strategy and also enabling, disabling, resetting or setting configuration parameters of the strategies.

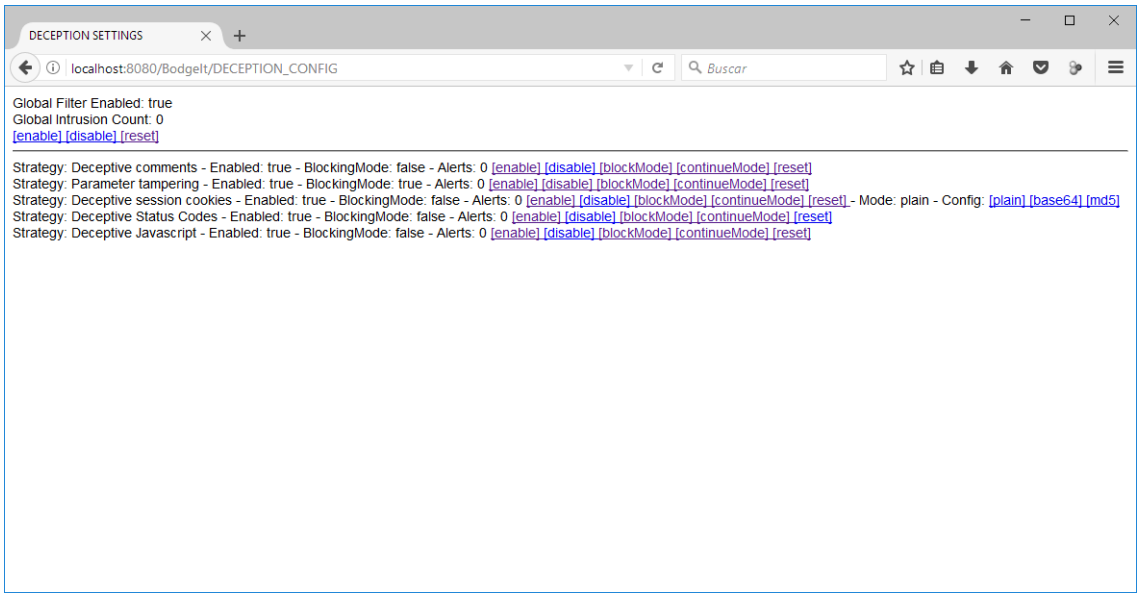


Figure 13 Deception artifact result and configuration screen

Configurable options can also be changed without the web browser, just by invoking HTTP requests adding the following parameters to indicate the main action: `disable`, `enable`, `reset`, `config`, followed by the index of the strategy: `global,0,1,2,3,4`. The `config` action can be followed by `param` and `blockingMode` parameters.

Table 16 Deception strategy configuration URLs

Parameters		
Main actions	Index	Additional parameters
disable, enable, reset	global,0,1,2,3,4	-
config	global,0,1,2,3,4	param, blockingMode
Configuration using URLs. Examples		
Action	URL	
Disable the deception artifact globally	http://host:port/app/DECEPTION_CONFIG?disable=global	
Enable parameter tampering strategy	http://host:port/app/DECEPTION_CONFIG?enable=2	
Disable deceptive cookies strategy	http://host:port/app/DECEPTION_CONFIG?disable=3	
Configure deceptive cookies strategy with the MD5 option	http://host:port/app/DECEPTION_CONFIG?config=3&param=md5	
Configure deceptive cookies strategy to blocking mode	http://host:port/app/DECEPTION_CONFIG?config=3&blockingMode=true	

When strategies are operating in block mode, a special page is returned showing a message and simulating that that the system is down for maintenance and stating the exact time when the system will be available again. When the block mode is activated by any of the strategies, this

page is returned to the user but requests and responses will continue to be processed by the deception artifact. This means that even if the user response is blocked with the special page, the requests will continue to be processed by all the enabled strategy objects and new alerts could continue to be triggered.

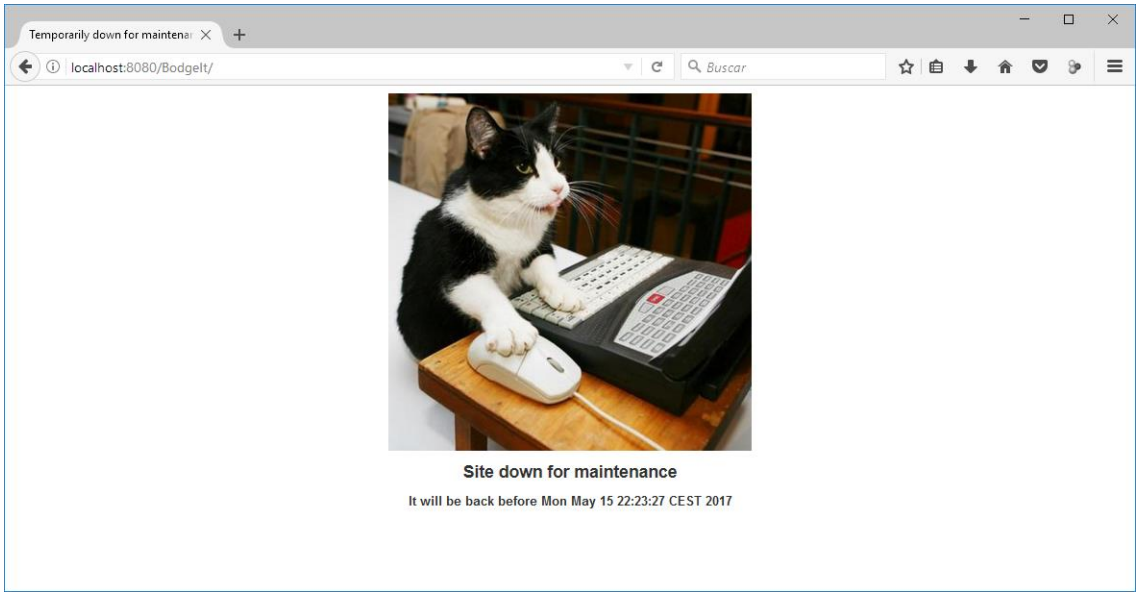


Figure 14 Deception artifact block mode response page

6.1.2. Penetration testing tool: OWASP ZAP

OWASP ZAP is a popular free an open source web application vulnerability scanner that provides many options to perform automatic or manual security penetration testing of web applications. OWASP ZAP is highly configurable and extensible using plugins and it is actively maintained by hundreds of international volunteers. The last version of OWASP ZAP is 2.6.0, released in March 2017.

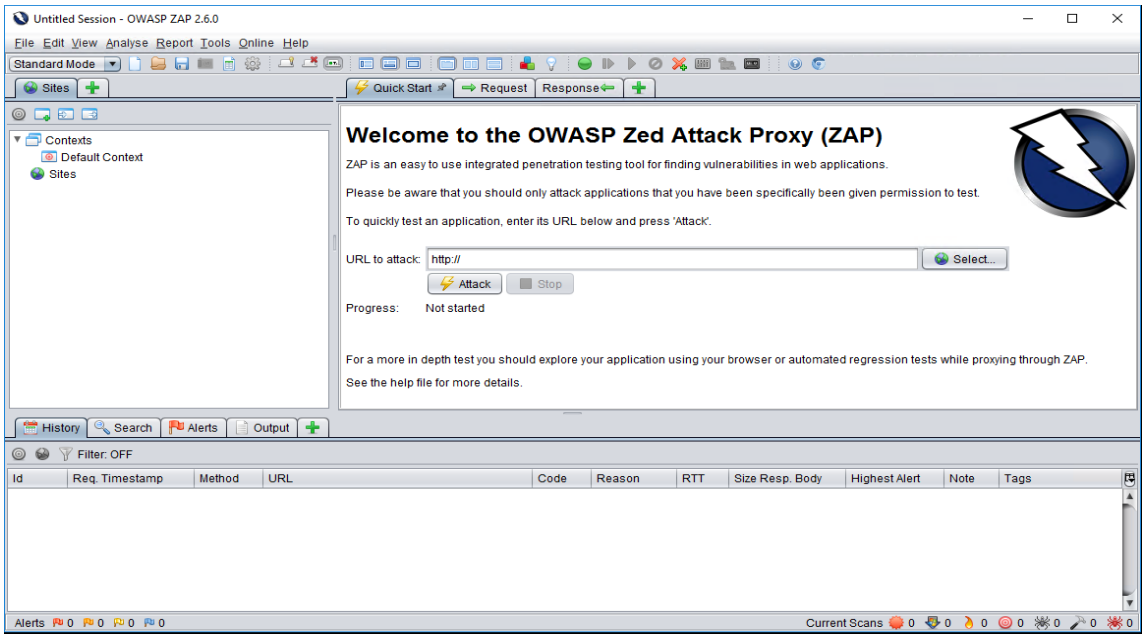


Figure 15. OWASP ZAP Application

6.1.3. Vulnerable web applications: BodgeIt Store and WAVSEP

BodgeIt Store is a vulnerable web application that runs on any web server supporting the Java Servlet specification. It mimics an online store with product listings, search page, shopping cart and checkout process and it uses an internal in-memory database. BodgeIt Store contains the following documented vulnerabilities: Cross Site Scripting, SQL injection, Hidden (but unprotected) content, Cross Site Request Forgery, Debug code and Insecure Object References.

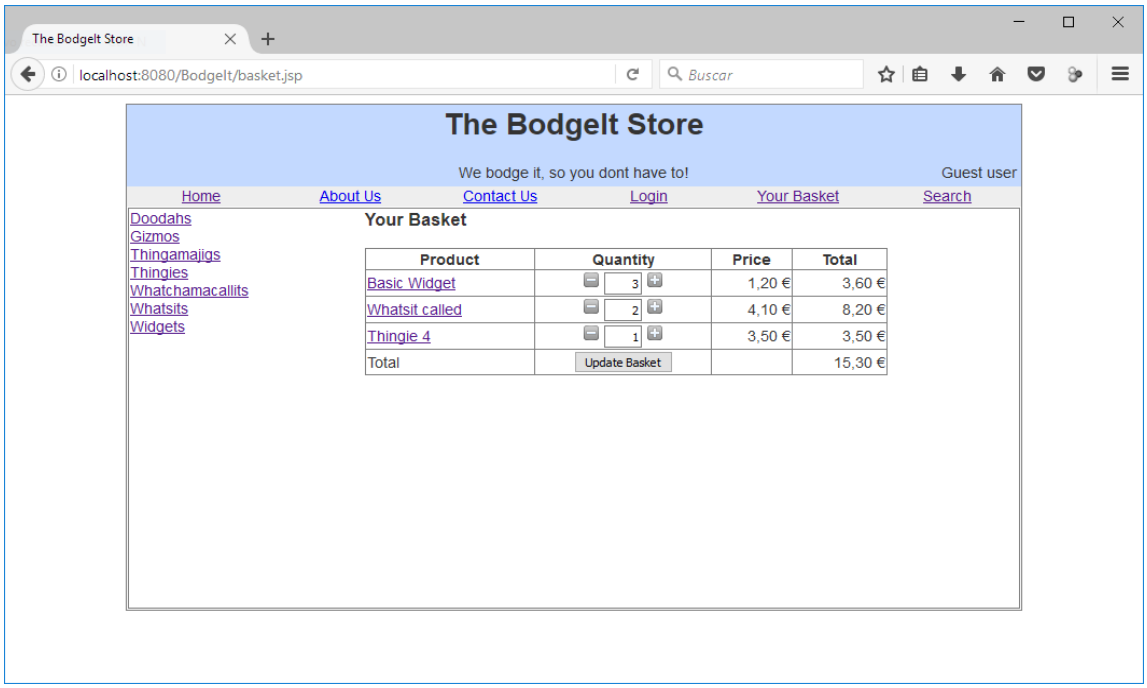


Figure 16. BodgeIt Store Web Application

WAVSEP is a vulnerable web application that can be deployed on web servers complying with the Java Servlet Specification and it is backed by a MySQL database that needs to be configured and initialized using a provided script. WAVSEP is a comprehensive application designed to help assess the quality and accuracy of web application scanners and it provides hundreds of unique URLs designed to test different properties and vulnerabilities. The documented vulnerabilities include: Path Traversal/LFI, Remote File Inclusion (XSS via RFI), Reflected XSS, Error Based SQL Injection, SQL Injection, Time Based SQL Injection, Unvalidated Redirects, Backup and Unreferenced Files, Passive Information Disclosure/Session Vulnerabilities, test cases of erroneous information leakage, and improper authentication / information disclosure.

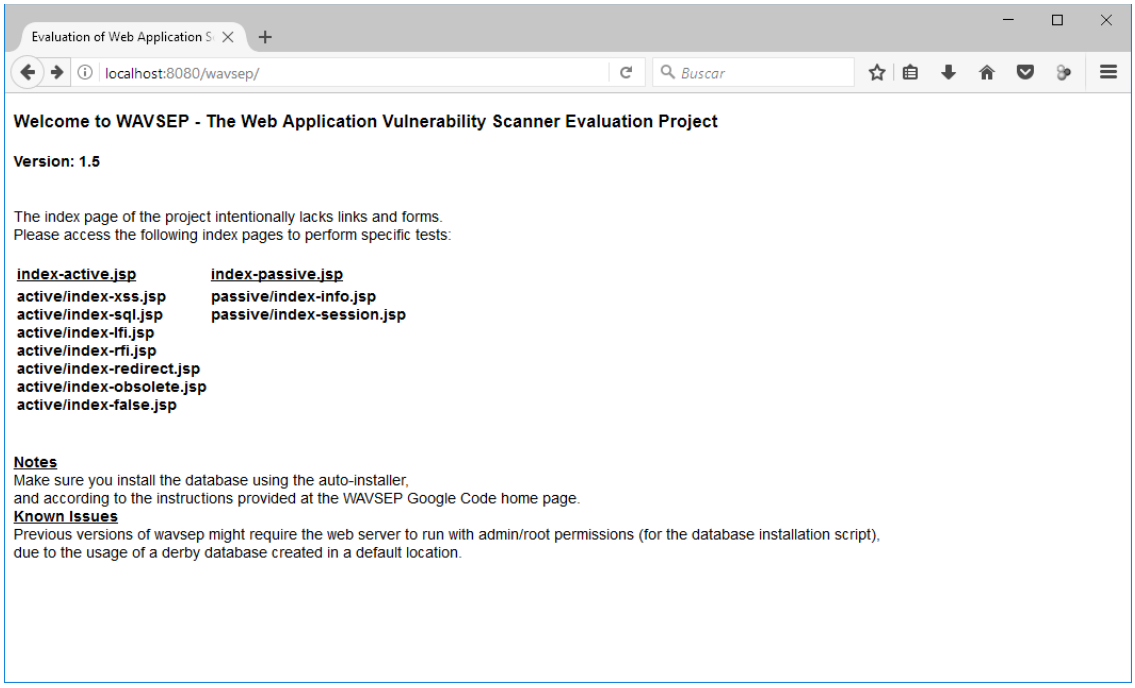


Figure 17. WAVSEP Web Application

6.2. Testing procedure

6.2.1. Deception artifact implementation functional tests

The correct operation of the implementation of the deception artifact has been verified with the creation of a script that launches different sets of requests that should and should not trigger different deception alerts and checking that result logs and alerts generated by the artifact match expected results.

The test script has been written in the Python 3.x [62] programming language using the "requests" [63] and "numpy" [64] libraries. The same script has also been used later to launch controlled requests and measure the operational performance of the artifact in section 6.2.2.1.

The script automatically configures the artifact enabling a single strategy at a time and launching a set of tests, as shown in the table below.

Table 17. Python test script steps

Python Script (test_operational.py)	
Configuration	Requests launched
Step 1: Strategy 1 (Deceptive Comments) ON	100 Requests that should trigger an intrusion alarm for strategy 1 100 Requests that should not trigger an intrusion alarm for strategy 1
Step 2: Strategy 1 (Deceptive Comments) OFF	200 Same exact requests from Step 1
Step 3: Strategy 2 (Deceptive Parameters) ON	100 Requests that should trigger an intrusion alarm for strategy 2 100 Requests that should not trigger an intrusion alarm for strategy 2
Step 4: Strategy 2 (Deceptive Parameters) OFF	200 Same exact requests from Step 3

Step 5: Strategy 3 (Deceptive Cookies) ON	100 Requests that should trigger an intrusion alarm for strategy 3 100 Requests that should not trigger an intrusion alarm for strategy 3
Step 6: Strategy 3 (Deceptive Cookies) OFF	200 Same exact requests from Step 5
Step 7: Strategy 4 (Deceptive Status) ON	100 Requests that should trigger an intrusion alarm for strategy 4 100 Requests that should not trigger an intrusion alarm for strategy 4
Step 8: Strategy 4 (Deceptive Status) OFF	200 Same exact requests from Step 7
Step 9: Strategy 5 (Deceptive JavaScript) ON	100 Requests that should trigger an intrusion alarm for strategy 5 100 Requests that should not trigger an intrusion alarm for strategy 5
Step 10: Strategy 5 (Deceptive JavaScript) OFF	200 Same exact requests from Step 9
Step 11: All Strategies combined ON	500 Requests that should trigger an intrusion alarm (100 per strategy) 500 Requests that shouldn't trigger an intrusion alarm
Step 12: All Strategies combined OFF	1000 Same exact requests from Step 11

6.2.2. Performance evaluation

The evaluation of the performance of the deception strategies has been completed from two perspectives: operational and security.

Measuring operational aspects of the deception artifact requires using exactly the same requests with the same content and timing, and this is not possible to achieve using automated tools like OWASP ZAP because the attacking behavior and exact number of requests generated gets changed and varies depending on the actions that the deception artifact is performing in a given moment. This is why a custom-made script has been used to launch exactly the same requests and measure the operational performance of the deception artifact. When it comes to measure the security performance of the deception strategies, several attack scenarios have been used using the OWASP ZAP tool.

6.2.2.1. Operational performance testing

The script has been configured to launch the same exact number of requests against the BodgeIt application and calculate the average roundtrip time for each step that is configured to enable a specific strategy and operating mode.

Roundtrip times have been measured using the `elapsed()` method of the requests library. According to the requests library documentation [63], the `elapsed()` method returns:

"the amount of time elapsed between sending the request and the arrival of the response (as a timedelta). This property specifically measures the time taken between sending the first byte of the request and finishing parsing the headers"

This means that the returned values provide an accurate server round-trip measurement because they are exactly measuring the time elapsed between the request and the arrival of the first byte of the response. The result is unaffected by how the response content is later consumed or parsed by the client.

```
Python 3.6.1 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: \\tsclient\C\Users\Mikel\Desktop\InfoSec\y2_2016-17\Sp3-4\seminar 5 30-05\test_operational.py
Press enter to start tests...
Average roundtrip time for Strategy 1 OFF: 0.001697265
Average roundtrip time for Strategy 1 ON: 0.0017669575
Strategy 1. ON Higher time 0.00960557908156 percent
Average roundtrip time for Strategy 2 OFF: 0.00186562666667
Average roundtrip time for Strategy 2 ON: 0.00199569125
Strategy 2. ON Higher time 0.00934827301902 percent
Average roundtrip time for Strategy 3 OFF: 0.001888197
Average roundtrip time for Strategy 3 ON: 0.00180877
Strategy 3. OFF Higher time 0.00957935003604 percent
Average roundtrip time for Strategy 4 OFF: 0.001803635
Average roundtrip time for Strategy 4 ON: 0.001807323125
Strategy 4. ON Higher time 0.00997959343878 percent
Average roundtrip time for Strategy 5 OFF: 0.00179441
Average roundtrip time for Strategy 5 ON: 0.0017965645
Strategy 5. ON Higher time 0.00998800766686 percent
Average roundtrip time for ALL 5 OFF: 0.00177079733333
Average roundtrip time for ALL 5 ON: 0.00185642225
ALL. ON Higher time 0.00953876378789 percent
>>>
                                                    Ln: 20  Col: 18
```

Figure 18. Python test script execution

CPU and memory overhead have been measured using the Performance Monitor included in Windows, using the option to measure the properties of a specific process. The Performance Monitor shows a graph and detailed information on screen, but it also gives the option to save full detailed results to a file for later study.

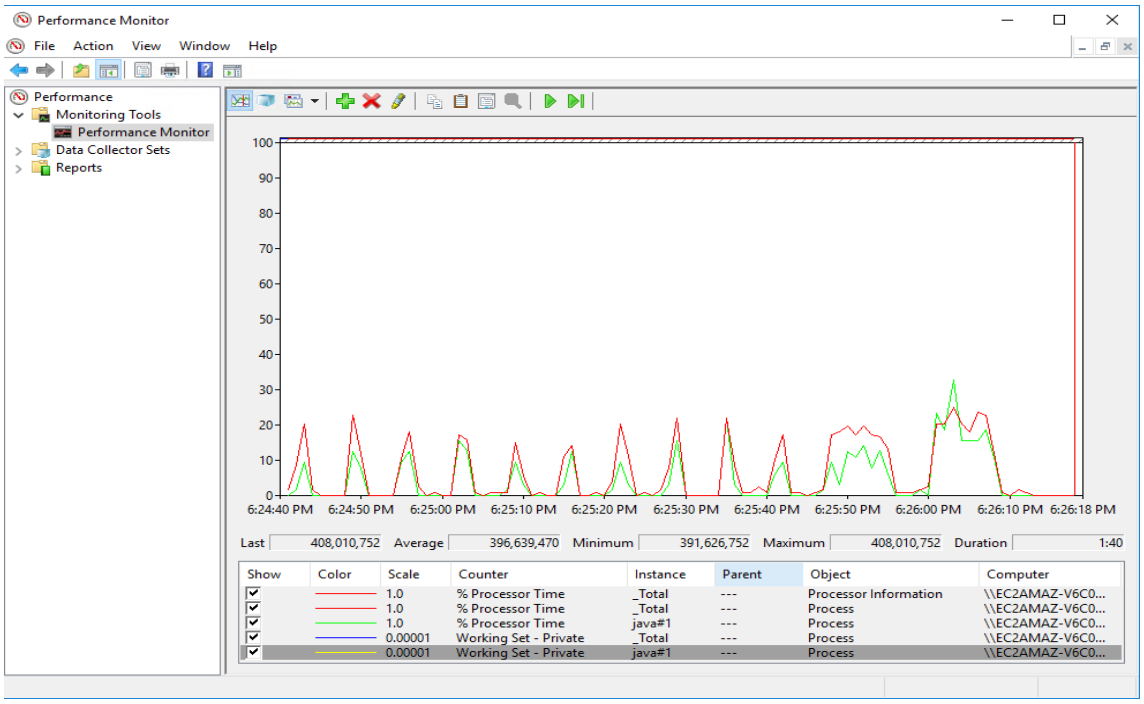


Figure 19. Windows Performance Monitor

6.2.2.2. Security performance testing

Security performance of the deception strategies has been evaluated executing attacks using the OWASP ZAP tool. All tests are executed on the same machine after a fresh reboot using different configuration options of the tools that are described below.

Table 18. Testing machine specs

Execution environment	
Element	Details
Operating System	Windows Server 2016 Datacenter
CPU	Intel Xeon E5-2676. 2x40Ghz
RAM	4.00 Gb

Table 19. OWASP ZAP baseline configuration

OWASP ZAP Baseline Configuration			
Configuration Option	Threshold	Strength	Quality
Directory Browsing	Default	Default	Release
Buffer Overflow	Default	Default	Release
CRLF Injection	Default	Default	Release
Cross Site Scripting (Persistent)	Default	Default	Release
Cross Site Scripting (Persistent) – Prime	Default	Default	Release
Cross Site Scripting (Persistent) – Spider	Default	Default	Release
Cross Site Scripting (Reflected)	Default	Default	Release
Format String Error	Default	Default	Release
Parameter Tampering	Default	Default	Release
Remote OS Command Injection	Default	Default	Release
Server Side Code Injection	Default	Default	Release
Server Side Include	Default	Default	Release
SQL Injection	Default	Default	Release
External Redirect	Default	Default	Release
Script Active Scan Rules	Default	Default	Release
Path Traversal	Default	Default	Release
Remote File Inclusion	Default	Default	Release
Scan Policy configuration file content			

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<configuration>
<policy>Default</policy>
<scanner>
<level>MEDIUM</level>
<strength>MEDIUM</strength>
</scanner>
<plugins>
<p6><enabled>true</enabled></p6>
<p7><enabled>true</enabled></p7>
<p40009><enabled>true</enabled></p40009>
<p40012><enabled>true</enabled></p40012>
<p40014><enabled>true</enabled></p40014>
<p40018><enabled>true</enabled></p40018>
<p90019><enabled>true</enabled></p90019>
<p90020><enabled>true</enabled></p90020>
<p0><enabled>true</enabled></p0>
<p20019><enabled>true</enabled></p20019>
<p30001><enabled>true</enabled></p30001>
<p30002><enabled>true</enabled></p30002>
<p40003><enabled>true</enabled></p40003>
<p40008><enabled>true</enabled></p40008>
```

```
<p40016><enabled>true</enabled></p40016>
<p40017><enabled>true</enabled></p40017>
<p50000><enabled>true</enabled></p50000>
</plugins>
</configuration>
```

The common attacking procedure of OWASP ZAP is divided in two basic steps: the crawling of the web application and the attacks or intrusion attempts. OWASP ZAP provides both "Spider" and "Attack" modules to perform these steps.

Crawling web applications using OWASP ZAP Spider

The Spider module of OWASP ZAP is used to scan the contents of a web application by following links and checking common URL patterns in order to return a list of available URLs that the application has. This step is mandatory because the attacking module that will be launched later needs to have a known list of URLs to launch the attacks. The crawling step is launched with the default options by using the "Attack->Spider" option

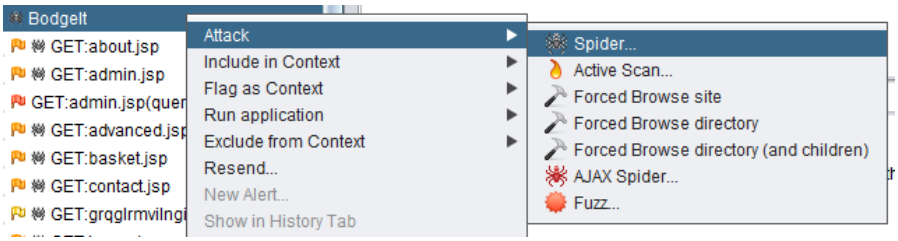


Figure 20. OWASP ZAP Spider invocation

Each application has a different starting point for the Spider, as shown in the table below.

Table 20. OWASP ZAP Spider starting paths

Starting Paths for the OWASP Spider	
Application	Starting Path
BodgeIt Store	http://localhost:8080/BodgeIt
WAVSEP	http://localhost:8080/wavsep/index-active.jsp

After launching the Spider, OWASP ZAP will compile a list of URLs that have been found in the application.

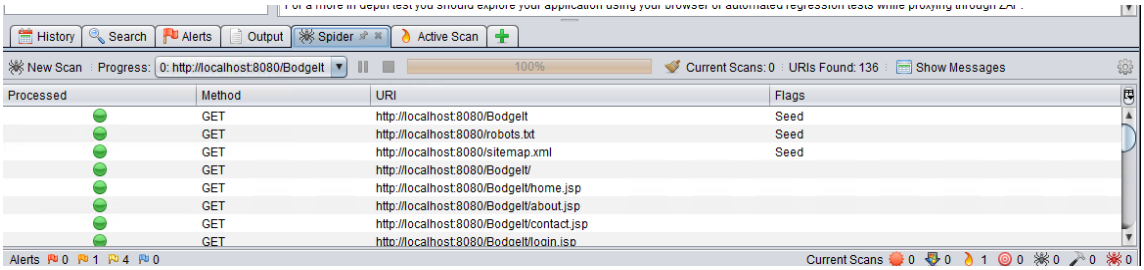


Figure 21. OWASP ZAP Spider running

Attacking the web application using the Active Scan module

The Active Scan module of OWASP ZAP allows to perform real active attacks to find and check the existence of vulnerabilities in the target web application. The Active Scan module can be launched using the "Attack->Active Scan" menu option.

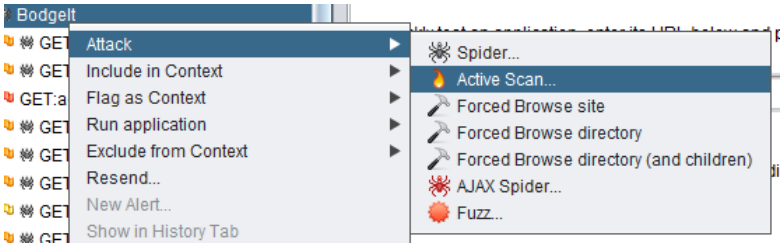


Figure 22. OWASP ZAP Attack invocation

The default attacking options will be changed by enabling the "HTTP Headers" and "Cookie Data" vectors (they are disabled by default). This will increase the time needed to perform the tests, but will increase the amount of attack attempts.

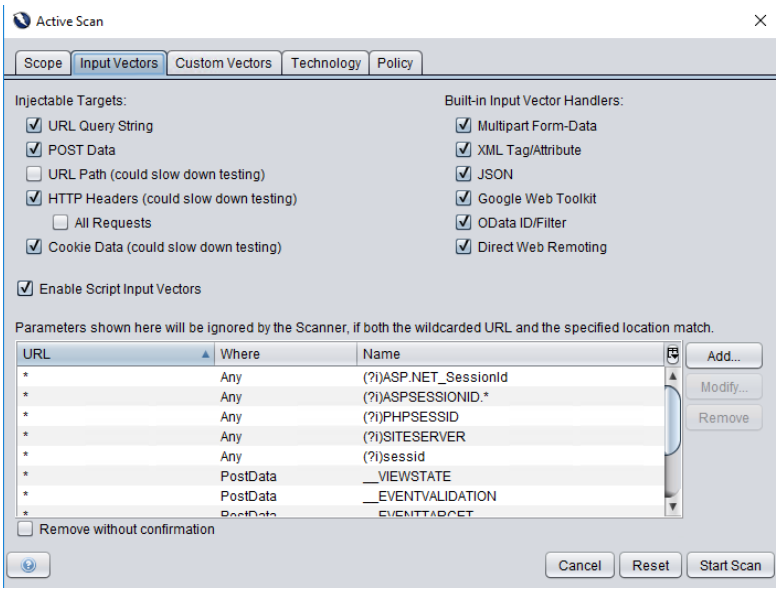


Figure 23. OWASP ZAP Attack parameters

After clicking on "Start Scan" OWASP ZAP will begin the attacks and will log and classify all found vulnerabilities in its database.

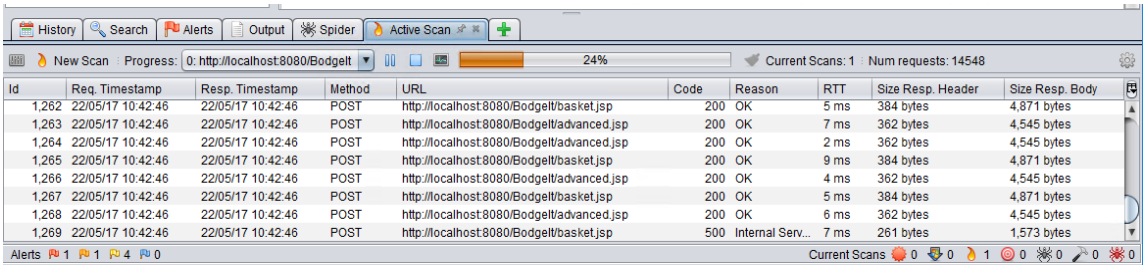


Figure 24. OWASP ZAP Attack running

The progress of the active scan process can be followed from the Progress Information window.

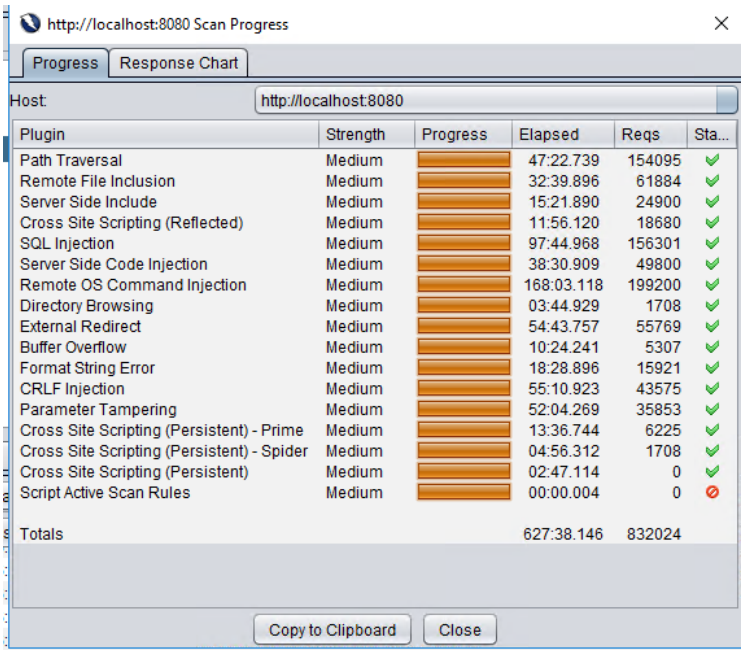


Figure 25. OWASP ZAP Attack progress information

The performance evaluation of the deception artifact using OWASP ZAP Spider and Active Scans will be measured using 6 different scenarios enabling and disabling the artifact and combining the blocking mode.

Table 21. Security evaluation scenarios

Security Evaluation of Deception Strategies			
Scenario	**Web Application**	**Deception Artifact (all strategies enabled)**	**Deception Artifact Mode**
Scenario 1	BodgeIt	OFF	-
Scenario 2	BodgeIt	ON	Continue
Scenario 3	BodgeIt	OFF	Blocking
Scenario 4	WAVSEP	OFF	-
Scenario 5	WAVSEP	ON	Continue
Scenario 6	WAVSEP	ON	Blocking

The following variables will be measured after the completion of each attack session: Number of requests generated, penetration testing time, amount and types of vulnerabilities detected by OWASP ZAP, number of alerts triggered by each one of the deception strategies.

CHAPTER SEVEN

7. RESULTS AND DISCUSSION

This chapter presents the results obtained by following the testing procedure from Chapter 6 showing both operational and security performance results.

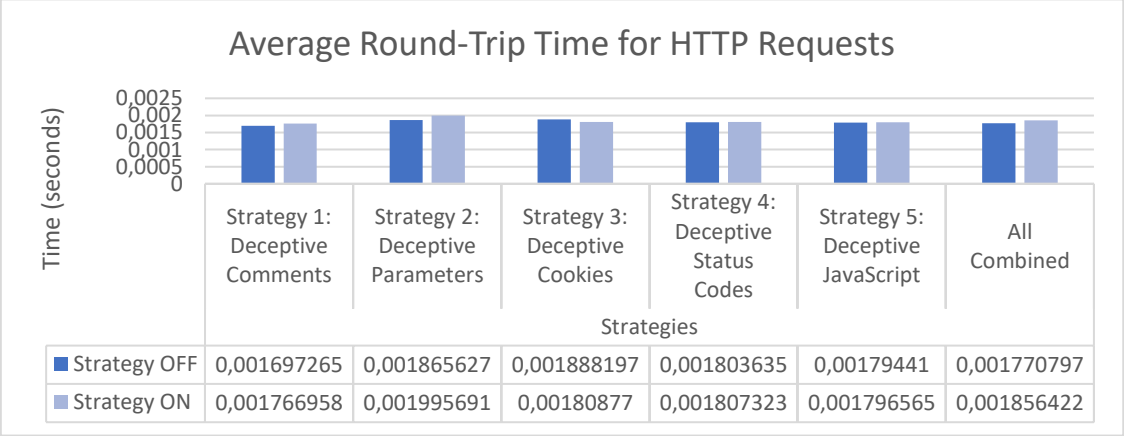
After presenting the results, several discussion sections are provided concerning the design iterations performed and the issues faced, reflections about the validity of the design, lessons learned and answering the research questions in the light of the results.

7.1. Operational Performance

Operational performance variables measure operating aspects that are not related with the security performance. In our case, server processing time, CPU load and memory resources that are consumed by the deception artifact implementation were measured.

7.1.1. Request Round-Trip Times

Request Round-Trip Times were measured as a way of calculating the server processing time added by the execution of the deception strategies. This was measured using the test script that configured the deception artifact enabling and disabling the strategies one by one, and sending the same set of requests each time.



	Strategy 1: Deceptive Comments	Strategy 2: Deceptive Parameters	Strategy 3: Deceptive Cookies	Strategy 4: Deceptive Status Codes	Strategy 5: Deceptive JavaScript	All Combined
Strategy OFF	0,001697265	0,001865627	0,001888197	0,001803635	0,00179441	0,001770797
Strategy ON	0,001766958	0,001995691	0,00180877	0,001807323	0,001796565	0,001856422

Figure 26. Average Round-Trip Time Results

Results indicated that the round-trip times of the requests that the python script was sending to our deception artifact and vulnerable web applications were very low: all responses were served in less than 0.002 seconds. The graph above shows a summary of the average round-trip times for each strategy execution.

When the deception strategies were ON, all strategies except Strategy 3 presented a slight augmentation in their response times. However, the difference was lower than 0.01 % in all cases and can be considered insignificant.

No reason has been found to explain why lower average round-trip times were measured when enabling the deception artifact with Strategy 3. Tests were repeated several times and

sporadically similar behaviors were observed where requests with the enabled deception strategy were obtaining a lower response time than the requests with the disabled strategy. The interpretation for this behavior is that the response times were extremely low and very close for both cases, and sometimes a minor interference caused by internal processes from the Operating System that were consuming CPU or I/O resources could generate these results.

7.1.2. CPU and memory usage

The graph below shows the CPU use percentage of the OS process running the deception strategies.

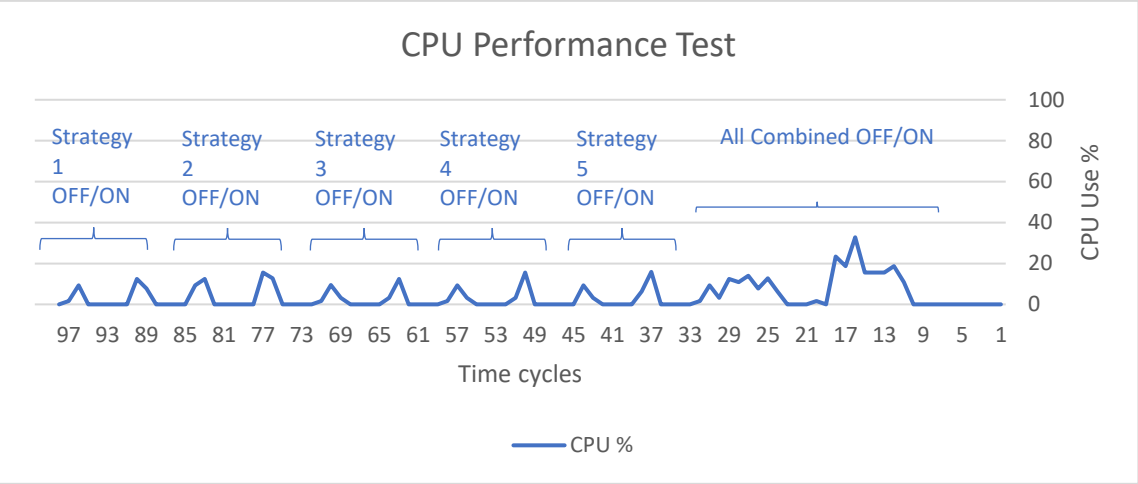


Figure 27. CPU Performance Results

CPU use data showed slight CPU use differences when requests were being processed by the deception artifact. This was due to the processing of the requests and responses by the artifact. The following table compiles the maximum CPU use % peak values that were measured for each scenario.

Table 22. CPU Performance peak results

CPU Performance Peaks			
Script test case	Artifact	Max. CPU %	CPU % Difference
Strategy 1	OFF	9.366057289	3.121919291
	ON	12.48797658	
Strategy 2	OFF	12.48773704	3.12223368
	ON	15.60997072	
Strategy 3	OFF	9.365931905	3.121842565
	ON	12.48777447	
Strategy 4	OFF	9.371905321	6.238021729
	ON	15.60992705	
Strategy 5	OFF	9.374795629	6.479466861
	ON	15.85426249	
All Strategies	OFF	14.04917295	8.73164439
	ON	32.78081734	

The obtained results showed that CPU use percentage increased between 3% and 8% for all cases when the deception strategies were being used. The last test case that combined all the strategies generated the highest number of requests and got the highest CPU use percentage increase with 8 points of difference.

These results are in line with the expected outcome, as the strategies must execute its logic and consume CPU time. The amount of CPU that was being consumed by the deception artifact was always under 10%, which is considered low and satisfactory.

The memory working set measures the amount of memory currently in use for a process. Our deception strategies and vulnerable web applications were running inside the same web server, which was running on top of a single Java Virtual Machine. All these elements were running on the same Java Virtual Machine instance (same java.exe windows process). In consequence, it was not possible to measure the amount of memory in use by the deception strategies specifically. The graph below shows the memory working set values that were obtained.

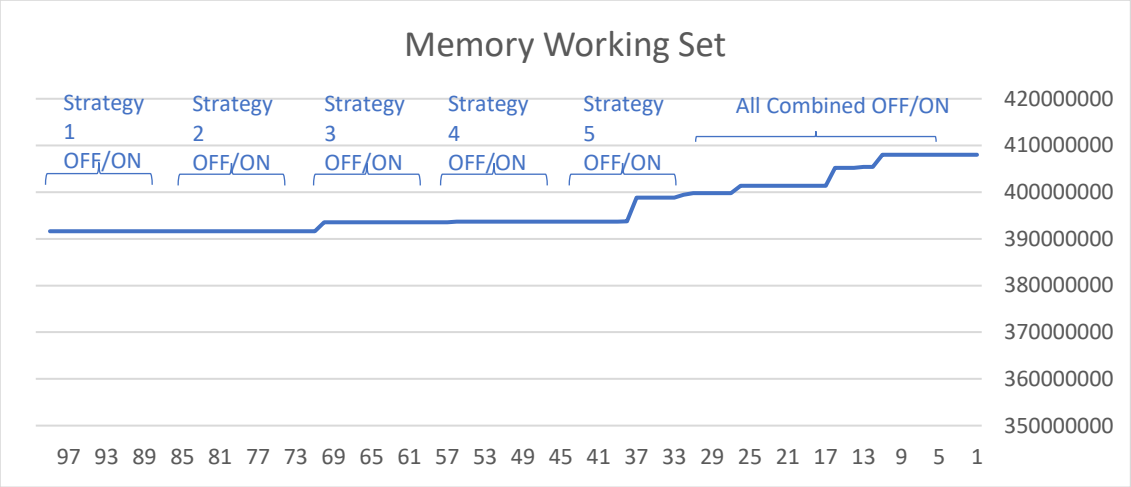


Figure 28. Memory Working Set Results

The results showed an overall memory usage increase from the beginning to the end of the test process, but they didn't reflect a clear memory usage difference between the moments when a certain deception strategy was enabled or disabled. Therefore, we can conclude that enabling the strategies did not generate a noticeable increase in memory usage.

The implementation of the deception artifact doesn't rely on any external database to store the active fake URLs and tokens that are generated and and need to be monitored. Therefore, they are stored in memory using java HashMaps and they consume memory. This means that there must be an increased memory usage when the number of generated/monitored tokens is high, even if the results form the testing environment did not show a noticeable increase.

7.2. Security Performance

Security performance variables measure aspects that are related with the security performance, that is, variables that could be linked with the ability of the artifact to detect or protect our web applications from attacks.

7.2.1. Generated HTTP Requests

This variable measures the number of HTTP Requests that are generated during each attack session. Although the number of requests by itself cannot be linked with security aspects, they are related with the time needed to perform the attacks that is discussed in section 7.2.2. The following figure shows and compares the number of HTTP Requests that were measured in different scenarios.

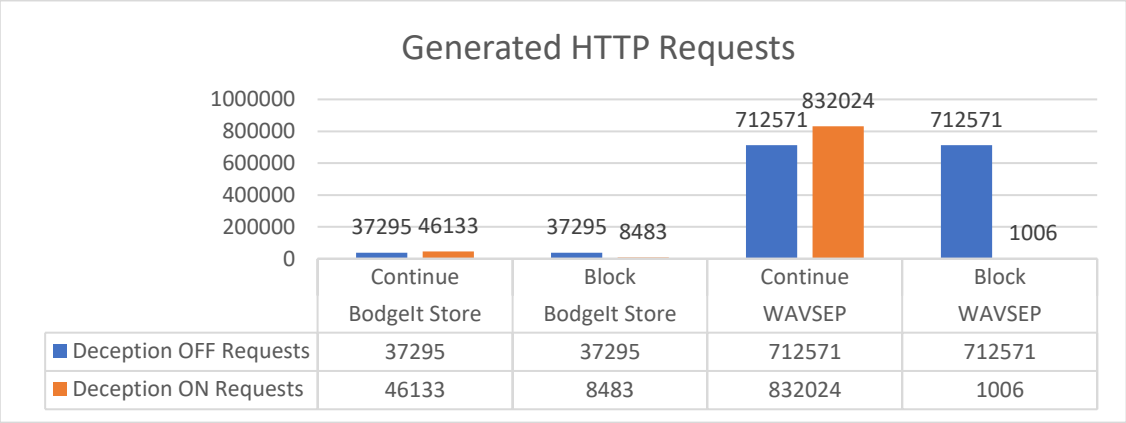


Figure 29. Generated HTTP Request Results

When the deception artifact was enabled in "continue" mode, OWASP ZAP attacks generated a higher number of requests (≈23% and ≈22% more requests). The reason behind this behavior is that the deception artifact was injecting new additional parameters and new fake URLs. Those new elements were detected by OWASP ZAP and were causing a higher amount of new additional requests targeting those new deceptive parameters and new URLs. The number of requests was higher with the WAVSEP application because it is a larger application with hundreds of URLs, whereas BodgeIt is a smaller application with a total of 18 URLs.

The behavior changed when the deception artifact was enabled in "block" mode. The spider module from OWASP ZAP was unable to successfully crawl the website and it was not able to find URLs to launch its tests and attacks against. This caused a significant drop in the number of requests: ≈80% less requests with the BodgeIt application and ≈99% less requests with WAVSEP. The reduction was smaller in BodgeIt because the structure of this application is simpler and pages are interlinked, which allowed the crawler to get information about more URLs with just some initial crawling attempts.

7.2.2. Penetration Testing Time

This variable measures the time needed to perform attacks and it can be directly linked with the economic cost of an attack. Therefore, a higher time would mean a higher needed effort by the attacker to attack the system. The graph below shows and compares the time needed by OWASP ZAP to launch attacks in different scenarios.

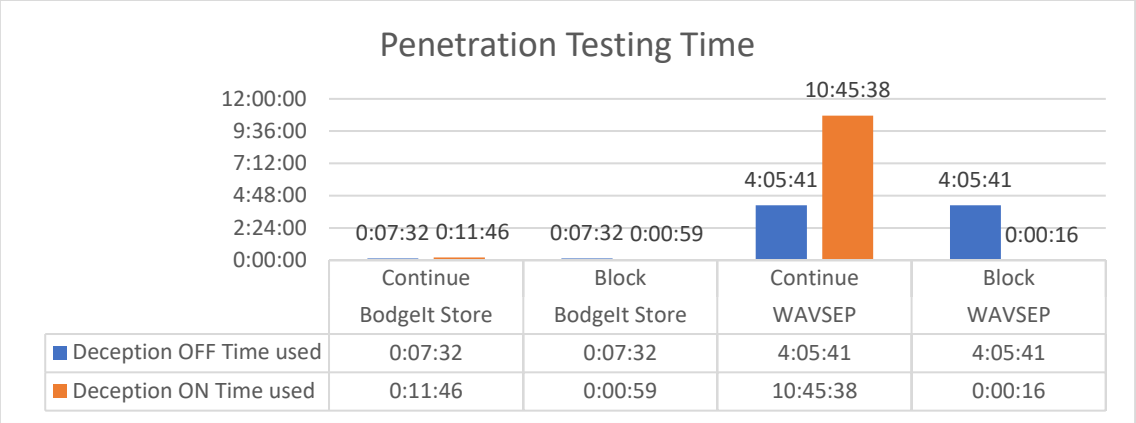


Figure 30. Penetration Testing Time Results

Penetration testing time is directly correlated with the number of requests that the attacking tool generates. When the artifact was configured in continue mode, the time needed to perform the tests was higher, which suggests that the effort made by an attacker would be higher, thus increasing the economic effort and increasing security.

When the artifact was configured in block mode, the penetration testing tool was unable to get enough information and was not able to proceed with the attacks. This also suggests an increased security, as an attacker would retrieve less information from the system to launch application-layer attacks.

7.2.3. Reported Vulnerabilities

Reported vulnerabilities show the number of vulnerabilities found by OWASP ZAP, which means that they were successfully attacked by the OWASP ZAP Active Attack module. A decrease in reported vulnerabilities would indicate that the deception artifact is able to protect the web application from certain types of attacks. The graph below compares the amounts of vulnerabilities reported by OWASP ZAP in different scenarios.

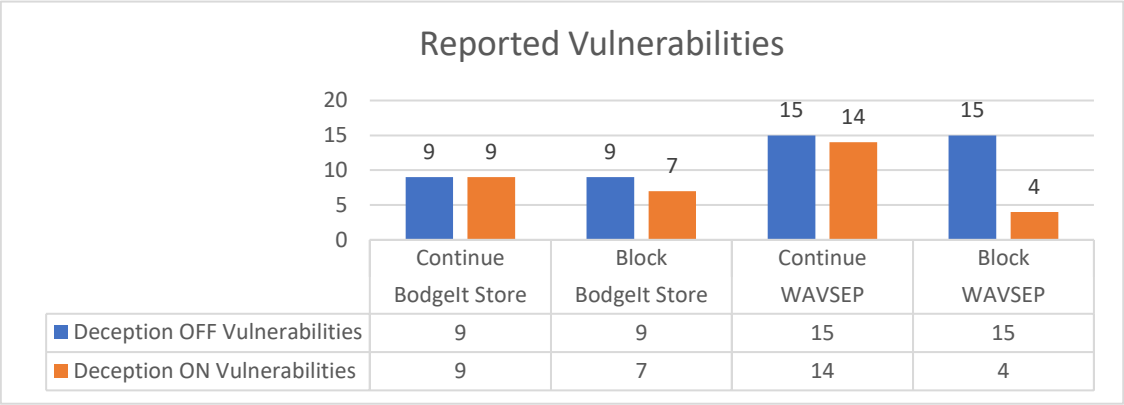


Figure 31. Reported Vulnerabilities Results

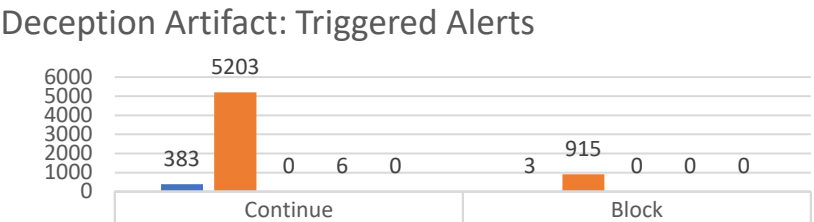
Obtained results showed that OWASP ZAP reported equal or less number of vulnerabilities when the deception artifact was enabled, and the results varied depending on the configuration.

When the artifact was running in continue mode, same number of vulnerabilities were detected in BodgeIt, and one less in WAVSEP.

When the block mode was enabled, the number of detected vulnerabilities decreased drastically in all cases, since OWASP ZAP was getting the blocked response and was not able to succeed with the attacks.

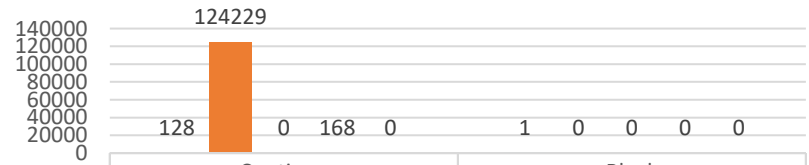
7.2.4. Triggered Alerts

Triggered alerts measure the number of attack attempts detected by the deception artifact implementing the deception strategies. A legitimate user of web applications would never access and modify any fake URL, fake cookie or fake variable injected by the artifact. In consequence, all the alerts are considered as true positives. A higher detection rate would indicate a better security performance. The following figures show the number of alerts triggered by each one of the strategies implemented into the deception artifact under different configuration options.



	Continue BodgeIt Store	Block BodgeIt Store
Strategy 1: Deceptive Comments	383	3
Strategy 2: Deceptive Parameters	5203	915
Strategy 3: Deceptive Cookies	0	0
Strategy 4: Deceptive Status Codes	6	0
Strategy 5: Deceptive JavaScript	0	0

Figure 32. Deception Artifact Alerts – BodgeIt



	Continue WAVSEP	Block WAVSEP
Strategy 1: Deceptive Comments	128	1
Strategy 2: Deceptive Parameters	124229	0
Strategy 3: Deceptive Cookies	0	0
Strategy 4: Deceptive Status Codes	168	0
Strategy 5: Deceptive JavaScript	0	0

Figure 33. Deception Artifact Alerts – WAVSEP

Results indicated that the Deceptive Parameters strategy generated the highest number of alerts by detecting all fake parameter tampering attempts in all cases. This proofs that the deception strategy is suitable to detect attack types based on parameter tampering. Deceptive comments and Deceptive Status Codes also generated a high number of alerts in continue operating modes. Deceptive Status codes didn't generate alerts in block mode, and this is because the blocking rule blocks the responses returning 200 OK responses before the execution of the Deceptive Status Codes strategy, which means that crawling attempts are not detected in this mode.

Deceptive cookies and deceptive JavaScript strategies did not generate any alerts and none of the provided fake cookies and fake deceptive JavaScript elements were detected and used by OWASP ZAP.

7.3. Design Iterations performed

The design of the deception strategies and the implementation of the testing environment were accomplished in two major iterations: an initial iteration and an improvement iteration.

The design did not have specific evaluation criteria to meet at the early stages of the design process, and the results obtained with the first iteration pinpointed some problems and suggested improvements that could be made.

One of the problems in the first iteration was related with the penetration testing tools being used (Aranchi Scanner, Vega and IronWASP). They often crashed and gave inconsistent results when repeating the same tests with exactly the same configuration. Those tools hadn't been updated since 2015 and they might not be actively maintained anymore. This situation suggested that the tools were not suitable for the purpose. OWASP ZAP was the only tool that did not crash, was fast, highly configurable and recently updated (last version from March 2017).

Regarding the deception strategies, the results from the first iteration showed that all strategies except Deceptive Cookies and Deceptive JavaScript performed well detecting attacks. However, the performance in attack prevention did not go so well. Although penetration testing time was increased enabling the deception artifact, reported vulnerabilities remained the same.

Taking those problems and results into account, the design went through a second iteration aiming to improve some of the detected shortcomings. The following table summarizes the most significant changes made for the second iteration.

Table 23. Design Iterations and improvements

Design Iteration Summary	
Problem in Iteration 1	Improvement in Iteration 2
Issues with penetration testing tools: Crashes and inconsistent results.	Remove problematic tools from testing environment and use OWASP ZAP only.
Attack detection: Deceptive Cookies and Deceptive JavaScript could not detect attacks.	Configure harder and deeper penetration tests aiming to trigger more alerts.
Attack prevention: Penetration testing time was increased enabling the deception artifact, but reported vulnerabilities remained the same.	Redesign deception strategies to improve attack prevention capabilities. Add a new "block" configuration mode to the strategies.

Operational performance parameters not being measured.	Redesign testing environment to measure HTTP Round-trip times, CPU and Memory performance variables.

Firstly, problematic penetration scanners were removed from the testing environment and OWASP ZAP was left as the only penetration testing tool used. Secondly, the configuration of OWASP ZAP tests was modified by enabling "HTTP Header" and "Cookie Data Vectors" to make attacks more comprehensive and lastly, deception strategies were redesigned adding a "block" configuration mode to provide a higher flexibility and the possibility to stop attacks.

In addition to those changes, operational performance measures and the python test script were added for the second iteration too.

7.4. Validity of the design

The design of the deception strategies presented in this work aims to be validated by building the testing environment and measuring operational and security performance variables as a proof of the effectiveness of the strategies to detect and stop attacks. In consequence, the validity of the design is tied to the implementation of the testing environment and as a result, the validity of the strategies to protect against human attackers in the real world is open to discussion. Human attackers could develop the ability to recognize when a certain system is returning deceptive responses and could find ways to ignore and neutralize deceptive elements. The opposite could also happen, for example in our testing environment Deceptive Cookies and Deceptive JavaScript did not get good results with OWASP ZAP, but they could have done well against a human attacker.

OWASP ZAP was used to launch attacks in the testing environment and this tool is also used to launch real attacks by human attackers in the real world. Therefore, a degree of validity of the design can be assumed because if the strategies were effective against attacks launched from OWASP ZAP in the testing environment, they could also be effective in the real world because OWASP ZAP is used by human attackers to perform real world attacks.

7.5. Lessons learned

The strategies designed and evaluated in this work are based on a main common principle which is the creation and monitoring of deceptive elements that an attacker would like to access but a legitimate user would never use. This pattern has been applied using different application-layer elements commonly employed in web applications and five examples have been created and evaluated. After the completion of the design iterations, those are the main design principles that have been identified when it comes to the design of successful deception strategies:

- The planning step is the key to develop a successful strategy. The strategic goals that want to be achieved, the attackers' behavior and biases must be comprehensively analyzed and considered.
- Deception must be believable for the attacker. The approach followed in this work to accomplish this has been to integrate deception into real production systems and real traffic

of applications. This approach proved to be satisfactory, but risks of interfering with the original behavior of the application must be identified and assessed.

- Developing strategies with the ability to block attackers are an effective way to prevent attacks.
- Integrating application-layer deception into existing web applications poses an engineering problem that can be solved with satisfactory results. Hooking a deception module inside the same web server that is serving applications gave good results and did not add any significant operational overhead.

7.6. Answer to Research Questions

RQ1. What is the performance of different deception strategies in detecting or preventing web application attacks?

This question has been answered in the Operational Performance and Security Performance sections of this chapter. The operational performance was very good in all the strategies and the tests performed showed no significant overhead in Round-Trip times, CPU and memory use.

In attack detection, Parameter Tampering strategy was the best performing one. Many common attacks such as SQL injection or XSS are based on parameter tampering and therefore the use of deceptive parameters is very efficient detecting them. Deceptive Status Code strategy was also able to detect web crawling attempts. Deceptive Cookies and Deceptive JavaScript strategies did not get good results because OWASP ZAP did not access any deceptive element injected by these strategies.

When it comes to attack prevention, the application of the deception strategies resulted in a higher number of requests and higher penetration testing time. This means that the cost of performing certain attacks got increased, which could decrease the possibility of those attacks to happen. Deceptive Status Code strategy and the block mode configurations were able to prevent a high number of attacks because the attacker was unable to access the application.

Deceptive Cookies and Deceptive JavaScript strategies did not perform well preventing attacks either as OWASP ZAP was not able to trigger any alerts related with these two strategies.

RQ2. How can a testing platform be developed in order to evaluate the performance of those deception strategies?

This question is implicitly answered in Chapter 6 and the design of the testing platform is detailed throughout the whole chapter.

CHAPTER EIGHT

8. CONCLUSIONS AND FUTURE WORK

The popularity of the internet has made the use of web applications ubiquitous and essential to our daily lives. Web servers and applications have become a valuable target for attackers and proof of this are the large volume of application-layer attacks that are succeeding daily. Existing misuse and anomaly-based detection and prevention techniques fail to cope with the volume and sophistication of new attacks that are continuously appearing, which suggests that there is a need to provide new additional layers of protection.

The use of deception for computer security defense is a concept that emerged with the popularization of honeypots in the early 90s but despite the time that has passed since then, its use still seems to be usually limited to ad-hoc implementations and isolated or repackaged systems such as honeypots. Although there are models and frameworks to plan and integrate deception into production systems from a theoretical perspective, there is a lack of works that are actually applying those models and frameworks to develop and implement deceptive strategies for real use cases.

Motivated by this situation, this work focused its attention on the application-layer of web applications and aimed to explore possible usages of deception that could be integrated into the application-layer of web applications as an additional layer of defense. Five new application-layer strategies were developed: Deceptive Comments, Deceptive Parameters, Deceptive Cookies, Deceptive Status Codes and Deceptive JavaScript. Their ability to detect and prevent web application attacks was measured in a testing environment that was also built for the purpose.

Several experiments were conducted to measure both operational and security performance variables of the deception artifact. Measured operational variables included the average HTTP request round-trip times, CPU and memory consumption. Concerning the security performance variables, the number of requests generated by OWASP ZAP, the time needed to perform the attacks, the number of vulnerabilities detected and the number of alerts triggered by the deception artifact were measured.

According to the executed experiments, Deceptive Parameter strategy got the best security performance results, followed by Deceptive Comments and Deceptive Status Codes. These strategies were able to detect all the attacks that were accessing and modifying deceptive elements. As legitimate users would never access those elements, the existence of false positives was discarded. These strategies also increased the penetration testing time needed to perform the attacks, thus incrementing the economic cost of carrying them out. When operating in block mode the deception artifact also increased the security of our web applications by preventing attacks, as OWASP ZAP was unable to continue with its attack procedures.

Deceptive Cookies and Deceptive JavaScript obtained the poorest security performance results since OWASP ZAP was unable to detect and use deceptive elements generated by these strategies. Alerts related to Deceptive Cookies and Deceptive JavaScript were only triggered by

the custom functional test script developed to verify the correctness of the implementation. Concerning the operational performance, results showed that the deception artifact could successfully be integrated with existing web applications without changing their source code and adding a very low operational overhead using an implementation based on Java Servlet Filters.

These results confirm that deception provides a useful layer of protection that can successfully be integrated into different layers of production systems that are susceptible of suffering from attacks such as the application-layer of web applications.

One of the limitations that this work might present is that the proposed deception approaches could be effective in the testing environment but may not perform well when having a human attacker. Therefore, we could say that one of the main problems of the use of deception is that it must be believable for human attackers in the long run and achieving this poses several challenges as strategies must be continuously believable and adaptive.

A future possibility to explore could be related with the use of machine learning techniques that could bring opportunities to develop models with the ability to learn from real application traffic and user behavior in order to create automatic deceptive application-layer elements that are similar to the real traffic without the need to configure or hardcode any predetermined deceptive content. These advanced and adaptive deception strategies could also be tested against human attackers instead of using automated tools to measure their effectiveness in the real world.

The landscape of computer security threats is worrying as there is an increasing volume of sophisticated attacks that current misuse and anomaly-based detection and prevention techniques are not able to defend against. The deception strategies proposed in this work show that deception is a valuable tool that has potential as an additional layer of protection to complement current techniques. Despite deception being an old concept, recent works from 2016 and 2017 providing theoretical frameworks and examples [6] [21] [32] confirm the interest and relevance of the topic nowadays. Lately, some commercial companies and startups like Cymmetria [65] or Illusive Networks [66] have emerged, claiming to use novel deceptive mechanisms in their security products. The interest of the industry in this field could also be a sign of the potential that deception could bring to improve security defense.

Taking everything into consideration, we could state that despite being an old concept, deception is a field that still has an enormous potential to be more studied, developed and applied in the future and doing so could bring interesting opportunities to improve the effectiveness of existing computer security defense systems.

REFERENCES

[1] S. Prandl, M. Lazarescu and D.-S. Pham, "A study of web application firewall solutions," in *International Conference on Information Systems Security*, Kolkata, India, 2015.

[2] OWASP Foundation, "OWASP Top-10 2013," 2013. [Online]. Available: https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf. [Accessed June 2017].

[3] OWASP Foundation, "OWASP Top-10 2017 (Release Candidate 1)," 2017. [Online]. Available: https://github.com/OWASP/Top10/raw/master/2017/OWASP%20Top%2010%20-%202017%20RC1-English.pdf. [Accessed June 2017].

[4] V. Prokhorenko, K.-K. R. Choo and H. Ashman, "Web application protection techniques: A taxonomy," *Journal of Network and Computer Applications,* no. 60, pp. 95-112, 2016.

[5] G. Deepa and P. S. Thilagam, "Securing web applications from injection and logic vulnerabilities: Approaches and challenges," *Information and Software Technology,* no. 74, pp. 160-180, 2016.

[6] M. H. Almeshekah and E. H. Spafford, “Cyber Security Deception,” in *Cyber Deception: Building the Scientific Foundation*, Cham, Springer International Publishing, 2016, pp. 23-50.

[7] N. Virvilis, B. Vanautgaerden and O. S. Serrano, "Changing the game: The art of deceiving sophisticated attackers," in *6th International Conference On Cyber Conflict (CyCon 2014)*, Tallinn, Estonia, 2014.

[8] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *2010 IEEE Symposium on Security and Privacy (SP)*, Oakland, CA, USA, 2010.

[9] T. a. G. C. a. R. K. a. L. P. Krueger, "TokDoc: A self-healing web application firewall," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, Sierre, Switzerland, 2010.

[10] IETF, "Hypertext Transfer Protocol -- HTTP/1.1," 1999. [Online]. Available: https://tools.ietf.org/html/rfc2616. [Accessed June 2017].

[11] J. J. Stephan, S. D. Mohammed and M. K. Abbas, "Neural Network Approach to Web Application Protection," *International Journal of Information and Education Technology,* vol. 5, no. 2, pp. 150-155, 2015.

[12] A. Moosa, "Artificial neural network based web application firewall for sql injection," *International Journal of Computer, Electrical, Automation, Control and Information Engineering,* no. 64, pp. 610-619, 2010.

[13] M. Kakavand, N. Mustapha, A. Mustapha, M. T. Abdullah and H. Riahi, "Issues and Challenges in Anomaly Intrusion Detection for HTTP Web Services," *Journal of Computer Science,* vol. 11, no. 11, p. 1041, 2015.

[14] Z. Dewa and L. A. Maglaras, "Data Mining and Intrusion Detection Systems," *International Journal of Advanced Computer Science and Applications,* vol. 7, no. 1, 2016.

[15] M. M. Rathore, A. Ahmad and A. Paul, "Real time intrusion detection system for ultra-high-speed big data environments," *The Journal of Supercomputing,* vol. 72, no. 9, pp. 3489-3510, 2016.

[16] H. T. Nguyen, C. Torrano-Gimenez, G. Alvarez, K. Franke and Petrović, "Enhancing the effectiveness of web application firewalls by generic feature selection," *Logic Journal of IGPL,* vol. 21, no. 4, pp. 560-570, 2013.

[17] M. Ahmed, A. N. Mahmood and J. Hu, "A survey of network anomaly detection techniques," *Journal of Network and Computer Applications,* vol. 60, pp. 19-31, 2016.

[18] "ModSecurity," [Online]. Available: https://www.modsecurity.org. [Accessed June 2017].

[19] J. Webster and R. T. Watson, "Analyzing the past to prepare for the future: Writing a literature review," *MIS quarterly,* pp. xiii--xxiii, 2002.

[20] M. Bercovitch, M. Renford, L. Hasson, A. Shabtai, L. Rokach and Y. Elovici, "HoneyGen: An automated honeytokens generator," in *2011 IEEE International Conference on Intelligence and Security Informatics (ISI)*, Beijing, China, 2011.

[21] C. De Faveri and A. Moreira, "Designing Adaptive Deception Strategies," in *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Vienna, Austria, 2016.

[22] C. Stoll, The cuckoo's egg: Tracing a spy through the maze of computer espionage, New York, NY, USA: Doubleday, 1989.

[23] B. Cheswick, "An Evening with Berferd in which a cracker is Lured, Endured, and Studied," in *Proc. Winter USENIX Conference*, San Francisco, CA, USA, 1992.

[24] L. Spitzner, "Honeypots: Catching the insider threat," in *Proceedings of the 19th Annual Computer Security Applications Conference*, Las Vegas, NV, USA, 2003.

[25] B. M. Bowen, V. P. Kemerlis, P. Prabhu, A. D. Keromytis and S. J. Stolfo, "A system for generating and injecting indistinguishable network decoys," *Journal of Computer Security,* vol. 20, no. 2-3, pp. 199-221, 2012.

[26] F. Cohen, D. Lambert, C. Preston, N. Berry, C. Stewart and E. Thomas, "A framework for deception," in *National Security Issues in Science, Law, and Technology*, CRC Press, 2007, pp. 123-219.

[27] T. E. Carroll and D. Grosu, "A game theoretic investigation of deception in network security," *Security and Communication Networks,* vol. 4, no. 10, pp. 1162-1172, 2011.

[28] N. Garg and D. Grosu, "Deception in honeynets: A game-theoretic analysis," in *Information Assurance and Security Workshop, 2007. IAW'07. IEEE SMC*, West Point, NY, USA, 2007.

[29] N. Rowe, "Planning cost-effective deceptive resource denial in defense to cyber-attacks," in *Proceedings of the 2nd International Conference on Information Warfare & Security*, Monterrey, CA, USA, 2007.

[30] C. Katsinis and B. Kumar, "A Framework for Intrusion Deception on Web Servers," in *2013 International Conference on Internet Computing, ICOMP'13*, Las Vegas, NV, USA, 2013.

[31] M. H. Almeshekah, *Using deception to enhance security: A Taxonomy, Model, and Novel Uses,* Ph.D. dissertation, Purdue University, Lafayette, Indiana, 2015.

[32] T. Ishikawa and K. Sakurai, "Parameter manipulation attack prevention and detection by using web application deception proxy," in *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, Beppu, Japan, 2017.

[33] S. M. Srinivasan and R. S. Sangwan, "Web App Security: A Comparison and Categorization of Testing Frameworks," *IEEE Software,* vol. 34, no. 1, pp. 99-102, January 2017.

[34] Y. Makino and V. Klyuev, "Evaluation of web vulnerability scanners," in *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Warsaw, Poland, 2015.

[35] N. I. Daud, K. A. A. Bakar and M. S. M. Hasan, "A case study on web application vulnerability scanning tools," in *Science and Information Conference (SAI)*, London, United Kingdom, 2014.

[36] N. Suteva, D. Zlatkovski and A. Mileva, "Evaluation and testing of several free/open source web vulnerability scanners," in *Proceedings of the 10th Conference for Informatics and Information Technology (CIIT 2013)*, Bitola, Macedonia, 2013.

[37] C. Joshi and U. K. Singh, "Performance Evaluation of Web Application Security Scanners for More Effective Defense," *International Journal of Scientific and Research Publications (IJSRP),* vol. 6, no. 6, pp. 660-667, 2016.

[38] F. R. Muñoz, E. A. A. Vega and L. J. G. Villalba, "Analyzing the traffic of penetration testing tools with an IDS," *The Journal of Supercomputing,* pp. 1-16, 2016.

[39] F. A. Saeed, "Using WASSEC to Analysis and Evaluate Open Source Web Application Security Scanners," *International Journal of Computer Science and Network,* vol. 3, no. 2, pp. 43-49, 2014.

[40] Web Application Security Consortium, "Web Application Security Scanner Evaluation Criteria," [Online]. Available: http://projects.webappsec.org/f/Web+Application+Security+Scanner+Evaluation+Criteria+-+Version+1.0.pdf. [Accessed June 2017].

[41] F. A. Saeed, "Using WASSEC to Evaluate Commercial Web Application Security Scanners," *International Journal of Soft Computing and Engineering (IJSCE),* vol. 4, no. 1, pp. 177-181, 2014.

[42] "Acunetix Web Vulnerability Scanner," [Online]. Available: http://www.acunetix.com/vulnerability-scanner/. [Accessed June 2017].

[43] "Arachni Web Application Security Scanner Framework," [Online]. Available: http://www.arachni-scanner.com/. [Accessed June 2017].

[44] "Burp Suite," [Online]. Available: https://portswigger.net/burp. [Accessed June 2017].

[45] "Skipfish Web application security scanner," [Online]. Available: https://github.com/spinkham/skipfish. [Accessed February 2017].

[46] "HP WebInspect," [Online]. Available: http://www8.hp.com/us/en/software-solutions/webinspect-dynamic-analysis-dast/. [Accessed June 2017].

[47] "IronWASP - Iron Web application Advanced Security testing Platform," [Online]. Available: http://ironwasp.org/. [Accessed June 2017].

[48] "Nessus Vulnerability Scanner," [Online]. Available: https://www.tenable.com/products/nessus-vulnerability-scanner. [Accessed June 2017].

[49] "Netsparker Website Vulnerability Scanner & Security Tool," [Online]. Available: https://www.netsparker.com/web-vulnerability-scanner/. [Accessed June 2017].

[50] "N-Stalker - Free Edition," [Online]. Available: http://www.nstalker.com/products/editions/free/. [Accessed June 2017].

[51] "OWASP Zed Attack Proxy Project," [Online]. Available: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project. [Accessed June 2017].

[52] "Vega Vulnerability Scanner," [Online]. Available: https://subgraph.com/vega/. [Accessed June 2017].

[53] "Web Application Attack and Audit Framework," [Online]. Available: http://w3af.org. [Accessed June 2017].

[54] "Damn Vulnerable Web Application (DVWA)," [Online]. Available: http://www.dvwa.co.uk/. [Accessed June 2017].

[55] "WackoPicko Vulnerable Website," [Online]. Available: https://github.com/adamdoupe/WackoPicko. [Accessed June 2017].

[56] "The Web Application Vulnerability Scanner Evaluation Project," [Online]. Available: https://github.com/sectooladdict/wavsep. [Accessed June 2017].

[57] OWASP Foundation, "OWASP Vulnerable Web Applications Directory Project - Offline Apps," [Online]. Available: https://www.owasp.org/index.php/OWASP_Vulnerable_Web_Applications_Directory_Project/Pages/Offline. [Accessed June 2017].

[58] R. H. Von Alan, S. T. March, J. Park and S. Ram, "Design science in information systems research," *MIS quarterly,* vol. 28, no. 1, pp. 75-105, 2004.

[59] K. Peffers, T. Tuunanen and M. A. a. C. S. Rothenberger, "A design science research methodology for information systems research," *Journal of management information systems,* vol. 24, no. 3, pp. 45-77, 2007.

[60] T. Mettler, M. Eurich and R. Winter, "On the use of experiments in design science research: a proposition of an evaluation framework," *Communications of the Association for Information Systems,* vol. 34, no. 1, pp. 223-240, 2014.

[61] "BodgeIt Store Vulnerable Web Application," [Online]. Available: https://github.com/psiinon/bodgeit. [Accessed June 2017].

[62] "Python Official Website," [Online]. Available: https://www.python.org/. [Accessed June 2017].

[63] "Requests Python Library 2.14.2 Documentation," [Online]. Available: http://docs.python-requests.org/en/latest/api/. [Accessed June 2017].

[64] "NumPy Library," [Online]. Available: http://www.numpy.org/. [Accessed June 2017].

[65] "Cymmetria," [Online]. Available: https://cymmetria.com/. [Accessed June 2017].

[66] "Illusive Networks," [Online]. Available: https://www.illusivenetworks.com. [Accessed June 2017].