Universiteit van Amsterdam

UNIFACE

Master's Thesis

Software Engineering

Effectiveness of Automated Security Analysis using a Uniface-like Architecture

Author:
Job P. Jonkergouw

Company Supervisor:
Ming Zhu

University Supervisor:
Magiel Bruntink

Internship location:
Uniface B.V., Amsterdam

July 8, 2014

"The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards - and even then I have my doubts."

Gene Spafford
Computer security expert

Abstract

This thesis examines the difference between using two types of automated security analysis of a web application: static analysis scans the source code while dynamic analysis simulates cyber-attacks. A specially crafted web application, designed in the Uniface framework, was developed that contains deliberately injected vulnerabilities. Multiple analysis tools were tested for their ability to discover these vulnerabilities. Few differences were observed between the tools that use static analysis and those that use dynamic analysis.

Contents

Chapter 1

Introduction

Ever since the introduction of the world wide web, websites have been plagued with cyber-attacks. Government websites are defaced, multinational cooperations have their customers' credit card numbers stolen and politicians get their emails leaked. Headlines on major security breaches reach the newspapers regularly. In the last few months the world witnessed eBay having to reset over a hundred million passwords [28] and Microsoft faced a serious weakness in Internet Explorer. Also widely publicized are security issues such as the recent OpenSSL 'Heartbleed' and last year's discovery of the NSA's surveillance program. It is clear that in this day and age getting our data and internet security is a hot topic.

One major point of failure in security are the websites. Neatly called *web applications*, they make up one of the most popular interfaces between a user and another machine. The communication is largely handled by a simple protocol known as HTTP, *Hypertext Transfer Protocol*. This protocol is augmented by a myriad of technologies, sometimes developed organically without much planning. To support these technologies, browsers have been racing to keep up and accept haphazardly designed standards. As a result, what is left is bad standardization (such as browser specific CSS tags and unpredictable JavaScript behavior) and perhaps worse: an inherent lack of security.

The previous narrative can be explained by a constant struggle between opposing software requirements: on one side security on the other side usability and flexibility. With the latter ones winning, it is left to the developers to secure their web application. However, this is not an easy task as many things can and will go wrong. The developers do not only have their own code to worry about, but also external technologies which can contain bugs in the APIs or security holes in software (such as the recent Heartbleed bug). And while these can be patched, this will offer little help against zero-day (i.e. undiscovered) attacks or languages like JavaScript or CSS that contain inherent security issues. Furthermore, not all development teams have the expertise or the time to worry about security. On a more fundamental note: secure software is like a secure house: you can lock 99 of your 100 windows, but forgetting even 1 window will allow a burglar to slip in. Just so in software: only one bug in a 100,000 lines of code could allow an attacker to get in.

Despite these challenges, multiple software tools have been released over the years to assist developers in fighting security holes. These *security analysis tools* allow developers to scan their application or code for vulnerabilities. Several types of tools exist, some are better in detecting certain security holes than others. The major distinction is made between *static* analysis tools, which scan the source code, and *dynamic* analysis tools, which probe website front ends for security holes. Most major analysis tools are one or the other.

Most of the commercial security analysis tools can be costly, so it is important the determine the right tool to use. This was also an issue raised by the developers of the *Uniface* software development framework. In an attempt look into it, this thesis we will look into the advantages and disadvantages of using different analysis tools: which one finds the most vulnerabilities in a Uniface application.

To execute the previously mentioned research, this thesis will first look into the different types of security holes and what techniques exist for finding these in the chapter thereafter. Next, an experiment is presented that involves injecting security holes on purpose in a Uniface web application. This will be explained by first examining the Uniface architecture, the experimental setup and which vulnerabilities are examined. The next step is scanning the application, which includes studying the different tools used and analyzing the experimental data. In the end, the consequences of the data will be discussed and how they influence the choice of analysis tool.

Chapter 2

Background

This chapter examines the root of this thesis: what are web application vulnerabilities and how are they detected? To answer this, one needs to understand how securing a web application works. Doing so makes it easier to notice what can be *wrong* in a web application. The resulting *vulnerabilities* can then be spotted using automatic analysis techniques, of which there exist two important types. What the effective differences between these two are the main topic of this thesis and will be examined in later chapters.

2.1 What is web application security?

Knowing how to secure a web application means knowing what to defend against. Threats are called *cyber-attacks* and are, depending on the intent, performed by 'hackers', 'crackers' or more abstractly, 'intruders'. Attacks can also be executed involuntary through the use of botnets. However, the origin is of no concern to this thesis as a secure web application should be safe from anyone. Hence the more neutral *attackers* is used from here on.

From the point of view of an attacker, attacks can be described by the method through which they are carried out, called the *attack vector*. Examples are e-mail attachments, JavaScript in malicious websites, files uploaded by a form or backdoors in the application code. Attacks often try to insert a piece of data or code into the web application, known as the *payload*. This can be a SQL-query, dangerous JavaScrip code or even a virus. The weak spot in the web application where the attack is launched is called the *entry point*.

From the application's point of view, the entry point is a flaw or weakness in the system. Carl Landwehr and others [4] define this as a type of bug: a part of the program that violates a security requirement. In [2], this is criticized by the fact that attacks like a 'buffer-overflow' do not always correspond to a requirement. Other authors and institutes apply a wide range of definitions and terminology: Lindqvist and Jonsson [5] distinguish the attack that exploits a certain security *flaw* or *vulnerability*, while the resulting *breach* in the system is the violation of a security requirement. Meanwhile, microsoft [30] defines a vulnerability as a weakness in a product that compromises one of the *CIA* principles (see next section). The authors Hoglund and McGraw make the following subdivision [18]: a bug is an implementation problem such as a bad line of code while a *flaw* is a design issue. The resulting *vulnerability* is a security bug or flaw that can be exploited by an attacker. Others distinguish [7] between threats and vulnerabilities, unmitigated *treats*. To conclude, this thesis uses *vulnerability* as a *security* weakness, which can be caused by either a *security* bug during implementation or *flaw* during design.

Types of vulnerabilities

One of the earliest attempts to classify security flaws was done in 1970s as part of the *Research Into Secure Operating System* (RISOS) project [19][p.175]. Some of the categories included "incomplete parameter validation" which reminisces of SQL injection, "implicit sharing of confidential data" and "inadequate authentication", all still relevant today. Later research [4] during the 1990s attempts to separate flaws that entered the software system intentionally, such as backdoors, from those that were added accidentally. In addition, flaws were divided according to their time of introduction: during development or during maintenance. However, neither divisions are really useful for this thesis as they describe the origin, not how they manifest themselves. In the same decade, research by Aslam, Krsul and Spafford [3] classifies flaws as *coding faults* or *emergent faults*. The former often include validation or synchronization errors, the latter are subdivided into configuration faults and environmental faults, such as browser or OS settings. Worth mentioning is that they hinted to one of the earlier uses of static security analysis by noting that configuration errors could be checked automatically.

Of course, the entire landscape changed with the advent of the internet. Therefore, there are newer taxonomies such as STRIDE, developed at Microsoft [31]. The acronym stand for *spoofing identity*, *tampering with data*, *non-repudiation* (illegal and untraceable modification of data), *information disclosure*, *denial-of-service* and *elevation of privilege* (gaining privileged access such as admin rights). Another modern classification is made by Tsipenyuk, Chess and Gary [6], which is used in the work on static analysis by Chess and West [16]. These consist of seven kingdoms including common types as *input validation*, *security features* like encryption and *error handling* which can leak information to attackers. But also *API abuse* by failing the API's specification, *time and state* by modifying the order of requests, *code* *quality* and *encapsulation*.

A widely used security model are the *CIA* principles: a web application should provide *confidentiality*, meaning it should hide the information and communication of a user from others. It should conform to *integrity*, meaning data can be proven to be secretly intercepted and modified. And finally, *availability* must be provided, guaranteeing access to the system and its data. Using this, vulnerabilities can be categorized by which principle they impair. For example, a *denial-of-service* attack violates availability.

Modern vulnerability databases and lists

As mentioned earlier, knowledge about web application security requires knowledge about different security vulnerabilities. Unfortunately, they are numerous and new types keep appearing each day. This necessitates the existence up-to-date databases for looking up the latest flaw or bug. One prominent example is the *Common Weakness Enumeration* [21] maintained by the MITRE institute which systematically lists commonly found types of security weaknesses. This database is linked to the *Common Vulnerability Enumeration* [22] which lists such weaknesses found in actual software systems.

As the entries in the databases are plentiful, there are publications that list the most frequent, dangerous and therefore the most relevant vulnerabilities for developers. One used by many organizations [29] is the *OWASP top 10* released every few years by the non-profit organization OWASP. Several of the security analysis tools discussed later on also refer to this list. It is shown in table A.1 of the appendix.

Another vulnerability listing is the CWE/SANS top 25, published in 2011 as a collaboration between the SANS and the MITRE institutes, shown in table A.2 of the appendix. The list is headed by some of the same vulnerabilities as the OWASP top-10, such as SQL injection and ‘cross-site scripting’ (XSS), and are ranked according to their CWE scoring. The main difference between the two lists lies in how the vulnerabilities are scoped and specified. For example, ‘missing encryption’ and ‘hard-coded credentials’ could be considered as part of ‘security misconfiguration’.

Basic security controls

Web applications are protected using a set of defensive techniques known as *controls*. One of the first steps is detecting any potential attack using *intrusion detection*, the ability to log suspicious activity and deploying counter measures if possible, such as closing the offending user account.

Other important controls are *authentication* and *authorization*. Authentication are techniques to ensure that users are who they say they are. This is usually enforced by a log-in system using a user name and password, but a successful implementation depends on multiple factors such as encryption, proper session management and database storage. Authorization is deciding which user is allowed to perform which action and is often implement by using different user roles.

Interaction between the server and the user often requires several in and output steps to the database. *Input validation* should be performed to check if the user provided valid data and should be *sanitized* of any control characters to prevent cross-site scripting or SQL injection. This is an important necessity as user input should never be trusted [23]. Likewise, interconnected software components should check each others input to prevent them from compromising each other.

Many developers enable debugging options and richly annotated error messages to assist them during development, but such options should be disabled for any live deployment of the web application. Error should not disclose any details about the server as this information may assist attackers.

2.2 Automated Security Analysis

Although the set of basic controls mentioned previously are well recommended, they cannot ensure a web application to be secure. As simply put by Howard and Leblanc in [17][p.27], *"Security features do not equal secure features"*. Security is an emergent feature of a software system [1], reactive measures are not enough. It is not a property that can be *"spray-painted on a system"* [1], but must be considered during the entire development life cycle.

Several models have been proposed that include security analysis during each stage of development [14]. Examples are *BSIMM*, Microsoft's *MS SDL* and OWASP's *SAMM*, which all cover roughly the same steps. Like with many software processes, these include steps for requirement specification, which should in security requirements and their respective use-cases (including 'threat-modeling'). The architecture and design should be reviewed for security and security test cases have to be included. More outstanding is the insistence of educating the developers and the maintenance of a strict security policy.

Despite all these steps it is difficult to gauge how secure a web application actually is; traditional test cases are limited to the imagination of the requirement engineers and the testers. Therefore two steps that also have to be included in the development process are *code review* and *penetration testing*.

Manual penetration testing is hard and requires expertise not always available to developers [14]. It involves launching hacker-like attacks on the application and demands up-to-date knowledge on modern hacking techniques. Fortunately, this can be automated by using automated scanning techniques called *dynamic application security testing* (DAST) also known as *dynamic security analysis*. In a similar fashion security code reviews can be automated by using *static application security testing* (SAST). As the dynamic analysis does not look at the source code while static analysis does, they are sometimes contrastingly called 'black-box testing' and 'white-box testing' respectively.

2.2.1 Static analysis

In principal, a static analyzer tries to understand and analyze the source code and design [1]. These can be simple bugs by the programmer or deeper design flaws. Depending on the ingenuity of its creators a static analysis tool can offer a diverse range of detectable vulnerabilities. However, all static analyzers are limited by one theoretical constraint: Rice's theorem states that it is not possible to fully compute a non-trivial property of an algorithm, without executing the algorithm itself. Therefore, the property `contains-vulnerability()` can only be computed by a static analyzer by using an approximation of this property. As a result, all static analyzers are intrinsically inclined to contain false positives. The only exception are trivial properties such as regular expression matches. One such example seen later in this thesis is a hard code credential (`this.password = password`).

Techniques for static analysis

Multiple techniques can be employed by a static analysis tool. Some of these are well-suited for security analysis while other are more appropriate for other types of code analysis. According to the book "Secure Programming with Static Analysis" [16], the following types can be distinguished:

Perhaps the most basic technique uses a search function or regular expressions to search for certain keywords. For example, the C-function `strcpy()` is considered dangerous nowadays. This technique is some called 'grepping', named after the Unix utility 'grep'.

Another simple technique is *type checking* any variables or functions by looking at its declaration. This allows rules to exists concerning dangerous data types, such as arrays that could lead to buffer overflows. A more complex method*style checking*, which can check matching brackets, inconsistent whitespace usage or comments that could be signs of bad coding. Today, both types are often included in modern text editors and are often taken for granted by current-day programmers.

Some of the more advanced tools and IDEs offer *program understanding* capabilities for their code checking. For example, the IDE Eclipse can scan call stack of a certain method or check its inheritance. For security purposes, one could check where vital objects such as those for HTTP requests and responses are accessed.

A more mathematical approach is using *program verification* scanning. Methods and functions can be written to a certain specification, which can be checked by an automated analysis tool. If only a part of the behavior is checked, it is known as *property checking*. One of its subtypes is *temporal safety properties*, which checks the sequence of statements. This can be used to ensure proper memory management in C to prevent buffer overflows. Examples include the checking of properly allocated and freed memory in C.

Another popular static analysis type is called simply *bug finding* by the authors of [16]. It looks for patterns that will likely behave differently from the programmers intention. One such example would be assignments(=) instead of comparisons(==) in C if-statements.

Static analyzers that scan for security vulnerabilities often apply several of the aforementioned scanning types. The earliest scanners used style checking techniques to find functions with known risks. More modern systems also make use of bug finding-like techniques and property checking.

2.2.2 Dynamic analysis

Dynamic analysis simulates test actual application behavior. Ultimately, this is a more realistic approach than static analysis. However, to find all vulnerabilities that an attacker might exploit the tool has to be at least as smart as the attacker. Static analysis avoids this arms race by challenging a resource that the attacker does not have: the source code.

For internet security, there are different types of dynamic security tools [10]. *Network security* tools scan all machines connected to a network such as PC's, printers, routers and firewalls. One way to do this is by *port scanning* all devices connected to a single machine. More specialized are dedicated *database security* tools, which directly scans for vulnerabilities in the database management system (DBMS), or other types of *security subsystem tools*. This thesis, however, examines security tools used for *web applications*, i.e. websites. Actual website attack are simulated, but plenty other types of techniques are used to cover a wide range of vulnerabilities.

Techniques for dynamic analysis

Basic dynamic analysis of a web application is divided into two parts [8]: *passive mode* where the tool maps the application and determines the entry points in terms of the URLs that can be tested. Automated web crawlers are used to search any webpage for URLs to new webpages. The application can also examined manually with web browser. The testing tool then records each accessed page by running a local proxy server that intercepts the traffic from the browser. Often this step also includes several other attempts at *information gathering*, such as analyzing error codes, determining the server type and version (fingerprinting) or searching the application on Google.

The second part is *active scanning* which consists of multiple steps depending on the tool used. OWASP [8] divides each step into several types, based on the type of security control that is used (see 2.1).

The first step is listed as *configuration management testing*, searching for weaknesses in the server setup. If HTTPS is used, the SSL version and algorithm is examined, but also unused files are browsed or the admin interfaces are examined. A tool might also try to send HTTP request to the server with exotic verbs such as TRACE or PUT instead of the usual GET and POST.

Authentication can be tested by guessing or brute-forcing login information or abusing password reset options. And *authorization* might be checked by attempting path traversal, navigating to hidden files in directory structure, or methods to access pages of other users by changing the URL. Also related is testing the *session management* by examining the cookie properties or trying to edit te session token.

Seen in most tools is *input validation*. Attempts are made at SQL injection, one of the most dangerous vulnerabilities, cross-site scripting or any other injection type. Often some of the input is *fuzzed*, meaning random characters are given to observe how the server reacts.

Another important step is denial-of-service testing. User accounts are often locked after multiple bad log in attempts, and a hacker might abuse this by locking all user accounts. Other attacks can involve storing to much data in a session.

More exotic methods are testing the AJAX , web services, or even testing the actual bussiness logic of the application

2.2.3 Important differences

Apart from the detected results, there are other practical differences between static and dynamic analysis that have to be considered. For instance, both techniques have different limits on when they can be executed:

- Dynamic analysis can be run without access to the source code.
- Static analysis can be run during the earliest stages of development when no executable version of the application is available.

Both techniques also differ in the limits of what they can scan:

- Dynamic analysis can have trouble navigating certain types of web pages such as AJAX requests, limiting the scanning range
- Many web applications combine multiple technologies, but not all can be understood by each static analyzer.

They also differ in how many and which vulnerabilities can be discovered:

- It is much harder to be certain of a vulnerability in static analysis resulting in more false positives.
- Static scanners might miss issues related to the interaction between different components.
- Dynamic analysis cannot see any of the inner workings of the application. E.g. flaws at the business logic side will never be found.

Finally, both techniques have to present their results in different ways:

- Static analysis results directly point to the source code that needs to be fixed.
- Dynamic analysis results are easier to relate to a concrete type of attack.

One major comparison that is left is what vulnerabilities both techniques can find. Is one analysis type able to find flaws that the other type does not?

2.2.4 Modern analysis software

Security analysis tools exists in both open source as well as commercial varieties. Nowadays, an increasing number of commercial tools are being offered as *software-as-a-service* (SaaS) instead of *software-as-a-product* (SaaP), meaning the tool is run from a web application instead of on the user's machine. This has the advantage of not requiring any installation and allowing the tool's own analyst a look at the results. A disadvantage is that this requires your application to be available from the outside world.

Each year, Gartner Inc. releases a comparative research report on different dynamic and static commercial security tools of which the latest version can be found in [25]. In this publication, the authors Neil MacDonald and Joseph Feiman analyze the advantages and disadvantages of each tool. Unfortunately, little is offered on how their metrics were calculated, except for a few general metrics. Also no open-source tools are compared as they are deemed insufficient for full enterprise employment.

A more numerical-based report by Shay Chen can be found at [27], focusing on only dynamic scanners. Prices are shown for commercial products but it also features many open source tools. It features measures such as the number of detected vulnerabilities on a tested web application called 'WAVSEP'.

Chapter 3

Research Question & Context

The previous chapter already mentioned some differences between static and dynamic analysis, such as the number of false positives. However, one major practical difference is which vulnerability is detectable by which technique. This is influenced by two factors mentioned earlier: dynamic analysis can only detect issues on the surface (i.e. limited depth), while static analysis can not always scan the entire range of components (i.e. limited breadth). For example, dynamic tools can not find any hard-coded password assignment while static analysis has trouble measuring many implementation flaws, such as forgotten access controls. To examine the extend of this effect an experiment is needed.

Before a proper experiment can be defined one needs to define a goal for the research. First, previous studies need to be considered to know which issues are relevant and to prevent redundant research. After that, the precise research questions can be asked. Finally the context of the research needs to be determined, as web applications come in many forms and there is no default application. For this experiment, a specially developed version of the Uniface framework was used to create the web application.

3.1 Previous research

The idea of intentionally inserting vulnerabilities into a web application has been done several times before. There exist numerous open source web applications with that in mind, although none using Uniface. These website are often called *vulnerable web applications* and can be used to practice the skills of a hacker, or test the performance of security analysis tool. Some of these, such as the ‘Bodgeit’ [37] emulate a typical web application while others such as ‘WAVSEP’ present a long list of vulnerabilities that can be tested for precise benchmarks. This latter application is used for the large scale dynamic analysis comparison of [32]. This website found there is useful as a comparison with the results of this thesis.

The Uniface framework version used in this thesis makes use of a Java-based frontend. An experiment that also deliberately injected vulnerabilities in a Java web application was performed by Meganathan and Geoghenan in [9]. This case study only used one analyzer, the Source Code Analyzer by Fortify, a commercial static analyzer. It shows how several types of vulnerabilities can be injected in a Java application and how they must be solved. The method by which information leakage was performed in this thesis was also used here.

In a 2005 case study [10], dynamic analysis is described as useful but far from trustworthy. The authors describe the risk of using it: *“Organizations using such tools should not be lulled into a false sense of security, since such tools cover only a small part of the spectrum of potential vulnerabilities”*. Instead, it should only be used as an aide for testers, automating repetitive test cases.

An experiment that is more similar to the one proposed in this thesis was executed by Ferreira and Kleppe [12]. Eight different vulnerabilities were tested using a combination of commercial and open source dynamic analyzers. It mentions some intrinsic vulnerabilities common in most tools. crawlers tend to be unreliable in finding all URLs and human intervention is often needed. Moreover, each tool only uses a selected number of the possible requests making each tool behave differently. More troublesome is that most tools could only detect a small number of the total vulnerabilities.

Similar results were recorded by Lary Suto in [11]. Three dynamic analyzers were tested on a Java web application and two of them scored very high numbers of false positives and false negatives. Only the commercial product NTOSpider got reliable numbers. Moreover, Johnson et all’s *“Why Johnny Can’t Pentest: An Analysis of Black-Box Web Vulnerability Scanners”* [13], shows an experiment with 17 vulnerabilities and 11 analyzers, many of them commercial. The results also yielded high number of false positives and most tools had a low detection rate. Issues noted were that many tools have problems supporting technologies such as JavaScript or multimedia data. Also continuing problems with the crawlers were reported.

3.2 Research Questions

The end goal of the experiment is to know the effective differences between the two automated analysis techniques. This is done by measuring the number of found vulnerabilities. The difference can then be determined by comparing which vulnerability is found by what technique. Moreover, it is also important to known if there are any vulnerabilities that cannot be found by any technique. Therefore, the following questions have to be asked:

Goal 1A *Which vulnerabilities can be detected by static and not by dynamic analysis (or vice-versa)?*

Goal 1B *Which vulnerabilities cannot detected by either techniques?*

Another issue that needs to be addressed is how both types of techniques are represented. There are a number of implementations available for each, all varying from one to another using different attack patterns or vectors. This means that if one analysis package fails to find a certain vulnerability it does not mean that another package will not. This thesis uses multiple tools for each type from which the combination of results might be able approximate an ideal analysis tool. However, there can be still differences left that need to be adressed. This raises the question:

Goal 2 *Which particular analysis tools are able to find which vulnerabilities?*

3.3 Context: Uniface

The Uniface is a development and deployment platform aimed at enterprise environments, that combined forms the *Uniface Application Platform Suite* (APS). It has a focus on portability, support a wide range of database systems and allowing deployment on a variety of platforms. One deployment option is using a web server, employing the so-called *Uniface Web Application Server* (WASV) to run the applications. As this thesis is concerned with web applications, this is also the method that will used in here. However, other methods such as a Windows executable or a mobile app are also available.

Applications are created using the *Uniface Rapid Application Development Environment*, which combines programs such as an IDE and a WYSIWYG web editor but also tools for business modeling and debugging. This development environment itself is also created in Uniface.

Architecture of a Uniface web application

Any web application starts off with the user entering its URL in their browser. If the URL were `http://www.example.com/dir/index.html?param`, the HTTP protocol would be used to access the resource located at the *path* `dir/index.html` on the server with the *hostname* `www.example.com`. This would be done by sending an HTTP request to the web server which includes multiple *headers* such as cookies but also the *URL query*, in this case `param`.

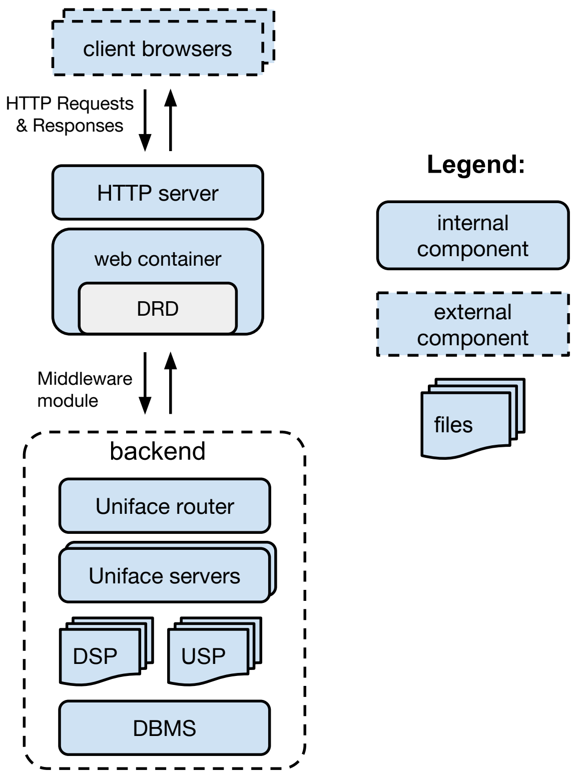


Figure 3.1: The component view of an architecture of a Uniface web application

What happens when a HTTP request is sent to a Uniface web application server is shown in figure 3.1 as a component view of the architecture. By default, Uniface 9. 6 uses Apache Tomcat 7, which serves as not only a web server handling HTTP requests and responses, but also a *web container*. Such a web container is a virtual machine that runs Java *servlets* to which requests are sent and from which responses are received. If a request is send to a web server running Uniface, any URL with the path `/uniface/DRD/` is send to the *dangerous request dispatcher* (DRD) servlet. This specially crafted request dispatcher is based on the regular *web request dispatcher* used in Uniface. The DRD, however, is the version compromised with the many vulnerabilities shown in this thesis.

The DRD analyses and extracts the contents of the request, such as the URL query, form input and cookies, and sends it to the Uniface router using a TCP socket. These contents are directly controlled by the user, so several validation and sanitization steps have to be done at this point. This makes this step and important focus point in security analysis.

The Uniface router check the user credentials and forwards the data to any of the available *Uniface servers* or creates one if none are available. The server runs the requested component which can be either a *Static (Uniface) Server Page* (USP) or a *Dynamic Server Page* (DSP) in web applications. These pages can run script from other Uniface components as well if necessary and DSPs can make us of AJAX requests. Components' scripts will often include in and output statements to a *database management system* (DBMS) connected to the Uniface server.

After the request has reached the Uniface server and has been processed, an HTML document with its necessary resources is returned in reverse direction back to the client.

Applied technologies

Each of the components mentioned in the previous paragraphs make use a range of different technologies. Tomcat, both the web server and the web container, as well as the DRD are written in Java. On the hand, backend components like the Uniface server and router, are written in C and C++. The USP and DPS pages that are executed make use of a custom built 4GL technology called the *Proc language*. This stage also requires calls to be made to the database which users SQL queries. The returned web pages are written in HTML including CSS for the makeup and JavaScript for several types of user interaction, including the *JQuery* library to process the AJAX-calls send back.

The multiple technologies can make it hard when it comes to security analysis. Fortunately, a few components stand out when it comes to security, such as the DRD, which will be the main focus point of the experiment.

Chapter 4

Experimental Setup

This chapter examines the setup and implementation of an experiment which tackled the problems mentioned in the previous chapter. The experiment consists of a modified Uniface web application that uses the previously mentioned DRD, a Java servlet injected with several dangerous vulnerabilities. The resulting web application that was created was called the *Vulnerable Uniface Website*.

The first part of this chapter examines which vulnerabilities were injected into the application and how they were selected. The second part is concerned with the application itself, while in the part mentions the different analysis tools that have been used. Finally, the metrics are discussed that were used to compare the different development tools.

4.1 Tested Vulnerabilities

As can be read in 2.1, a large number of vulnerabilities appear in modern web applications and can be found in databases like the CWE. But as each experiment has its limits, only a select number of vulnerabilities can be examined. Therefore, the ones that are most common and most risky have to be used, as well as the ones that are relevant in the context of Uniface web application.

Lists of the most relevant modern vulnerabilities are compiled by OWASP and SANS/MITRE and are shown in table A.1 and A.2 respectively. Some vulnerabilities can be discarded immediately as these are not relevant for the Uniface architecture, the DRD and the scope of this research. For example, in the OWASP top 10 “components with known vulnerabilities” is not the most useful vulnerability in this experiment as this refers to another vulnerability in a different component rather than any specific vulnerability. Similarly, "security misconfiguration" can often enable other vulnerabilities (such as the denial-of-service seen later on) but is harder to use by itself.

The SANS/CWE top-25 contains "Reliance on Untrusted Inputs in a Security Decision" (S10), discarded for the same reason, and no external code is executed as in S14/S16. This thesis focuses on the DRD which uses Java. Hence, "classic" buffer overflow (S3/S20) does not apply as the standard libraries generally contain well tested functions (no S18). Log forging is mostly a responsibility for the Tomcat server, hence not applicable to this experiment.

List of vulnerabilities

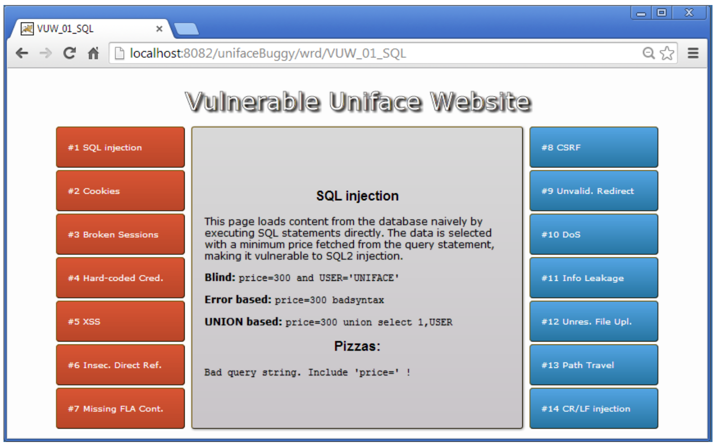
Table 4.1 shows the vulnerabilities that have been selected in the end. Other reasons that been considered for including the vulnerabilities is how well they could be implemented into the application. A detailed description of each vulnerabilitiy is listed in the appendix.

Flaw ID	**Description**	**Source**
# 1	SQL injection	A1, S1
# 2	Bad Cookie Settings	A5, S18
# 3	Broken Session Management	A2
# 4	Use of Hard coded credentials	A2, S7
# 5	Cross-Site Scripting	A3, S4
# 6	Insecure Direct Object Reference	A4
# 7	Missing Function Level Access Control	A7, S5
# 8	Cross Site Request Forging	A8, S12
# 9	Unvalidated redirect	A10, S22
# 10	Denial-of-Service	OWASP top10 2004-A9,
# 11	Information Leakage	OWASP top10 2007-A7, [9]
# 12	Unrestricted file upload	S9
# 13	Path Traversal	S13, [9]
# 14	Response splitting	[15]

Table 4.1: Vulnerabilities in experimental setup. Sources are either from the OWASP top 10 (A's), OWASP top 10 notes, SANS/CWE top-25 (S) or otherwise specified in the bibliography.

4.2 Implementation of the vulnerabilities into the web application

Each vulnerability listed in table 4.1 is implemented in the vulnerable web application. However, not all vulnerabilities could be implemented into the DRD, while other were not directly observable in the web frontend, such as 'hard coded credentials'. For each vulnerability a separate Uniface server page was created. In this way each had a separate URL that could be identified in the scanner results of dynamic analyzers. The resulting web application is shown in 4.1.



SQL injection

This page loads content from the database naively by executing SQL statements directly. The data is selected with a minimum price fetched from the query statement, making it vulnerable to SQL2 injection.

Blind: `price=300 and USER='UNIFACE'`

Error based: `price=300 badsyntax`

UNION based: `price=300 union select 1,USER`

Pizzas:

```
Bad query string. Include 'price=' !
```

Figure 4.1: The look of "Vulnerable (Uniface) Website", starting at the SQL injection page.

Location of the tested vulnerabilities

Not all vulnerabilities could be inserted into the DRD as many are caused by a faulty design implementation. Where each vulnerability is present is shown in table 4.2. Static analysis tools can only detect vulnerabilities in the DRD as this is the only part that was scanned for this experiment.

Testable Vulnerabilities		
Flaw ID	**Description**	**Location**
# 1	SQL injection	DRD
# 2	Bad Cookie Settings	DRD
# 3	Bad Session Management	design
# 4	Use of Hard coded credentials	DRD
# 5	Cross-Site Scripting	DRD
# 6	Insecure Direct Object Reference	design
# 7	Missing function level access control	design
# 8	Cross Site Request Forging	design
# 9	Unvalidated redirect	design
# 10	Denial-of-Service	DRD
# 11	Information Leakage	DRD
# 12	Unrestricted file upload	design
# 13	Path traversal	DRD
# 14	HTTP response splitting	DRD

Table 4.2: The location of each tested vulnerability: either at the design level or injected into DRD.

4.3 Methodology

To compare the different scanning techniques a consistent method of analyzing the test results of each tool is required. While the static and dynamic techniques serve the same purpose, they work in different ways. Static analysis show the flagged lines of code while dynamic analysis present a list of suspicious HTTP requests. For the former this means that we can aspect a much higher number of false positives for numerous reasons. For one, it is easier for a dynamic analysis to be certain of vulnerability as it can directly prove whether a SQL injection, path traversal or XSS is actually working. Second, the same vulnerability can be flagged at multiple lines in static analysis. Another difference is that static scanning has strictly defined search range: all lines of code. Dynamic scanning on the other hand must use a spider or proxy to find all pages and will often tries to guess hidden pages. As result the range can differ from one static scanner to another.

Metric

The most important data to record are how many vulnerabilities have been found. In principle, the number of targets is 14, as seen in table 4.1, but not each technique can theoretically find each vulnerability. For example, "hard-coded credentials" could not ever be found by dynamic scanner.

In addition to the theoretical difference between the two techniques, not all scanning tools are designed the same way. Each static scanner is made-up a set of rules and each dynamic scanner contains a number of different attack vectors. Not one tool has the same set of rules or vectors as another.

Measurements

Each tool was tested using the available automatic scan with any relevant option turned on. As most dynamic analysis tools had trouble with making sense of the JSON messages used for AJAX, the offending pages were navigated manually with the Firefox web browser. Any traffic could then by intercepted by a proxy running on the analysis tool. Such a technique was available for most of the scanners.

Not all dynamic analysis tools could properly define the scanning range: their web crawler discovered more locations beyond the 14 vulnerability pages. This was amplified by the fact that Apache Tomcat also shows a page for any non-existing URL. Any (false positive) vulnerability discovered on such a page was discarded. The total number of discovered (both correct and false positive) vulnerabilities was determined by counting the number of distinct HTTP requests (including the URL query).

The static analyzers only scanned the Java source code of the DRD or its compiled byte code. All line numbers flagged by the scanner were counted. This means that any correct detection could contain multiple detected lines, but can also results in high number of false positives. Some vulnerabilities consisted of multiple lines of code, such as multiple lines of invalidated data for cross-site scripting, but only one detected line was necessary for a vulnerability to be discovered.

For each detector, only the count of vulnerabilities were recorded that were theoretically predicted. For example, unvalidated redirect was not implemented into the DRD source code, thus any such vulnerabilities detected by a static scanner were discarded.

4.4 Selected tools

The scan injected vulnerabilities in the setup, 10 different analysis tools were used – five dynamic and five static detectors. By combining multiple results the limitations of a particular implementation are reduced and any idiosyncratic behaviors or bugs minimized. This should yield a better picture of what any of the two techniques are capable of.

There were various reasons for choosing a particular analysis tool. In the case of static analyzers, the options were limited to those that understood the Java programming language. For the dynamic analysis, the tools were selected by to their performance according to earlier studies, such as [12] and [32].

4.4.1 Dynamic Analysis

IronWASP

The *Iron Web application Advanced Security testing Platform* is open source scanning tool with a focus on customizability and ease of use. As part of the standard automated tests, it offers detection of several vulnerabilities that were relevant to the experiment: cross-site scripting, SQL injection, local file include (similar to the path traversal of the experiment), bad cookie settings and cross-site reference forging.

N-Stalker

N-Stalker is a commercial security analysis system that also exists as a free versions, offering a limited set of features. It also supports several specialized options not seen in other tools, such as Flash and Silverlight support. Manual browsing through a proxy is not allowed in the free version. This will make it hard for the scanner to navigate some parts of the web application.

Off features in the free versions, three are relevant for the experiment: information leakage, cross-site scripting and cookie settings.

The free version used for the experiment has the build number 10.13.12.8.

ZAP

The *Zed Attack Proxy* (ZAP) is an analysis tool developed as part of OWASP, providing a wide range of penetration testing techniques. Besides active (automated) scanning and a manual proxy, it provides a range of specialized features.

By setting up the active scan, one learns that the vulnerabilities it offers cross-site scripting, SQL injection, response splitting, unvalidated redirect and path traversal. Also numerous minor security warnings are given for bad security policies. One such is a session id showing in the URL query, known to the experiment as 'bad sessions management'.

The version used for this experiment is ZAP 2.3.1.

Wapiti

Wapiti is a plain command-line tool that uses several modules to detect a small range of vulnerabilities. It includes vectors for testing cross-site scripting, response splitting, SQL injection information leakage.

It could have trouble navigating some pages as it has no feature for manual proxy browsing. The experiment used Wapiti version 2.3.0.

w3af

The *Web Application Attack and Audit Framework* (w3af) is large package that contains a large number of independent plugins. Some of these have overlapping functionality and some are very tiny, such as searching the host on Google.

Of the many plugins that were offered, the ones used enabled the detection of SQL injection, bad cookie settings, cross-site scripting, information leakage, unvalidated redirection, unrestricted file upload and response splitting.

This thesis used w3af version 1.1.

4.4.2 Static Analysis

Find Security Bugs

Find Security Bugs is a security related plugin for the all-purpose bug finder FindBugs. Created at the University of Maryland in Java, FindBugs includes a code browser in a GUI but is also available as an Eclipse plugin, which is the required setup for Find Security Bugs. The version of the Find Security Bugs plugin used here was 1.2.0 with FindBugs version 3.0.

The detector uses a total of 45 bug patterns [33]. Most of these signify only mild security risks, but only 5 of these were directly relevant for the experiment. These were 'potential path traversal', 'Tainted filename read' (related to path traversal), 'potential SQL injection' , 'potential XSS in servlet' and 'hard code password'. Moreover, the pattern 'request query string' was as it used for the SQL injection, as well as the path traversal vulnerability.

There were some other patterns similar to one of the detectable vulnerabilities such as 'unvalidated redirect', but this vulnerability is not implemented in the request dispatcher. Moreover, the pattern 'cookie usage' is only related to contents of a cookie, not to its settings. As a result 4 types of vulnerabilities were expected to be found by this detector.

LAPSE+

The OWASP *LAPSE+* tool is a plugin for Eclipse specifically designed for the Java EE framework, which is used by the request dispatcher. It uses a special technique for detecting *taint propagation*, enabling it to detect sanitization errors. A detailed description of this technique can be found in [16][p.101], but can be summarized as detecting any input *sources* and the corresponding output *sinks*. Proper sanitization actions should have been performed along the way from source to sink. The sinks are not given in terms of vulnerability, but instead are given by their output type such as 'header manipulation' or 'cookie poisoning'. This made the sinks hard to compare to with other vulnerabilities, thus these were not taken into account during the experiment. As this tool focuses on injection attacks, it was expected to find SQL injection, cross-site scripting, path traversal and response splitting. These vulnerabilities are also listed as such in its documentation [34].

For this experiment, version 2.8.1. was used as a plugin for Eclipse Helios release 2.

Visual Code Grepper

VCG is a fast analyzer that supports C++, C#, Visual Basic, PHP and Java. It takes it name from the Unix Grep utility, as it uses regular expressions to search for bad or dangerous lines of code, as in the Unix Grep tool. Depending on the source code this could lead to high numbers of false positives if a certain keyword is found often in the particular coding style. For this experiment, VCG version 1.6.1 was used.

According to the patterns found in the configuration files, directly relevant vulnerabilities include hard-coded credentials. It also includes less precise patterns such as 'input validation', suggesting SQL injection, XSS and response splitting, but also 'Operation on Primitive Data Type' which could suggest a denial-of-service. These vulnerabilities difficult to identify this way as it is difficult to identify such problems using only keywords.

Veracode

This detector was the only commercial tool used in this experiment. Although it is software-as-a-service, it avoids the need to upload source code by scanning Java byte code instead.

A full coverage of both the OWASP top-10 [35] and the CWE/SANS top-25 [36] is claimed. However, the exact methods by which each vulnerability is detected could not be determined as implementation details of this commercial product are not fully disclosed. Moreover, the claims are based on using both dynamic as well as static analysis and this experiment only uses Veracode's static analysis due to the used license.

According the previously mentioned sources, for this experiment Veracode's static analysis should detect SQL injection, cross-site scripting, path traversal, use of hard coded credentials, information leakage and response splitting. Veracode also supports unvalidated redirect, cross-site request forgery, missing access control and unrestricted file upload. But these are not present in the source code of the scanned DRD.

For keeping track of the version, the static scan was executed on the 23rd of June 2014.

Yasca

Yasca is a simple static tool that works similarly to VCG as it 'greps' important keywords. However, the latest version also supports a simple source / sink algorithm. It allows various plugins to add extra functionality, such as a Find Security Bugs plugin, but for this experiment only the bare version was used. In this case the version used was Yasca 2.21.

According to its manual [20], it is able to detect hard-coded credentials, denial-of-service, SQL injection and cross-site scripting.

Summary expected results

According to the attack patterns each tool offers the vulnerabilities shown in table 4.3.

Scanner	**# of detectable vulnerabilities**	**Detectable vulnerabilities**
Dynamic theoretical	13	1,2,3,5,6,7,8,9,10,11,12,13,14
IronWASP	5	1,2,5, 13,14
N-Stalker (free)	3	2,5,11
Wapiti	4	1, 5,11, 14
w3af		1,2, 5, 9,11,12,13,14
ZAP	7	1,2,3,5,9, 14,15
Static theoretical	8	1,2,4,5,10,11,13,14
FindBugs	4	1,4,5, 13
LAPSE+ [a]	4	1, 5, 13,14
VCG	5	1,4,5,10,13
Veracode [b]	8	1,2,4,5,10,11,13,14
Yasca	4	1,4,5,10

Table 4.3: The vulnerabilities that can be expected to be found by each tool according to their feauture list. Eac
for each category is based on the location of the vulnerability, DRD or desing level.

[a]LAPSE only detects injection type attacks.
[b]Binary scan.

Chapter 5

Experimental Results

This chapter shows the results of the experiment with vulnerable web application. The outcome of each individual analysis tool is examined, such as the number of discovered vulnerabilities. Additionally, the number of false positives, non-existing vulnerabilities, and the false negatives, undiscovered vulnerabilities that were expected to be found, are counted. When possible, the origin of these mistakes are described.

5.1 Results of the dynamic analysis tools

IronWASP

IronWASP managed to detect the insecure cookie, cross-site scripting and the response splitting vulnerability. However, it could not find the SQL injection. Many blind SQL injections were inserted that resulted in the error page, but no error code was returned so IronWASP did not recognize them.

For the path traversal, the tool managed to insert double dots in the correct place. However, it could not recognize the resulting response as most insertions have no effect on the returning web page. Another cross-site scripting was discovered on the the cross-site reference forging page as the insertion was returned back in a AJAX response. However, the returned JSON object did not affect the site and could therefore not be considered a successful XSS, but a false positive instead.

N-stalker

Of the three expected vulnerabilities only cross-site scripting was detected. The bad cookie settings could not be found as the scanner could not properly handle the AJAX interaction that retrieved the cookie. Other scanners could fix this by using a proxy server, but this feature is disabled in the free version. Information leakage was also not discovered, the stacktrace could not be recognized.

Wapiti

The only vulnerabilities that were detected were 4 instances of cross-site scripting. The normal location was correct, but it flagged false positives on the pages of path traversal, cross-site request forgery and unvalidated redirect. These vulnerabilities were detected by modifying the URL query and comparing the returning response with the unmodified URL query. In this case, Wapiti detected false positives by being too sensitive in comparing the pages.

w3af

The cross-site scripting vulnerability was found in the correct place as well as the response splitting. The SQL injection was discovered by using a URL query [1] not used by any other tool.

The unvalidated redirect was not found even though it did provide URL that contains the vulnerability. However, w3af could not execute the JavaScript that makes the redirect work. The insecure cookie was received correctly but w3af did not notice its flawed configuration. A file upload was done, but the detector searched for the presence of the file anywhere on the webpage, which is not the way it was implemented.

No false positives have been detected.

ZAP

Of all expected vulnerabilities, cross-site scripting, response splitting, bad cookie setting and bad session management were found correctly and without any false positive. However, SQL injection was found in the right location but showed up at the file upload form instead. The Uniface widget for file upload returns part of the file back to the page. Thus after using the file upload with a SQL string, ZAP noticed a change in the web page and signalled the SQL injection.

The SQL injection was not found on the actual SQL page even though it managed to find the right input method (the URL query). Any used input did not result in a different page and was therefor not recognized. Other

[1] URL query `?price=` was used without any price value. The result was a clear database message.

error-based SQL input did result in the Uniface error page but was not recognized as such by the tool. Similarly, the path traversal page was properly injected with dot-dot-slash combinations, but not right answer could be found that yielded a recognizable response.

5.2 Results of the static analysis tools

Find Security Bugs

This FindBugs plugins raises many flags on the source code, but most of these were related to categories not considered for the experiment. Many instances of path traversal were tagged but all of these were linked to output to log files. It did, however, find 9 cases of unsafe ‘query string’ editing, 2 of which were due to the ‘SQL injection’ and ‘path traversal’ vulnerabilities of the experiment. In total, 15 false positive lines of code were found.

There were several false negatives. The XSS was not found as it did not recognize the the commented out string operations. The hard coded password was not found as Find Security Bugs uses a pattern not compatible with the injected vulnerability.

LAPSE+

The OWASP LAPSE tool finds a total of 24 lines of code, categorized as either ’cross-site scripting’, ‘path traversal’ or ‘HTTP response splitting’. However, the first two types were false positives caused by log file output. Both instances of response splitting were justified.

The other vulnerabilities were not found as LAPSE could not find their proper sinks. The objects that would act as sinks were extracted by looping over arrays containing Java methods. As result, LAPSE could not identify these individual methods properly.

Veracode

Veracode correctly detects the hard coded credentials, insecure cookie and the response splitting vulnerability. A false positive cross-site scripting was detected on a `print()` statement used for copying input data from the Uniface server to the response. Many other false positive signaled information exposure but were caused by writing output to log files.

Why more than half of the vulnerabilities were not discoverd could not be determined as Veracode offers no insight in its inner workings.

Visual Code Grepper

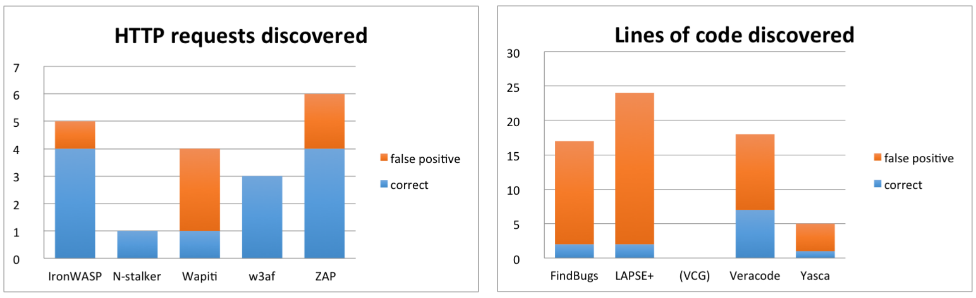
VCG managed to find all of its expected vulnerabilities. The hard-coded credentials were found two times due to the presence of the keyword ‘password’, one of which a false positive. 331 lines were found that contained ‘operations on primitive types, but with only 11 being part of the denial-of-service vulnerability. Similarly, 21 lines were attributed to bad input validation, but only a few of them were related to the injection attacks. Besides the false positives, a bigger issue for these flags is that they do not know exactly what kind of vulnerability it was.

Yasca

This scanner detected two cases of stored credentials, of which one was a false positive. Moreover, one false positive case of cross-site scripting case was found due to a `println()` statement to a log file. Also falsely discovered were the two lines of denial-of-service vulnerabilities, which were already commented out.

5.3 Combined Results

The experimental results for each detector against each vulnerability is shown in table 5.1. For each static analyzer the number flagged lines of code have been counter and for each dynamic analyzer the number of flagged HTTP requests. Note that one vulnerability can correspond to multiple flagged requests or lines of code, and vice versa. A graphical view of the number of false positive and correct detected vulnerabilities is shown in figure 5.1.



(a) (b)

Figure 5.1: Bar graphs of the number of discovered HTTP requests and lines of code per analysis tool.

		Dynamic scanners					Static scanners				
Flaw ID	**Description**	*IronWASP*	*N-stalker*	*Wapiti*	*w3af*	*ZAP*	*FindBugs*	*LAPSE+*	*VCG*	*Veracode*	*Yasca*
	Total flags	5	1	4	3	6	17	24	354	18	5
	Flags correct	4	1	1	3	4	2	2	19	7	1
	Flags false	1	0	3	0	2	15	22	335	11	4
# 1	SQL injection	−		−	+	−	+	−	+	−	−
# 2	Bad cookie settings	+	−		−	+				+	
# 3	Bad session management					+					
# 4	Use of Hard coded credentials						−		+	+	+
# 5	Cross-site scripting	+	+	+	+	+	−	−	+	−	−
# 6	Insecure direct object reference										
# 7	Missing function level access control										
# 8	Cross-site request forging										
# 9	Unvalidated redirect				−	−					
# 10	Denial-of-service								+	−	−
# 11	Information leakage	−	−	−	−					−	
# 12	Unrestricted file upload				−						
# 13	Path traversal				−	−	+	−	+	−	
# 14	HTTP response splitting	+		−	+	+		+		+	
	Vulnerabilities expected	5	3	4	8	7	4	4	5	8	4
	Vulnerabilities found	3	1	1	3	4	2	1	5	4	1

Table 5.1: Vulnerabilities found per detector where ‘+’ signifies correct detection while ‘−’ means predicted but not found vulnerabilities. Flags are the counted hits by the detector, lines of code for a static tool and unique HTTP requests for a dynamic tool. The false positives are those flags that did not contain a vulnerability. Down the bottom are the total numbers of vulnerabilities found and expected.

Chapter 6

Analysis and Discussion

In this chapter the results of the previous chapter are discussed. First the total number of vulnerabilities for the tools are examined. Then the precisions of the discoveries is discussed by looking at the false positives. Finally, the data is combined into a new set to compare the two principal detection techniques with each other.

6.1 The amount of vulnerabilities discovered

In table 5.1 it is apparent that few tools manage to scan a significant number of vulnerabilities. As shown in the bottom of the chart, most tools could barely discover half of the expected numbers.

The number of expected vulnerabilities is different for each tool. Thus, in the case of dynamic analysis no one tool is objectively better than another. However, in the case of static analysis, VCG clearly jumps out with 5 vulnerabilities discovered out of 5 expected. But as a side note this tool does contain a very high number of false positives. Moreover, most detected lines were not explicitly notated as a certain vulnerability but were given broader categories such as 'input validation'.

Causes for false negatives

There were several reasons why some of the expected vulnerabilities were not found. In the case of dynamic analysis, several tools had difficulty with properly navigating all functionality. This problem is also mentioned in several other studies, as seen later on.

Some vulnerabilities were inherently difficult to recognize. Path traversal attempts are easily executed, but the effect was hard to measure. The right combination had to be known to make any effect on the web application.

A similar problem was found with the SQL injection attempts. In the case of blind and UNION based injection it becomes hard to get right combination to observe any change in the HTTP response. Such attacks are easier when performed by a human as it requires a bit of puzzling and creativity. The error based SQL attacks had the problem that the resulting error page did not get recognized as such. No error status code was returned, which is a encouraged policy from a security perspective. Other vulnerabilities, such as unrestricted file upload had similar problem, as it was not always clear if the file upload was successful.

In the case of static analysis, determining the reason for false negatives is harder as most tools do not leave insightful log files. For some pattern searches it is clear that some tools use other patterns that do not always recognize the same vulnerabilities, such as for hard coded credentials. Another reason is the difficulty of properly doing data flow analysis and determining where any sanitization has taken place.

6.1.1 Answers to research questions

Now that the discovery rates of each tool is known the following research questions can be answered:

Goal 2 *Which analysis tools are able to find which vulnerabilities?*

Most analysis tools manage to find similar types of vulnerabilities. In many cases, the difference between one tool detecting a vulnerability and another one not rests on implementation details. Some will work better with certain websites than with others. However, only ZAP provided a way for detecting the based session management. For static analysis, only VCG could discover the denial-of-service vulnerability.

6.2 Discovery accuracy

Another import detail is the number of false positives that each tool found. Roughly, this acts as measure for the accuracy of the tools. More false positives means that it is harder to use the results of the tool.

In figure 5.1, the total number of correct discovered vulnerabilities is shown in combination with the false positives. Although statistically insignificant, a clear trent is showing that static analysis typically leads to more false positives.

Causes for false positives

In the case of dynamic analysis again several of the reasons can be attributed to the behavior of the Uniface framework. Data returned from AJAX requests were hard to handle. For instance, IronWASP discovered a false cross-site scripting vulnerabilities in a otherwise unused JSON object. On the other hand, other studies (as seen in section 3.1) showed similar problems with other types of web applications.

Static analysis has a much higher false positive rate than dynamic. However, most of them were due to output to a log file which was often mistaken for a cross-site scripting issue. A more curious reason is Yasca that observed a vulnerability in a commented out line.

6.3 Comparison with other studies

The results are reminiscent to those of [12]. There, all dynamic analyzers could also only find a small part of the vulnerabilities. Moreover, in earlier results from [11], two out of three scanners also got very low hit rates. Issues with lack of technology support and vulnerable requests that were not recognized are also reported in [13].

6.4 Comparison of the two techniques

Although each analysis tool could only detect part of the expected vulnerabilities, combining the results for each category leads a different image. Table 6.1 shows which vulnerabilities were detected and expected when each tool is combined. This views helps in answering the questions that were asked in section 3.2.

6.4.1 Answers to research questions

The first set of questions read the following:

Goal 1A *Which vulnerabilities can be detected by static and not by dynamic analysis (or vice-versa)?*

Goal 1B *Which vulnerabilities cannot detected by either techniques?*

Using the results from table 6.1, these questions can now be answered. Vulnerabilities found only by dynamic analysis is *bad session management* and *Information Leakage*. The latter was expected for the static tools but never found. A large range of other vulnerabilities were expected to be exclusive to dynamic analysis but could not be discovered by any tool. These are *insecure direct object references*, *missing function level access control*, *CSRF*, *unvalidated redirect* and *unrestricted file upload*.

Other vulnerabilities were only detectable by static analysis but not dynamic analysis. As expected this includes hard-coded credentials. Others were expected for dynamic analysis but were only detected by static analysis. These include *denial-of-service* and *path traversal*.

Vulnerabilities that were not discovered by either technique are earlier mentioned *insecure direct object references*, *missing function level access control*, *CSRF*, *unvalidated redirect* and *unrestricted file upload*. However, theoretically these should be detectable, and features for finding some of these vulnerabilities are included in some tools.

		Dynamic scanners		**Static scanners**	
Flaw ID	**Description**	Expected	Found	Expected	Found
# 1	SQL injection	x	x	x	x
# 2	Bad Cookie Settings	x	x	x	x
# 3	Bad Session Management	x	x		
# 4	Use of Hard coded credentials			x	x
# 5	Cross-Site Scripting	x	x	x	x
# 6	Insecure Direct Object Reference	-			
# 7	Missing function level access control	-			
# 8	Cross Site Request Forging	-			
# 9	Unvalidated redirect	x			
# 10	Denial-of-Service	-		x	x
# 11	Information Leakage	x		x	
# 12	Unrestricted file upload	x			
# 13	Path Traversal	x		x	x
# 14	HTTP Response splitting	x	x	x	x

Table 6.1: The results of each analysis technique combined. Expected shows which vulnerabilities are expected according to there location, the DRD or at design level (see table 4.2). Expected shown with a '-' means that the vulnerability was not supported by any of the used analysis tools.

Chapter 7

Conclusion

From the results there was no large divide visible between the vulnerabilites found by dynamic and static analysis. Most of the vulnerabilities that were only expected for dynamic analysis could not be discovered anyway. Notable exception for static analysis was ‘hard-code credentials’ and ‘bad session management’ for dynamic analysis. The denial-of-service vulnerability was also unique to static analysis, but could in principle be supported by other dynamic tools.

Some severe limitations were detected for the dynamic tools. As mentioned in early studies, many tools had issues with properly navigating the web application, especially when AJAX was involved. Another important issue is that multiple injection attack vectors relied on measuring a change in the webpage. However, such changes were not always supported well by the application’s infastructure. As a result some tools were too sensitive while others failed in detecting error pages when no error code was returned.

Static analysis showed high numbers of false positive, agreeing with theoretical predictions. In the case of VCG this also means that a higher detection rate equals a higher false positive rate. Many of the false negatives were injection vulnerabilities. Some of these are hard to detect, while others – such as response splitting – were relatively easy. In short, d ifficulty of such vulnerabilities varied widely.

7.1 Further research

More research could be performed on top of what was investigated for this thesis. For instance, similar research could be done on other parts of the Uniface framework. More general investigations could be made on the Apache server, which also contains some important security steps However, the source code is quite larger than the DRD (over a 100 thousand lines of code). Not too much for most static analysis tools but the researcher himself needs to be deeply knowledgable about the code. Another part could be JQuery, although JavaScript scanners are harder to come by and the code harder to understand.

Besides looking into other parts of the architecture, other types of security issues are available for analysis. HTTPS was not used in this thesis as it increases the difficulty of the rest of the vulnerabilities. Moreover, the 'Heartbleed' bug was discovered during the design of the experiment. Also Uniface support a wide range of authentication and session management options. This thesis only used a simple cookie-version but more advanced options are left to be explored. The downside is that not all tools support user login equally well.

Chapter 8

Acknowledgements

By finishing this thesis my internship at Uniface BV has come to an end. It was educative and productive period where I teached myself a great deal about application security analysis and became familiarized with the Uniface framework, which despite its age can still be relevant for modern companies. I also enjoyed the multicultural work environment and experienced developers. Therefore I wish to thank a number of people for supporting me during my internship.

First of all I wish to thank my daily supervisor and Beijing native Ming Zhu. Besides his valuable supervision, his passion and insightful opinions on world politics helped me through the week. Next, I thank our team leader Jorge Núñez for his eagerness to help whenever necessary, and my desk neighbor Nick Hall for assisting me with some technical issues.

To Maarten van Leer I wish to express my gratitude for his willingness to create an internship position. And of course to Ton Blankers for letting me get in touch with Uniface.

Finally, I wish to thank my University supervisor Magiel Brutink and especially his patience with me searching a research topic.

Chapter 9

Bibliography

9.1 Referenced Articles

[1] G. McGraw, “Software security,” *Security Privacy, IEEE*, vol. 2, pp. 80–83, Mar 2004.

[2] G. McGraw and B. Potter, “Software security testing,” *IEEE Security and Privacy*, vol. 2, pp. 81–85, Sept. 2004.

[3] T. Aslam, I. Krsul, and E. Spafford, “Use of a taxonomy of security faults,” in *Proc. 19th NIST-NCSC Nat. Information Systems Security Conf*, pp. 551–560, 1996.

[4] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, “A taxonomy of computer program security flaws,” *ACM Comput. Surv.*, vol. 26, pp. 211–254, Sept. 1994.

[5] U. Lindqvist and E. Jonsson, “How to systematically classify computer security intrusions,” in *Proc. 1997 IEEE Symp. Seurity and Privacy*, pp. 154–163, 1997.

[6] K. Tsipenyuk, B. Chess, and G. McGraw, “Seven pernicious kingdoms: A taxonomy of software security errors,” *IEEE Security and Privacy*, vol. 3, pp. 81–84, Nov. 2005.

[7] R. Johnston, “Being Vulnerable to the Threat of Confusing Threats with Vulnerabilities: Viewpoint Paper,” *Journal of Physical Security*, vol. 4, pp. 30–34, 2010.

[8] OWASP Foundation, *OWASP Testing Guide v3.0*, 2008.

[9] D. C. Wyld, J. Zizka, and D. Nagamalai, eds., *Testing for Software Security: A Case Study on Static Code Analysis of a File Reader Java Program*, vol. 166 of *Advances in Intelligent and Soft Computing*, Springer Berlin Heidelberg, 2012.

[10] C. C. Michael and W. Radosevich, "Black box security testing tools," 2005.

[11] L. Suto, "Analyzing the effectiveness and coverage of web application security scanners," 2007.

[12] A. Ferreira and H. Kleppe, "Effectiveness of automated application penetration testing tools," 2010.

[13] C. Kreibich and M. Jahnke, eds., *Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners*, vol. 6201 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2010.

[14] N. Teodoro and C. Serrao, "Web application security: Improving critical web-based applications quality through in-depth security analysis," in *Information Society (i-Society), 2011 International Conference on*, pp. 457–462, June 2011.

[15] M. Zalewski, *The Tangled Web: A Guide to Securing Modern Web Applications.* San Francisco, CA, USA: No Starch Press, 1st ed., 2011.

[16] B. Chess and J. West, *Secure Programming with Static Analysis.* Addison-Wesley Professional, first ed., 2007.

[17] M. Howard and D. E. Leblanc, *Writing Secure Code.* Redmond, WA, USA: Microsoft Press, 2nd ed., 2002.

[18] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code.* Pearson Higher Education, 2004.

[19] R. Bace and P. Mell, *Intrusion Detection.* Sams Publishing, 2nd ed., 2000.

[20] *Yasca User's Guide.*

9.2 Referenced Websites

[21] MITRE, "Common weakness enumeration." `http://cwe.mitre.org/`.

[22] MITRE, "Common vulnerabilities and exposures." `http://cve.mitre.org/`.

[23] J. Manico, "Owasp top ten proactive controls." `http://vimeo.com/79810133`.

[24] J. McCray, "Defcon 2017: Advanced sql injection," 2001. `https://www.youtube.com/watch?v=rdyQoUNeXSg`.

[25] OWASP, "Top 10: The Ten Most Critical Web Application Security Risks," 2013. `https://www.owasp.org/index.php/Top10`.

[26] SANS/MITRE, "CWE/SANS Top 25 Most Dangerous Software Errors," 2011. `http://cwe.mitre.org/top25/#CWE-807`.

[27] Neil MacDonald and Joseph Feiman, "Magic Quadrant for Application Security Testing," 2013. `http://www.gartner.com/technology/reprints.do?id=1-1GUOVQS&ct=130703`.

[28] Businesswire, "ebay inc. to ask ebay users to change passwords," May 2014. `http://www.businesswire.com/news/home/20140521005813/en/eBay-To%C2%A0Ask-eBay-Users-Change-Passwords`.

[29] "Industry Citations of OWASP." `https://www.owasp.org/index.php/Industry:Citations`.

[30] "Definition of a Security Vulnerability." `http://technet.microsoft.com/en-us/library/cc751383.aspx`.

[31] "The STRIDE Threat Model." `http://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx`.

[32] "Price and Feature Comparison of Web Application Scanners." `http://www.sectoolmarket.com/`.

[33] "Find Security Bugs: Bugs." `http://h3xstream.github.io/find-sec-bugs/bugs.htm`.

[34] "OWASP LAPSE Project." `https://www.owasp.org/index.php/OWASP_LAPSE_Project`.

[35] "OWASP TOP 10 | Veracode." `http://www.veracode.com/directory/owasp-top-10`.

[36] "CWE/SANS TOP 25 | Veracode." `http://www.veracode.com/directory/cwe-sans-top-25`.

[37] "The Bodgeit Store." `https://code.google.com/p/bodgeit/`.

Appendix A

Tables

OWASP top-10 2013	
A1	Injection
A2	Broken Authentication and Session Management
A3	Cross-Site Scripting (XSS)
A4	Insecure Direct Object References
A5	Security Misconfiguration
A6	Sensitive Data Exposure
A7	Missing Function Level Access Control
A8	Cross-Site Request Forgery (CSRF)
A9	Using Components with Known Vulnerabilities
A10	Unvalidated Redirects and Forwards

Table A.1: The 10 most critical security risks in web applications according to the Open Web Application Security Project (OWASP) [25]

SANS/MITRE top-25 2011	
S1	SQL injection
S2	Command injection
S3	Classic Buffer overflow
S4	Cross Site Scripting (XSS)
S5	Missing Authentication for critical functions
S6	Missing Authentication
S7	Use of hard-coded Credentials
S8	Missing Encryption of Sensitive Data
S9	Unrestricted Upload of File
S10	Reliance on Untrusted Inputs in a Security Decision
S11	Execution with Unnecessary Privileges
S12	Cross-Site Request Forgery (CSRF)
S13	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
S14	Download of Code Without Integrity Check
S15	Incorrect Authorization
S16	Inclusion of Functionality from Untrusted Control Sphere
S17	Incorrect Permission Assignment for Critical Resource
S18	Use of Potentially Dangerous Function
S19	Use of a Broken or Risky Cryptographic Algorithm
S20	Incorrect Calculation of Buffer Size
S21	Improper Restriction of Excessive Authentication Attempts
S22	URL Redirection to Untrusted Site ('Open Redirect')
S23	Uncontrolled Format String
S24	Integer Overflow or Wraparound
S25	Use of a One-Way Hash without a Salt

Table A.2: The 25 most dangerous software errors as of 2011 according to the SANS and MITRE institutes [26]

Appendix B

Tested Vulnerabilities

This part of the appendix lists all vulnerabilities that have been used by the experiment. For many of the vulnerabilities the detectability (search difficulty) is determined by consulting its properties in the OWASP top-10 [25] if it is available.

SQL injection

SQL injection consistently tops many vulnerability charts, and not without reason. Improperly sanitized input wrecks havoc among databases, allowing hackers access to valuable user or customer data.

Most SQL injections can be categorized into three types (as discussed in [24]): The easiest to accomplish is error based, which tries to analyze the information sent by the database upon bad input. More dangerous is *UNION based* which uses the SQL `UNION` command to inject one's own commands into the query – such as the infamous `DROP TABLE`. Lastly there is *blind sql*: trying to differentiate the output when asking true or false questions to the database.

SQL injection can be prevented by handling any incoming data with care. In principal, any user input is unreliable and should first pass the proper sanitation checks before being used. Especially suspect are characters such as the dash '-', which can be used to comment out parts of a command, the semicolon, which closes a statement, and the single quote '´' which allows malicious users to jump out of an input value and right into the SQL-query itself.

#1 Implementation Due to the complex way in which data is accessed through Uniface it is hard too achieve a true SQL injection: data is abstracted through a model of entities and occurrences that are not the same as the corresponding database objects. The data is fetched through a series of database transactions that make it hard to predict what the queries look like. This makes the traditional SQL injection very hard when using Uniface entities.

Fortunately for the experiment, Uniface allows custom SQL queries through the “sql” Proc statement. It is discouraged for regular data access due to its inherent risks, but is still available to Uniface developers. The downside of implementing it this way is that it is not very representative for a regular Uniface application.

detectability Any static scanner has to find the right sanitation, which can be hard as this involves the control flow and can pass different software systems. Fortunately, all of the SQL sanitation is done only in the Dangerous Request Dispatcher (DRD). Most dynamic scanners include a SQL attack of some type, but the difficulty lies in recognizing the result.

#2 Bad Cookie Settings

Cookies are a fundamental building block of modern world wide web. Although sometimes criticized for privacy concerns involving third party cookies, it is hailed by security aspects as a solid way of tracking user data on a website without resorting to databases. Users can access their own cookies themselve. However, this poses a risk it not only requires security on the part of the server but also on machine of the client. Some of the problems can be mitigated by making use of *httponly* cookies, which disallow editing through (potentially malicious) JavaScript code.

Implementation Uniface allows editing and reading of cookies through Proc code using the COOKIEIN and COOKIEOUT channels. These options are stored in associative array and are transmitted to the DRD. At the client the string is read and the cookie is set using the Javax API. To make the website vulnerable the DRD has been modified by deleting the lines that configure the `secure` and `httponly` settings.

detectability This vulnerability is very easy to detect by dynamic scanners as they can detect the absence of the *httponly* tag on a cookie. It is harder for static scanners as they have to track the method calls on the cookie object.

#3 Broken (Authentication) & Session Management

Given as number 2 in the OWASP top-10, broken authentication or session management can be major flaw for any serious web application. One example is a session-ID stored in the URL query. This makes it very susceptible to session-fixation as it is easy to change and it shows up types of logs that show the requested URLs on a server. However, many other possibilities also fall under this category.

Implementation The web application uses a login screen. The user provides his credentials which are checked by the server. If correct, the user is forwarded to user-page summoned by the session-id.

Detectability A dynamic scanner could potentially find this when it notices a session like token in the URL query. As this is handled by Proc code in the Uniface server page, it is not visible in the DRD and therefore not visible to any static detector.

#4 Use of Hard coded credentials

. Password should not be stored hard-coded in the source code, such as in the request dispatcher. Code can be shared by developers and is therefore vulnerability to leaking.

Implementation The DRD connects to the Uniface router and server using the credentials of the user that is running the router and server. This string is normally stored in separate configuration files, but in the case of the DRD no file is load but the string is directly defined in the Java code.

Detectability It is impossible for a dynamic scanner to look into the source code and is therefore unable to detect this issue. However, a static scanner should be able to find it as it contains typical keywords such as password.

#5 Cross-Site Scripting (XSS)

Cross-site scripting is one of most common and infamous of modern hacker attacks. Detection of this vulnerability is one of the most common features of both static as dynamic scanners.

Implementation A page contains a URL query of which the contents are returned into the HTML, which is known as *reflected* XSS. Normally, such input should be sanitized. However, such action is commented out in the source code of the DRD.

Detectability Should be observable by some of the static scanners focusing on data validation but requires sophisticated scanning techniques. For most dynamic scanners this is one the most basic features.

#6 Insecure Direct Object Reference

Each web server uses different URL syntax strategies to link to a certain aspect of a web application. Sometimes this includes parameters in the URL query. A vulnerability could occur if an attacker can access hidden parts of the website by editing the URL.

Implementation On the web application there is a control panel available for logged in users, displaying their user information. An attacker can access this page of other users by modifying the URL of this page to right user id.

Detectability The vulnerability is not present in the DRD so it discoverable by static scanners. For a dynamic scanner this would require sophisticated rules to detect and knowledge of which pages should be accessible, making it hard.

#7 Missing Function Level Access Control

Sometimes certain features of website lack the proper access controls, making them accessable to everyone. It could caused by a bug, or simply by sloppiness of the developer.

Implementation A user control page, including a logout button, is still accessable after the user is logged out. This vulnerability was caused by a bad design in a Uniface server page.

Detectability The vulnerability is not present in the request dispatcher making it invisible to static scanners. Dynamic scanners have usually no way of knowing which page should require login so they too fail in detecting this vulnerability.

#8 Cross Site Request Forging (CSRF)

Cross-site request forging is a type of a “confused deputy attack”. A hacker tricks a user into loading a link from the hacker’s controlled website to the trusted website that makes them execute something the attackers wants them to. Normally, a hacker cannot let the user access a the trusted web application from the controlled website due to same-origin policy. However, by letting them go to the link containing the right parameters, the action could be unwittingly be executed by the user.

Implementation The vulnerability is implemented by bad design in a Uniface server page One of the pages allows user to send money to each other, determined by the URL query. By sending a link for sending money to the attack to a logged in user, the CSRF is executed.

Detectability The vulnerability is not present in the request dispatcher making it invisible to static scanners. OWASP describes CSRF as easy detectable but not many dynamic scanners feature this attack.

#9 Unvalidated redirect

On many websites users occasionally have to be forwarded to different pages. Usually the location is determined by the server, but when it is implemented badly it could be changed the client. This has the side-effect of allowing a hacker to create a link that forwards the user from this site to an evil website. As this makes the domain and URL be that of the trusted site, this allows them to trick users into thinking the link is trustworthy.

Implementation A forward is implemented by a few lines of JavaScript that uses the URL query as input. As this is purely client-sided, nothing in the DRD or beyond is accessed.

Detectability Impossible for a static scanner as it does not involve the request dispatcher. Dynamic scanners can test for it by using a URL of an extern website as URL query input. As such, OWASP considers this vulnerability easy to find.

#10 Denial-of-Service

A Denial-of-Service attack are infamous as is its bigger version, the “Distributed Denial-of-Service” (DDoS) attack, which makes the news from time to time. There are many ways of overloading a web application. On a proper web server this could only be accomplished by having more resources available, but for badly configured ones hackers could let the server execute functionality that take very long to load.

Implementation Cookies can be converted in the request dispatcher to base-64, a very simple reversible encryption algorithm. A user can then modify their own cookie to be of several megabytes long. This will make the server choke on the method for around ten seconds or more.

Normally the size is checked in the base-64 algorithm, but this check is removed from the DRD. As an additional measure, the default Tomcat configuration only allows requests of few kilobytes in the length, this configuration too was modified.

Detectability A static scanner might be able to sense that several heavy array operations are executed in the algorithm without a length check in sight. The attack is reasonably detectable for a dynamic scanner as it notices the server stuttering. However, most tools do not have the right attack vector to execute this vulnerability.

#11 Information Leakage

Information leakage is showing internal information of the server of web application to the user, which may aid hackers into deciding how to attack the application. Examples can be OS information, server version or even source code leakage.

Implementation An exception is raised in DRD when the right URL is requested. The resulting stacktrace is added to the response and shows up in the body of the HTML page. The trace contains valuable information regarding the DRD. A similar vulnerability was seen in the article by Meganathan et all [9].

Detectability As it done in the DRD and shows up on the web page it should be detectable by both techniques. Not many static scanners support it but many dynamic scanners do. However, this requires them to recognize contents of the stacktrace.

#12 Unrestricted File Upload

Web applications can contain forms for upload files to the server. Those files could contains dangerous script or viruses, thus such file should be validated first before storing it on the server.

Implementation The web application contains a upload form that has no input validation. This vulnerability is a design flaw and therefore not present in the DRD.

Detectability It cannot be detected by static analysis tools. Dynamic scanners have ways of uploading files to forms, but it is hard to recognize that the file was successfully uploaded. Not many dynamic scanners have feature the attack.

#13 Path Traversal

When accessing a page multiple additional HTTP requests are made to access different resources, such as scripts or media files like an image. This is often given by a link to a relative path in the directory structure of the server. By injecting the dot-dot-slash (`../`) sequence into the resource path an attacker could download files outside the proper directory.

Implementation Given in the URL query is a parameter for the language of the site visitor. This directly corresponds to a flag image which is store as `*language-code*.png` on the media directory on the server. By changing the parameter to `../hack.png` the attacker changes the image into a image called `hack.png` that should not be accessible.

In the implementation of this experiment, injecting the dot-dot-slash also means accessing a file on the server. Therefore, in this context the vulnerability path traversal is the same as the local or server side file include vulnerabilities that are seen in many analysis tools.

Detectability The dot-dot-slash is sanitized into the DRD at the same location as for the SQL injection and hence should be detectable. Many dynamic scanners attempt to insert double dots into file paths. However, it is difficult to recognize a successful attempt and is easier performed by humans.

#14 Response Splitting

Another injection method is inputting HTML or String encoded newlines into a request parameter. Some of these parameters are sometimes returned by the server and be interpreted as an additional header parameter. Hence, this attack is also called "header injection" or "carriage return/line feed (CR/LF) injection", as can be read in [15, p.45]. An attacked could potentially make a response look like another page by inserting entire new header lines such as the status code into the response header, effectively allowing them to fake a web page.

Implementation A page returns its URL query back as response headers. The proper sanitation is removed in both the Tomcat server as well as the DRD. Therefore, new headers can be created by inserting new lines into the URL query.

Detectability Response splitting is a common attack pattern by dynamic analysis tools and can be easily executed by injecting CR/LF characters. A sanitation line is commented out in the DRD which should be detectable by static detectors.