Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Comparison and Analysis of Web Vulnerability Scanners

Alexander Lis

Course of Study:	Informatik
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Dipl.-Inf. Guido Schmitz
Commenced:	November 20, 2018
Completed:	May 20, 2019

Abstract

Within the last years the commercial relevance of web applications increased steadily. They developed from simple information sharing platforms to serious business applications like online-banking, e-commerce and social media platforms. Unlike most other technologies, web-based applications are accessible from around the world continuously. Additionally, they are very susceptible for vulnerabilities as there are various technologies interacting. These factors render web applications to very attractive targets for criminals because they are often easy to attack, globally accessible and yield valuable exploits.

As a consequence, much effort was put into research to prevent, detect and eliminate web application vulnerabilities. However manual security audits are time-consuming, costly and demand expert-knowledge. Web vulnerability scanners tackle this problem. They are programs that test web applications for the existence of vulnerabilities. Additionally they categorize and report them. Because these tools work automatically, faster as humans and reduce the necessary knowledge in network security, they became an interesting supplementation to traditional security audits.

On the other side web vulnerability scanners also have their limits. They can not test for the absence of vulnerabilities and thus produce false positives or miss weaknesses. Furthermore previous research has shown that there are also vulnerability classes that are especially intricate to detect like stored SQL injections or stored cross-site scripting vulnerabilities. .

Nonetheless web vulnerability scanners show very much potential and there is a growing interest into automatic web application testing. This is reflected in the increasing diversity of commercial web vulnerability scanners that can be found online. Thus this thesis compares and examines three web vulnerability scanners, namely Acunetix, Arachni and w3af. Focus is set on delineating the current capabilities and limits of state-of-the-art vulnerability scanners.

Contents

1 Introduction

When the British scientist Tim Berners-Lee invented the Web in 1989 at CERN, it was originally conceived as an information sharing platform for academic institutions [CERN19]. Since then, it has developed to an extremely large and commercially used system which is spanning the globe. Web applications like online-banking, social media platforms or video streaming services became widespread. Additionally, its heterogeneity and its complexity increased too, in order to add various functionalities to the system. As a result, web applications became increasingly dynamic and susceptible to vulnerabilities.

Web applications generally pose only very few constraints regarding the time of use, place or person for the end-user. They are accessible at any time and from any Internet access point normally. This design concept is chosen to provide access to the largest possible group of potential consumers. However, in this case the qualities making web applications attractive to consumers also render them to interesting targets for web attacks. Additionally they have few means to differentiate an honest user from a malicious one. Interactions take place by exchanging request/response messages containing only a limited amount of information. Furthermore attackers can use services for anonymous communication to conceal their traces. Another reason which makes detection of web attacks more intricate is that the total amount of exchanged information is considerable thus distinguishing honest incoming requests from malicious ones is difficult.

The information which is processed by web applications is often of considerable value. Therefore institutions which offer web services in fields like online-banking, e-commerce and e-voting have become increasingly security-aware. The large number of upcoming tools that aid for automatic detection of vulnerabilities in websites reflects this recent trend. These automated scanning tools are called web vulnerability scanners or web application scanners. They are programs that test web applications for the existence of vulnerabilities. Additionally they classify and report them. Web vulnerability scanners aid Web developers to create and maintain secure Web applications. They ease developer work because ordinary manual security audits are expensive, time-consuming and demand expert knowledge. Web application scanners on the other side conduct security assessments automatically or semi-automatically while reducing the necessary knowledge in the field of web security. This decreases development and maintenance costs as well as unexpected costs to correct the consequences of a successful web attack after deployment.

Recent research suggests that state-of-the-art web application scanners improved [IG17]. However many of these tests were conducted using well-known web applications that are used for web vulnerability scanner assessment. web vulnerability scanners (WVSs) implement several approaches in automatic security testing like black-box and white-box scanning. They were found to have difficulties to cope with dynamic and interactive content [VMN15]. Especially Flash and complex Javascript procedures were found to impede the scanners [DCV10].

This thesis analyses and compares a selection of Web vulnerability scanners. We evaluate Acunetix, Arachni and w3af using a new test application to delineate current capabilities of web application scanners and examine their limits. For this purpose we created a social media platform which we call Vulnerable Social Network. The rest of this thesis is structured as follows:

In the next chapter we will summarize basic concepts of Web applications and underlying technologies. In Chapter 3 we will discuss Web vulnerabilities and introduce a classification of Web application weaknesses. Chapter 4 examines WVS in more detail and discusses underlying concepts. The experimental arrangement is explained in chapter 5. Moreover we give an functional overview and explain design decisions of the web application. In the following chapter 6 we present the statistical results of our experiment. Furthermore we evaluate and discuss the results further. Section 7 provides an outlook about interesting research topics with a connection to Web application testing. Chapter 8 concludes this paper by emphasizing central insights we gained from our examinations.

2 World Wide Web

Talking about the Web in its simplest form we refer to the global collection of HTML-Documents that are connected by Hyperlinks. The purpose of Hyperlinks is to route users from one site to another. Thus they provide a means to structure information in relational pattern, while the information itself is provided by servers located in the Internet. This is necessary as one is alternatively required know or resolve the specific address of a server that contains the information in order to access it. Accessing in this case means retrieving an document that is stored on a web server like Apache or Nginx by sending a request to it and displaying the response.

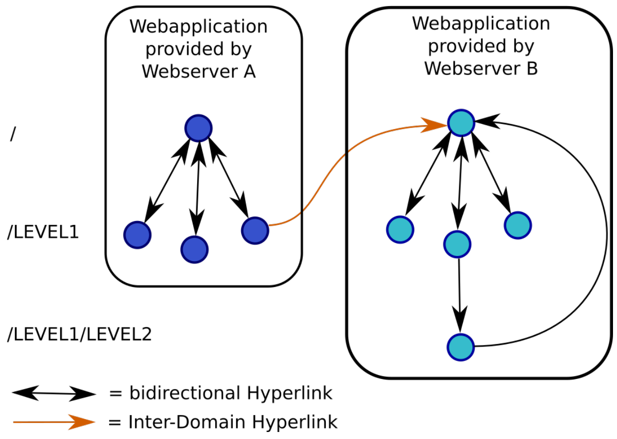

Webapplication provided by Webserver A
Webapplication provided by Webserver B
/
/LEVEL1
/LEVEL1/LEVEL2
= bidirectional Hyperlink
= Inter-Domain Hyperlink

Figure 2.1: The diagram shows the Hyperlink structure of two small Web applications. The website on the left contains one link to a page of the right website.

However providing data statically has crucial disadvantages. For instance retrievable documents need to be preprocessed to provide possibilities for customization or integration of external information. As a consequence server-side scripting languages such as PHP: Hypertext Preprocessor (PHP) and Active Server Pages (ASP) were introduced. However web applications also need to be very responsive on the user-side in order to improve user experience and usability. This problem was tackled by user-side scripting-languages like JavaScript. Later technologies like Asynchroneous JavaScript and XML (AJAX) were introduced to extend the functionalities of modern web applications further.

If the content of the web page is created dynamically, the HTTP response depends on the value of several Hypertext Transfer Protocol (HTTP) request parameters. Possible inputs that influence the server's response and origin from the server-side are the database state and its file system. However there are numerous other factors that can affect the output like server-side time. On the client-side the server considers several information from the HTTP-Request like headers, cookies, the HTTP-method and the requested URL.

2.1 Web Applications

Fong et al. introduced central definitions and functions regarding web applications and WVS [FO07]. They define web applications as `software applications, executed by a web server, which respond to dynamic web page requests over HTTP`. In this thesis we want to resort to their definition. Figure 2.2 depicts a web application's general architecture.

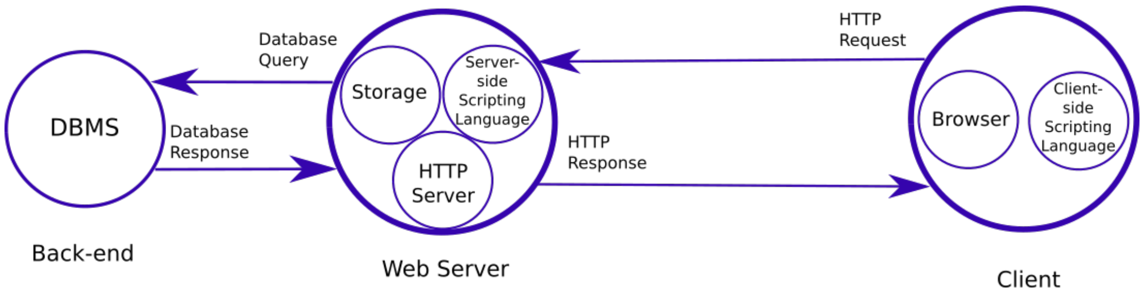


Figure 2.2: Visualization of the structure of a web application. Arrows represent possible interactions between individual components.

Alzahrani et al. pointed out that modern web applications are constituted of three program layers [AAZ+17]. The first layer comprises a client side program that is able to submit HTTP-Requests and properly process incoming responses. This program is called a browser and needs to be understand client-side scripting-languages such as JavaScript. The second component is a Web server program. A Web server listens for incoming client requests, processes them properly by executing server-side scripting languages until it finally returns a response to the client. The generated response is affected by web server's local storage. Lastly, there is a back-end which comprises components like Database Management Systems (DBMSs) that are used to store data. The web server communicates with the back-end by sending queries to the DBMS. These requests are processed by the back-end and then returned to the web server. There is no direct communication between the client and the back-end.

2.1.1 Client-Server Communication

From the perspective of a client (and likewise an attacker) the server behaves like a black box. Normally, he has no information about the underlying source code. He can only extract information from the web application by interacting with it and closely observe its response. The analysis of the web server's behavior is further complicated by the server's internal state machine. The representation of its state comprises the state of many components such as the local filesystem, the content of the databases, or the the current time. An abstract visualization of a web server's state transition function is depicted in Figure 2.3. The interaction is based on the request response architecture however the web server does not send requests towards the client. The protocol which is mainly used for communication is HTTP respective Hypertext Transfer Protocol Secure (HTTPS). Because these protocols are stateless the client may need to store information locally in cookies. For instance a user might save a session token or the content of a shopping cart of an e-commerce web application in this way. Storing information on both sides complicates the system's architecture further and increases the risk for existing vulnerabilities.

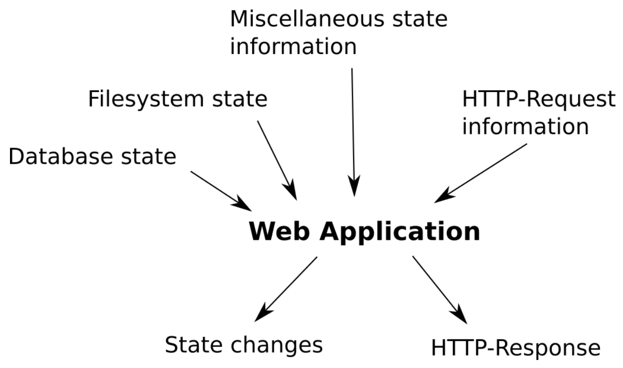


Figure 2.3: Abstracted Visualization of Input-Output-Flow of a Web Application.

2.2 Security of Web Applications

According to the latest security report of Symantec the total number of web attacks they detected in 2018 was more than 1.3 million [Sym19]. This means an increase by 56 percent compared to 2017 [Sym19]. They also determined in the same report that formjacking attacks were one of the big trends in 2018. In a formjacking attack an adversary uses a JavaScript weakness to steal sensitive information from forms like payment forms. Alzahrani et al. mention the maturity of network level protocols as one possible reason for the growing interest in application layer and especially web attacks [AAZ+17]. According to them attackers have to search in other layers for vulnerabilities. Another aspect which seems to influence the security of web applications is their increasing complexity [AAZ+17]. This development is a result of a growing demand for features on the one hand. On the other hand Alzahrani et al. also assume that the increasing heterogeneity is another influential factor [AAZ+17]. As a result the probability for existing software weaknesses is high. Fonseca et al. noticed that the security of a web application is closely linked to its robustness against malformed or unexpected input [FVM07]. Thus security tests are a special case of robustness tests for software applications [AV12].

2.3 Heterogeneity

Web applications apply various technologies like server-side scripting languages such as php or client-side scripting languages like JavaScript. Often DBMS like MySQL-Servers are used too. As a result of this heterogeneity the complexity of web applications increases. Web developers not only need to understand the unique properties of a single scripting language, they need to know them for a multitude of languages and applications. Furthermore, developers need to understand the interactions of those systems because a plain and secure string as an output of one technology can be a crucially misinterpreted input to other execution contexts resulting in harmful behavior of those applications.

3 Web Vulnerabilities

The Common Vulnerabilities and Exposures (CVE) standard defines vulnerabilities as locations in the computational logic that allow violations against specific security properties like confidentiality, integrity or availability [Vuln19]. In this chapter we will introduce a typology of web vulnerabilities which we will use to precisely discuss our test results in Chapter 6. The classification is deduced from the vulnerability classification in the Open Web Application Security Project (OWASP) Top 10 list from 2017 [OWASP19]. However we omitted some categories and unified others.

3.1 Existing Vulnerability Taxonomies

There are several existing web vulnerability classifications and enumerations online. Many of them are not restricted to web vulnerabilities but also list general application weaknesses. The Common Weakness Enumeration (CWE) is a community-developed list of software weaknesses that contains around 800 different types of vulnerabilities [CWE19]. CWE provides three types of hierarchical arrangements over their listed vulnerabilities. Hierarchies can be build based on research, development or architectural concepts.

OWASP is another website that comprises valuable information about security of web applications ranging from classifications of vulnerabilities and attacks to the OWASP Top 10 list which enumerates the most important vulnerability types according to their security risk [OWASP19]. The risk of a vulnerability is deduced from the likelihood that such a weakness exists and the potential impact of an exploit [OWASP19]. Because the classification is derived from a representative set of real world applications, the OWASP Top 10 nomenclature is very widespread. Several WVSs have been adapted to it and provide operating modes or reports that are customized to OWASP's vulnerability types. According to the list the 10 most critical web vulnerabilities from 2017 are:

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities
5. Broken Access Control
6. Security Misconfiguration
7. Cross-Site Scripting
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities

10. Insufficient Logging and Monitoring

There are several advantages that come along with such databases. Firstly, these databases serve as a common terminology [CWE19]. This is important as it allows us to precisely discuss web vulnerabilities and compare web vulnerability scanners. Without a common language it would be cumbersome to determine which tool is more appropriate for a particular use because their reports might define attack types differently. Additionally they provide a guideline for security audits, supporting the identification and prevention of vulnerabilities in software [CWE19]. Moreover a common classification helps to develop a precise understanding of relations between different vulnerabilities. Being able to arrange vulnerability classes in a hierarchical order would imply that we have an advanced knowledge about specialization and generalization relations. Many web application scanners include links to articles of such databases in their reports to provide end-users with information about existing vulnerabilities.

3.2 Classification

Conceiving a well-defined taxonomy for Web vulnerabilities is a difficult task. The problem's complexity seems to be derived from the heterogeneity and layered architecture of modern Web applications. A practical property of a classification would be that all elements within the same class can be fixed using related techniques. However terminologies that distinguish vulnerabilities by constitutive properties are quite common [CWE19; NVD19; OWASP19]. Because OWASP's classification is the most widespread, we will refer to it in a slightly adapted way in this thesis. Figure 3.1 shows an high level overview of the customized vulnerability categories that are relevant in our tests.

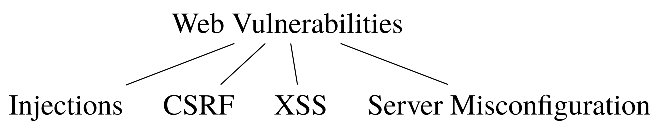


Figure 3.1: High level overview of several vulnerability classes the scanners were supposed to detect. The hierarchical structure reflects their specialization/generalization relations.

3.2.1 Injection Vulnerabilities

Injection vulnerabilities exist when an attacker is able to include malicious code via insufficiently sanitized user input into an existing script that is eventually executed. According to the OWASP enumeration injection vulnerabilities are the most significant class of web application weaknesses when we consider the risk and impact of a vulnerability [OWASP19]. Injection attacks can occur in many different contexts, ranging from Database Query Languages (SQL, NoSQL) and operating system scripts to protocols like Lightweight Directory Access Protocol (LDAP) or XPath. Web applications rely on a multitude of different interpreters that are interacting, each one with a distinct execution context. As a result, injection vulnerabilities may occur easily during interactions. Generally, the consequences of injections are severe. They range from information disclosure to a system's compromise in the worst case [OWASP19]. We can prevent injections by sanitizing

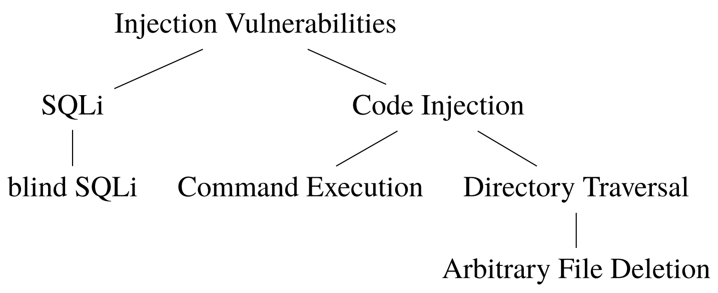


Figure 3.2: Tree-diagram of the injection vulnerabilities the scanners were supposed to detect in our test application.

user input before including it into scripts. However, because we have different execution contexts, different sanitization procedures need to be applied. Figure 3.2 shows an overview of the injection vulnerability types that were included in our test application.

SQL injection vulnerabilities

SQL injection (SQLi) vulnerabilities occur where a Structured Query Language interpreter receives a query that is amended with insufficiently sanitized user input. In a successful SQLi attack an adversary may be able to disclose sensitive database information, alter existing database records or compromise existing authorization mechanisms [OWASP19]. Compared to other vulnerabilities SQLi weaknesses are relatively easy to prevent. It is assumed that prepared statements prevent any form of SQLi attacks. Prepared statements are directives which allow the programmer to label locations in the query where escaped user input will be inserted.

By default the interpreter of a database query language returns an error message when a query contains syntax errors. Normally these error messages are supposed to provide valuable information to the developers for debugging. However publicly displayed messages are also very interesting for adversaries because they indicate when an input provoked an error. As a consequence public error messages provide an easy way for adversaries to check whether a SQLi vulnerability exists. To prevent this leak of information modern web applications hide error messages. Blind SQL injection (bSQLi) vulnerabilities are SQLi vulnerabilities that do not return error messages when the syntax of the underlying query is incorrect. Thus bSQLi vulnerabilities are much harder to detect and more difficult to exploit than traditional SQLi.

Code Injection Vulnerabilities

Code Injection (CI) vulnerabilities occur when server-side scripting languages as PHP or ASP execute code that is completely or partially derived from user input without proper sanitization. They differ from SQLi vulnerabilities in the capabilities of the interpreter. Generally the server-side script interpreter has fewer limitations because it is able to read from and modify the file system. Additionally it is able to submit database queries. A code injection vulnerability may also be exploited to create a Command Execution (CE) vulnerability.

Command Execution vulnerabilities are weaknesses in web applications that can be used to execute arbitrary Operating System (OS) commands in the corresponding shell. They may be caused by an existing code injection vulnerability that can be escalated by injecting a string that elevates program execution to a system shell. The impact of CE vulnerabilities is fatal as the corresponding interpreter is the operating system's shell. While using the remote shell an attacker could try to establish a backdoor to the system and try to infect adjacent machines [AcuCI19].

Directory Traversal (DT) Vulnerabilities are a special case of code injection vulnerabilities too. In a DT attack a server-side script includes user input into a path that is used to load or manipulate resources. One countermeasure against directory traversal attacks is filtering of the user input for particularly dangerous characters such as `“.”, “/” and “\”`. However some filter-mechanisms can be bypassed by using URL encoding for instance. In the context of a directory traversal attack the underlying OS is relevant as the file systems and file path formats of Windows and Unix machines differ.

An arbitrary File Deletion (FD) vulnerability occurs when user input is appended to an argument for a delete-function like the unlink-instruction in PHP [AcuFD19]. If such an input is passed unfiltered, an attacker can use character sequences to perform a DT to delete arbitrary files. FD vulnerabilities can be used to weaken the implemented security mechanisms by deleting files that are used to implement it. For instance .htaccess files or security plugins might be targets of such attacks [wp18].

3.2.2 Cross-Site Request Forgery Vulnerabilities

Cross-Site Request Forgery (CSRF) vulnerabilities allow an attacker to trick an authenticated victim into submitting specially crafted requests that trigger unintended actions of the victim. Password resets, financial transactions and other state changing actions are examples of functions that have to be protected from CSRF attacks. The root cause of this vulnerability is that browsers normally append the authentication cookie to any request that is transmitted to the web application which originally set the cookie. This is also true for requests that are sent because an attacker tricked a victim to submit it. CSRF attacks can be performed via crafted links or hidden links as well as manipulated websites that automatically submit form data to the vulnerable web application. In order for a CSRF attack to be successful, the attacker has to know valid parameter values for all parameters that are necessary for the intended action. CSRF poses few requirements on the technology that is used by the system under attack. On the other side the attack can only be performed when the user is authenticated.

3.2.3 Cross-Site Scripting

Cross-Site Scripting (XSS) vulnerabilities allow an attacker to inject malicious javascript code into the HTTP-response of the webserver in a way that this code is later executed by the victim's browser. Frequently we distinguish three types of XSS: reflected Cross-Site Scripting (rXSS), persistent Cross-Site Scripting (pXSS) and DOM-based Cross-Site Scripting (DOM-XSS) [GG17]. Figure 3.3 shows their relations in the classification. All XSS vulnerabilities have in common that they are limited by the abilities of the client-side's scripting language which is normally JavaScript. Thus successful XSS attacks allow an attacker to access cookies that are associated to the website, leading to a compromise of confidentiality on the one hand. On the other hand JavaScript is also able to manipulate the Document Object Model (DOM) likewise resulting in the loss of integrity. As Acunetix pointed out, an attacker may also be able to access sensors like the camera or microphone of the victim's machine via HTML5 APIs using XSS [AcuXSS19]. This attack class can also be used as a stepping stone to exploit other vulnerabilities such as CSRF. Normally XSS can be prevented by proper escaping of the user input. The appropriate escaping procedure depends on the location where the user input is inserted. For instance, an injection into a document's body tag requires different sanitization as user input that is inserted into an HTML source attribute [OWASP19]. For further information about XSS the Acunetix website contains a very interesting article [AcuXSS19].

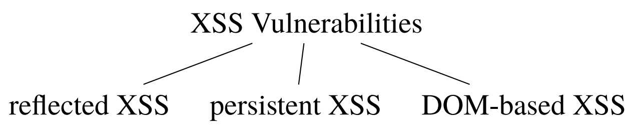


Figure 3.3: Tree-diagram showing the classification of XSS vulnerabilities

Reflected Cross-Site Scripting

Reflected Cross-Site Scripting vulnerabilities occur when a web application processes the parameters of an HTTP request improperly. Concretely, the web application includes the parameters without proper escaping into the corresponding HTTP response such that an attacker is able to inject valid JavaScript code. Because an attack does not leave obvious traces in a server an adversary might be able to secretly test a web application for the existence of an rXSS vulnerability and potentially exploit it without anyone noticing.

Persistent Cross-Site Scripting

In contrast to rXSS a pXSS vulnerability does not necessarily process the user input in the subsequent response. Instead, in a first step the user input is stored persistently in a database or somewhere else in the server's state. Later this data is retrieved to create a new response for another client request. However since the server's answer is influenced by the previously submitted data, the response may contain malicious JavaScript code that an adversary intentionally stored in the web server.

DOM-based Cross-Site Scripting

DOM-based XSS is the third kind of XSS that is distinguished in literature [GG17]. In a DOM-XSS vulnerability the final malicious code is injected by the victims web browser and not by the server. Thus these attacks require two injection steps. As a consequence DOM-XSS is more difficult to perform and detect than the other XSS variants. DOM-XSS vulnerabilities are based on functions that manipulate the DOM during script execution.

4 Web Vulnerability Scanners

Web vulnerability scanners (WVS) or Web application scanners are programs that test web applications for existing vulnerabilities and return their findings as a vulnerability report to the user. We distinguish web application scanners by differentiating the abstraction that they apply to the tested program. Currently there are two main approaches differentiated. They are called white-box and black-box testing.

4.1 Black-Box Scanners

Black-box WVSs are applications that test their targets by interacting with them. Their dynamic approach reduces web applications to their behavior while assuming the source code is not available. Thus black-box scanners take an external view of the application like an attacker or end-user [FVM07]. To be more precise, black-box WVS analyse web applications by comparing the expected response with their actual output to a request. Modern scanners conduct tests by sending HTTP requests to an deployed server normally. However black-box testing can also be applied at several other layers of abstraction like unit or integration tests [AV12]. Generally small scale black-box tests can be performed either automatically or manually. However in large scale scans automated black box scanners save time by executing repetitive and cumbersome tasks automatically [AV12]. Considering the context of other testing methods the scanners implement a specialization of robustness testing which focuses on inputs that compromise security.

As we will see, the common architecture of these programs relies on the idea that we can divide the complex task to test a web application for security into several subtasks. As a consequence modern WVS are designed according to a modular architecture [DCV10]. Furthermore the modular design eases application's maintenance and provides easy extensibility. Figure 4.1 depicts the widely adopted structure of black box vulnerability scanners.

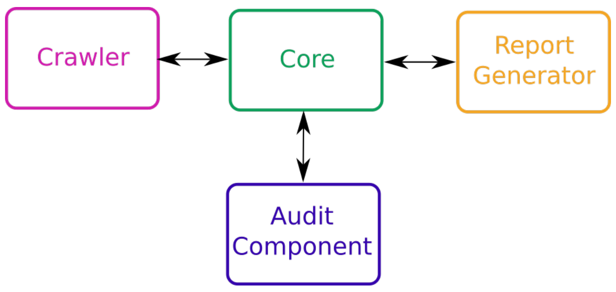


Figure 4.1: Modular structure of a black-box web vulnerability scanner. The arrows represent communication paths.

A core component monitors and coordinates the activity of the other modules that actually conduct the security test. The first component that actively participates at a web application scan is the crawler. Its purpose is to navigate through the web application and collect as many valid Uniform Resource Locators (URLs) as possible for the audit phase. In the audit phase the audit component tests the URLs that were collected in the crawling phase for vulnerabilities. Lastly the report component summarizes the information about the scan in a test report. The next sections discuss the different components more precisely.

Core Component and Configuration

The WVS's core orchestrates execution of the other components and monitors the scan. Furthermore web application scanners often provide various configuration functionalities. For instance, the tester may enable or disable the search of particular vulnerabilities. Except for that, one can specify technologies which are used by the web application to support the scan. However most WVSs detect these technologies automatically by a precise analysis of the web server's responses. Furthermore vulnerability scanners provide a means to prevent the scanner from executing undesirable actions. State changing actions like a logout function or a function to delete a user account are instances of unwanted actions. This function is crucial as it is able to prevent a web application scanner from triggering actions that would otherwise compromise the scan like a permanent logout.

The automatic establishment and maintenance of a valid session is another central topic to Web scans. Without successful authentication a scanner may only be able to test the parts of a website that are publicly accessible. During our research we encountered the following authentication methods:

- **Form Authentication:** Requires the tester to specify the exact location and content of the login form. Furthermore the user needs to specify a URL and a corresponding substring that emerges in the response if the scanner is authenticated. The second input is necessary to guarantee that the scanner does not quit the session during the test.
- **HTTP Authentication:** Affords submission of username and password in HTTP request headers.
- **NTLM Authentication:** Provides authentication via Microsoft's NTLM protocol
- **Login Sequence Records:** Proprietary technology of Acunetix that can be used to record manual logins and specify intervention points to manually pass obstacles like CAPTCHAs. Furthermore the Login Sequence Recorder (LSR) detects session validation patterns automatically during a record. Additionally user can specify undesired actions by first executing and then labeling them in a context menu. The login sequence has to be recorded before the scan is initiated.

Crawling Component

The crawler component is supposed to provide the application endpoints that are tested in the audit phase. It receives an initial URL as an input which serves as a starting point for its navigation through the Web application. Modern scanners traverse web applications by following available links that are situated on known pages. Thus crawlers follow static links on the one hand while they also navigate to the targets of dynamically created URLs on the other hand. Prior research has shown that especially dynamic URLs are challenging [PTBR15]. However other work has shown that the majority of URLs in modern web applications are derived dynamically using JavaScript which amplifies the problem [ZD12]. Additionally the spread of other dynamic technologies like Flash and AJAX creates new crawling challenges.

The web application's statefullness is another aspect that crawlers have to consider. Otherwise, careless crawling through the web application could instigate unwanted side effects resulting in compromised crawling results. This problem refers to a generalisation of the authentication issue which was mentioned previously. For instance we can think of an e-commerce application that requires an user to fill his shopping card before the button which redirects him to the check-out is included in a response.

Furthermore we can distinguish two different crawling approaches. The first one is text-based and searches the responses for URLs. Additionally browser components assist the crawling components by executing client-side script and triggering events [BGBK11].

Audit Component

The audit component takes the URLs which were discovered by the crawler as an input and tests them for the existence of vulnerabilities. Therefore the scanner submits a combination of predefined and fuzzed HTTP requests to the endpoints and subsequently analyzes the responses. The audit component itself can be subdivided into modules that are specialized for distinct vulnerability types.

Regarding the detection of injection and xss vulnerabilities the submodules contain several predefined test strings and fuzzing techniques that are designed to trigger extraordinary behavior. Furthermore modules implement an individual evaluation function which decides for each test if the scanner is sufficiently convinced of the existence of a vulnerability. Many web application scanners mainly use two different approaches to calculate a confidence value. They are called differential and timing analysis. Differential analysis is performed by comparing the response to the input that is assumed to be secure with the response to a malicious input. For instance Balduzzi et al. adapted a pattern matching algorithm from Ratcliff/Obershelp [RM88] for the detection of HTTP Parameter Pollution (HPP) vulnerabilities [BGBK11]. The timing analysis approach tries to inject code that causes an delay in the program execution. If a vulnerability exists, this delay can be detected in the Web application's response time. Nonetheless, because of the Scanner's lack of information about the underlying program logic, a black-box scanner can never make irrevocable statements.

Report Component

After the crawling and audit components finished the scan successfully, the report component summarizes their results in an output file. Sophisticated scanners provide different classes of reports which are each adapted for a distinct audience. The Acunetix WVS for instance provides reports for developers and executives [Acurep19]. While the executive's report focuses on a clear overview and risk assessment, the developers report is more verbose and includes tips to fix the security issue and the location of the vulnerability. Besides an overall information, such as an evaluation of the web application's security as well as statistics about the amount of findings, ordinary reports normally comprise the following information about each distinct vulnerability:

- Identification which is conformant to CWE or a similar classification
- Severity level as an assessment of the potential impact
- Location of the vulnerability as a filename and potentially with a line number
- Request which successfully exploits the vulnerability (as a proof of existence)
- Hints regarding prevention mechanisms
- References to further literature concerning the vulnerability

Black-Box Vulnerability Scanner Implementations

There is a great number of commercial and open-source black-box WVS available. Websites like the sectoolmarket website [SecMa16] or the website of OWASP [OWASPwvs19] foster lists containing commercial and open-source scanners. Some well-known commercial web vulnerability scanners are the Acunetix WVS, AppScan from IBM, Netsparker from MavitunaSecurity, WebInspect from HP and Burp Suite from PortSwigger.

Apart from these, there also exist several open-source/public-source WVSs. Some prominent examples are w3af, Zed Attack Proxy and Arachni. These tools often have a strictly modular architecture which we discussed in Section 4.1. In most of their cases this structure eases the open-source-communities effort to extend the tools.

Furthermore there are also numerous scanners that were developed by researchers. Often these implementations were developed as a proof of concept. Thus they are restricted in their functionality. Shariar et al. provide a great overview about scientific efforts for black-box as well as white-box and grey-box testing tools [SZM09]. Their listing focuses on scanners designed to detect XSS, SQLi, format string or buffer overflow vulnerabilities until 2009. Some more recent scanners are mentioned later in this thesis.

4.2 White-Box Scanners

White-Box WVSs are programs that examine the static source code of a web application to detect vulnerabilities. Thus these scanners reduce the application under test to a formal description. Subsequently, white-box scanners take a diametral point of view compared to black-box scanners. They examine the application from the perspective of a developer or software architect. Disregarding nondeterminism white-box scanners thus have perfect information. Consequently they are able to prove their statements and draw absolute conclusions. Thus they are also able to avoid false positives [AV12]. Antunes et al. differentiate white-box scanners depending on their sophistication regarding how accurate the dependencies between instructions are modeled [AV12]. According to them, there are scanners that restrict themselves to individual lines of code and scanners that take dependencies between instructions into account.

Similarly to black-box scanners, complexity is a considerable factor that limits white-box scanners. However in comparison to black-box scanners this complexity is represented in the application's source code rather than in its dynamic behavior. Because white box scanners are dependent on a precise model of the underlying source code, they are specialized for a single scripting language such as PHP [DCKV12]. Thus one white-box scanner is not able to test web applications which are using technologies that are not modeled. This might be one of the reasons for the dominance of black-box scanners over white-box scanners on the market.

4.2.1 White-Box Vulnerability Scanner Implementations

Nonetheless there are several white-box scanners that were developed by researchers to demonstrate the correctness of their approaches. Jovanovic et al. implemented Pixy a XSS detection prototype tool to demonstrate the success of their flow-sensitive, interprocedural and context-sensitive static source code analysis [JKC+06]. The underlying idea of their approach is to treat user input as tainted data that should be sanitized before being inserted into sensitive execution points that they call sensitive sinks. Their tool identifies potential vulnerabilities by tracing tainted variables and the operations performed on them to check whether unsanitized input to sensitive sinks may occur. Pixy is adapted for the server-side scripting language PHP. According to them, their approach could be expanded to detect various classes of taint-style vulnerabilities like SQLi and command injection [JKC+06].

Another approach was proposed by Medeiros et al. [MNC16]. They combined taint analysis with principles from data mining. Taint analysis was used to detect variables that were potentially delivering unsanitized user input to vulnerabilities. Additionally data mining ideas further decreased the number of false positives. Based on their combined approach they proposed an idea of automatic error code correction by correcting unsafe instructions in the source code. The tool WAP is a prototype implementation of their ideas. Concluding tests to compare their tool with other white-box testing scanners suggested an improvement in accuracy and precision.

4.3 Automated Web Vulnerability Testing in the Software Development Cycle

Gioria et al. [Gio09] and Idrissi et al. [IG17] pointed out that its best to test a web application regularly, beginning in early development stages of the software development cycle. Thus this proactive and non-responsive approach saves costs as the complexity of the web application increases during development phase and potential vulnerabilities in a released web application can have fatal ramifications. Even after the development stage regular security audits of the web application should be conducted as new vulnerabilities and attacks against existing technologies are devised continuously.

4.4 Limitations

Previous research already addressed some evident limitations of black-box scanners. The first thing to take into account is the statefullness of modern web applications which was adressed for instance by Doupe et al [DCKV12]. In their work they prototyped a state-aware black-box scanner that gradually constructs a model of the web applications state by observing the application's responses [DCKV12]. Their key idea was to model the web application as a Mealy machine which was build successively. For the detection of different states they applied the heuristic that a state change occurred when the same request produces the same response. As a proof-of-work they developed a prototype scanner which was able to discover more vulnerabilities than other scanners at that time.

Another restriction of the black-box approach represents its inability to safely decide whether a vulnerability exists or not. This is based on intrinsic restriction of black-box testing. Conclusions about the existence of vulnerabilities are made by observing the Web application's behavior applying heuristics to the responses.

Furthermore current WVS primarily focus on rather the technical web vulnerabilities that are introduced during implementation. Other vulnerabilities like design or program logic flaws are neglected. This problem was briefly addressed in Doupe et al. [DCV10] since no tested scanner was able to detect a distinct logic flaw in their WackoPicko Web application.

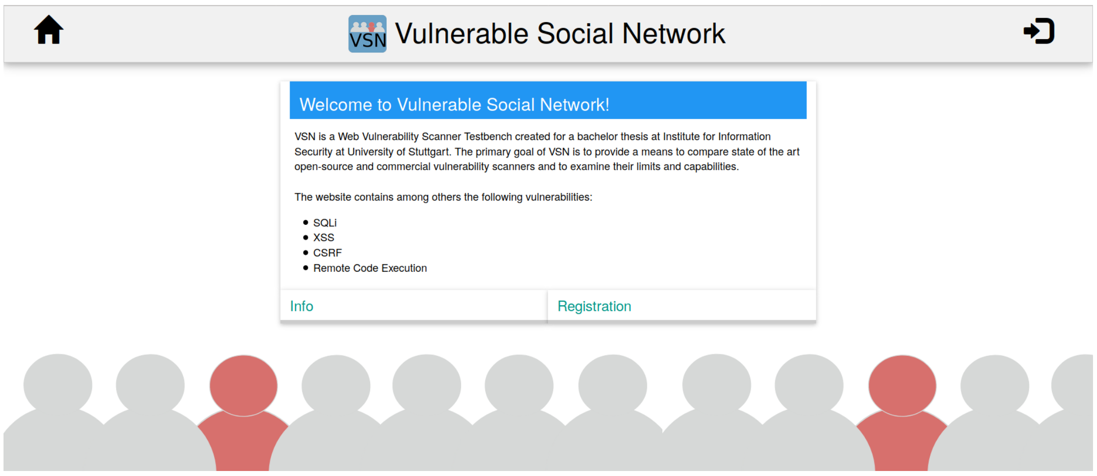
5 Experimental Arrangement

For our experiment we set up an Ubuntu Server machine that was running an Apache Web server. PHP was used as the server side scripting language and the back-end was implemented using a MySQL DBMS. Regarding the web vulnerability scanners we used Acunetix WVS in its online implementation, Arachni and w3af. These tools were instructed to scan the web application we created for this thesis.

5.1 Vulnerable Social Network

As a result of our development work we created a vulnerable Web application which we called Vulnerable Social Network (VSN). VSN is intended to represent a vulnerable social media platform that allows honest users to use it as an application to chat with friends and meet new people on a small scale. Figure 5.1 depicts the homepage of the platform. However for dishonest users it provides an playground to exploit vulnerabilities. The web application was conceived with two additional aspects in mind:

1. **Challenging crawling structure:** the application was designed to pose several challenging tasks for the crawlers.
2. **Existence of a Web application logic:** to test if the scanners are able to detect faults that require to understand the application logic.



Welcome to Vulnerable Social Network!

VSN is a Web Vulnerability Scanner Testbench created for a bachelor thesis at Institute for Information Security at University of Stuttgart. The primary goal of VSN is to provide a means to compare state of the art open-source and commercial vulnerability scanners and to examine their limits and capabilities.

The website contains among others the following vulnerabilities:

- SQLi
- XSS
- CSRF
- Remote Code Execution

Figure 5.1: Screenshot of the VSN homepage.

Although there are already several vulnerable web applications for the purpose of evaluating WVS online, we decided to develop a new web application because of two major reasons. Firstly, existing WVSs test platforms are well-known. Web vulnerability scanner developers are aware of that too. Therefore one can assume that developers tend to optimize their scanners for security audits of popular test websites in order to guarantee their scanner a better ranking result in online comparisons. Another point is that by developing a web application from scratch, we can design the application to comprise vulnerabilities that are particularly interesting for our research. Furthermore we have a very good understanding of the functional aspects of the web application. This is advantageous when we try to analyze and evaluate the results.

VSN - Architecture and Functionalities

Figure 5.2 depicts an overview of the VSN platform's structure. As we can see in the Link-Diagram, the homepage (which is marked as a green dot) allows an arriving user to perform one of three actions: login, registration or a change of the password. After a successful authentication an user is redirected to the main part of the application that basically consists of five distinct subpages. A user can navigate between the five pages arbitrarily. The following list briefly explains the most important functionalities of these pages:

- The profile page provides an user the possibility to adjust publicly available user information like profile image, age, profession, etc.
- On the chat page a user can send and receive new messages as well as react to friend requests.
- The standard search page lists all existing user profiles and contains a search bar that filters usernames according to the inserted string.
- The seed-search page implements a function that receives a string as an user input which influences a Pseudo Random Number Generator (PRNG) that outputs another account.
- The transfer page allows an user to send points to peers that can be used to unlock additional emojis for the chat.

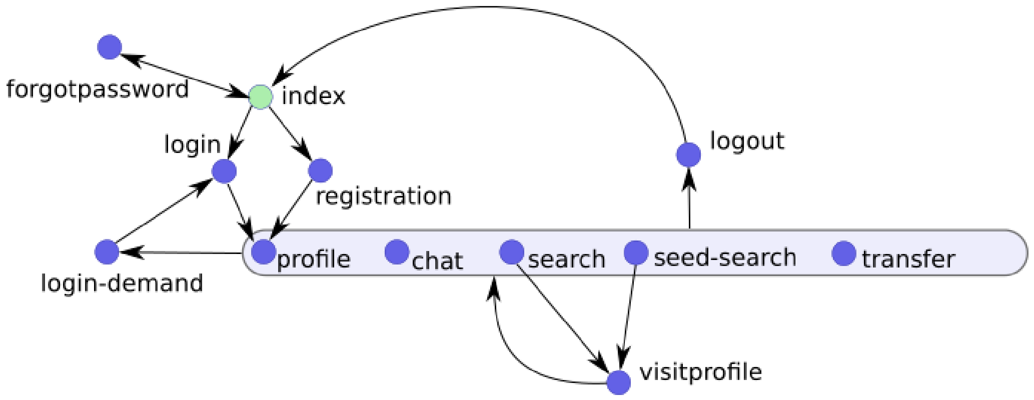


Figure 5.2: Structure of the VSN Web application. Nodes represent pages of the application. The entry point was emphasized with a green node. A light blue bar indicates that the pages are connected as a complete network of arrows. Arrows originating or aiming to the closure refer to all 5 included subpages

Furthermore the search and seed search pages refer to a page where one can visit other user's profile pages. Finally there are a login-demand and a logout page. The login-demand page is shown when an unauthenticated user tries to access protected pages.

VSN aims to test for the limits of modern web application scanners by using technologies such as JavaScript and jQuery. The intention was to show if WVSs are now able to cope with these dynamic technologies. Additionally VSN was conceived to contain challenging crawling features that are based on event handling.

Furthermore we wanted to test how WVSs behave when confronted with simple logic tasks that are no problem to detect for a human penetration tester. Therefore we implemented a chat function that requires a user to exchange friend request before it is enabled. Additionally we designed the user accounts to be unable to visit the page that is displayed to other users. In this case an human tester would simply create a dummy account to see the consequences of changes in the profile and activate the chat function by exchanging friend requests.

We implemented the Web application in three different variants to which we will refer as VSNg, VSNp and VSNf. After conducting the tests on the VSN variants we designed additional tests to further examine the reasons why the scanners missed particular vulnerabilities.

VSNg - VSN Variant Which Primarily Uses GET Requests

The first version of VSN relies on GET-Requests as much as possible. Therefore we will denote it as VSNg subsequently. VSNg contains several plain bSQLi vulnerabilities that were distributed across the application. Some of these bSQLi vulnerabilities were situated at AJAX endpoints and one vulnerability was actually displayed in the browsers URL search bar. Consequently that vulnerability was particularly easy to discover and exploit. Additionally we included several persistent and one DOM-XSS weaknesses in the profile and search page. Another rXSS vulnerability was implemented as a hidden parameter. Furthermore we omitted the CSRF token in some forms. They were counted as CSRF vulnerabilities if they did not represent a more specific attack. In the other case we only considered the vulnerability that was more specific, which means it required the lack of the CSRF token. Additionally we included some URLs which omitted user authentication and accepted arbitrary requests that induced vulnerabilities in the business logic. For instance a malicious user was able to initiate friend requests between arbitrary users by specifying the sender and receiver in a crafted message. We classified these flaws as directory traversal attacks because that was the underlying vulnerability needed for the exploit that could be detected. Except for that we integrated a code execution vulnerability into the custom search function of the application. Lastly there was one FD vulnerability situated in the function that removed user images.

VSNp - VSN Variant Which Primarily Uses POST Requests

In the second variant of the application, GET requests were substituted by POST requests when their submission triggered a state change. Additionally we altered the locations of some bSQLis vulnerabilities to increase the necessary crawling effort to detect them. Furthermore we introduced an additional rXSS vulnerability in the chat function and increased the number of CSRF vulnerabilities. The code execution vulnerability was now reduced to affect only one of the received parameters and the set of allowed characters was reduced to a reasonable set that still allowed dangerous commands. Lastly we added one directory traversal vulnerability which allowed to upload images for arbitrary users.

VSNf - VSN Variant Which Applies Filtering

VSNf implemented some insufficient sanitization and filtering functions. Additionally we added CSRF tokens that were easily guessable for human penetration testers but might be difficult to detect for the automated scanners. Incremental patterns or fixed CSRF token are instances of the vulnerable mechanisms we used to protect some requests.

Additional Tests

Lastly the most interesting scenarios were extracted from the web application and included into several single page web applications. This was done to determine why the scanners missed these vulnerabilities. In this test row we primarily focused on different insufficient filter mechanisms for rXSS and examined the scanners performance when they were encountering several types of CSRF token implementations. We also added one single page application that contained a visible CSRF token which was supposed to trigger a false positive. Nonetheless, we also included test cases for the remnant vulnerability types.

5.2 Acunetix

Acunetix is a commercial web application scanner with black-box and white-box characteristics [Acu19]. Additionally the scanner also extends ordinary vulnerability detection features by supported network scanning functions that allow security audits of other network protocols such as Teletype Network (Telnet), Secure Shell Protocol (SSH) and File Transfer Protocol (FTP). To test Web applications for a wide range of client-side technologies Acunetix supports web application crawling with different user agents. Furthermore the web application scanner exports scan reports in various types ranging from developer reports to executive summaries or a simple listing of the affected items [OWASP19]. Additionally, test specifications can be adjusted according to the end user's needs. Restrictions to specific vulnerability classes such as XSS vulnerabilities are possible. Restrictions to the tested vulnerabilities are useful because they are able to reduce the scan time significantly. Regular expressions can be used to specify paths that are supposed to be excluded from the scan. Apart from that Acunetix can be configured to send custom cookies or headers. Moreover Acunetix supports HTML5 and AJAX crawling via its DeepScan technology [AcuDeep19]. Regarding authenticated scans, Acunetix supports HTTP-Authentication as well as NTLM-Authentication and Login Sequences [AcuLsr19].

Furthermore the black-box approach of the Acunetix WVS can be extended by white-box testing methods by setting up its AcuSensor technology [AcuSen19]. The AcuSensor functions are currently supported for three server-side scripting languages PHP, ASP.NET and Java. To activate it the tester first has to include a file containing AcuSensor instrumentation code into the web application project. Then he instructs the script interpreter to prepend the AcuSensor code to every server-side script. As a consequence AcuSensor is able to submit local information about the server state to the black-box scanner. Activating Acusensor implies among others the following benefits:

- Better coverage of vulnerabilities by detection of a larger range of vulnerabilities.
- Increased quality of test results as a result of decreased false positive and false negative rates.
- Addition of verification functionalities that clearly decide the existence of vulnerabilities.
- Collection of additional information for developer reports like the location of the vulnerability in source code.
- Back-end crawling which provides the scanner with a list of local files that can be further examined to detect hidden parameters.

5.3 Arachni

Arachni is a free WVS published under a public source license. This means Arachni is free to use for activities that do not involve commercialization [ArachLic19]. The tool can be deployed in 4 different variations [Arach19].

- Quick scans can be performed via the command line interface
- A Ruby library supports customized scans
- A Web user interface is capable of multi-User, multi-Scan and multi-Dispatcher execution
- Remote agents can be used to deploy distributed systems

Arachni supports HTML5, AJAX, DOM and advanced JavaScript crawling. Just as Acunetix WVS Arachni adapts itself automatically to the technologies which are used by the web application under test. The tool also supports specialized tests for mobile applications.

5.4 w3af

w3af stands for web application attack and audit framework. It is an open-source WVS which is designed to be easy to use and extend. It comprises a graphical user interface and a console user interface. Additionally a Web user interface is available. w3af consists of several modules that contribute to the vulnerability scan via activated plugins which are coordinated by a central coordination component. The different plugins and their corresponding tasks are summarized in the following listing:

- **Crawling:** Detection of as many valid application URLs that are delivered to the audit plugins as possible
- **Audit:** Execution of black-box tests for distinct vulnerabilities
- **Authentication:** Establishment and maintenance of a user session
- **Bruteforce:** Iterative testing for weak credentials

- **Grep:** Text-based analysis of HTTP-responses
- **Infrastructure:** Detection of server-side technologies
- **Mangle:** Dynamic modification of exchanged requests
- **Output:** Creation of scan reports

w3af scan profiles can be used to store a plugin configuration for later use. Interestingly w3af provides plugins that enable users to exploit vulnerabilities that have been found. For instance, a tester can exploit CE vulnerabilities using the os_commanding plugin.

6 Comparison of Web Vulnerability Scanners

Table 6.1 shows the total number of vulnerability occurrences in the VSN applications and in the tests performed afterwards. The additional tests were conducted with each vulnerability being implemented by a designated single page web application which did not require advanced crawling techniques in most cases. Thus these applications were rather examining the scanners audit capabilities. Furthermore the additional tests row was the only series of tests where we counted each test case instead of counting the vulnerabilities solely. This was done to consider the test cases which were supposed to produce false positives in the results.

Platform	**SQLi**	**XSS**	**CSRF**	**CI**	**CE**	**DT**	**FD**
VSN$_{GET}$	12	10	5	0	2	5	1
VSN$_{POST}$	12	11	10	0	1	6	1
VSN$_{Filter}$	2	8	15	0	1	2	1
Additional Tests	1	6	5	1	1	2	2
Total	27	35	35	1	5	15	5

Table 6.1: Overview of the total occurrences of vulnerabilities in the tested platforms.

6.1 Analysis of VSNg Test Results

Acunetix WVS and w3af detected most of our bSQLi vulnerabilities. To be more precise both scanners detected the same bSQLi vulnerabilities situated in login and registration forms as well as ones located in additional internal application functions. Nonetheless two vulnerabilities were missed. The first vulnerability received two arguments of whom one parameter was reported by the scanners correctly. By examining the source code we determined that the missed parameter was a password parameter that was restricted in length which might have prevented both scanners to detect the vulnerability. The other vulnerability was not restricted at all. Regarding this weakness

VSN$_{GET}$	**SQLi**	**XSS**	**CSRF**	**CI**	**CE**	**DT**	**FD**
Reference	12	10	5	0	2	5	1
Acunetix	10	0	3	0	2	1	0
Arachni	5	1	1	0	2	0	0
w3af	10	0	0	0	1	0	0

Table 6.2: Numerical results of the vulnerabilities which the scanners detected in VSNg.

one can consequently assume that the scanner was not able to detect the weakness due to insufficient crawling or evaluation. Additional analysis has shown that the request that was required was initiated by an AJAX request which was triggered by an JavaScript event. Consequently we consider the first explanation more likely.

Regarding XSS vulnerabilities the scanners did not perform well. Merely Arachni detected a single DOM-XSS on the search page. Because such a coverage rate was not expected we contacted the Acunetix support to aid us determining potential causes for the low detection rate regarding XSS vulnerabilities. After a manual examination of the web application under test they indicated that the low detection rate seems to be related to a bug in their DeepScan Technology. To further examine the behavior of the scanners we decided to conduct additional tests which we will discuss in 6.4. Nonetheless we also have to consider that the included weaknesses were pXSS vulnerabilities. Parvez et al. pointed out that this kind of XSS is especially hard to detect for automated scanners [PZK15]. Additionally there was an hidden rXSS parameter on the search page. We expected Acunetix WVS to be able to detect this vulnerability by an observation of the source file using gray-box testing techniques that are provided by AcuSensor. However the scanner missed that input just as the other scanners.

Accurate observation and comparison of the CSRF vulnerabilities indicates that Acunetix as well as Arachni restricted their search for CSRF token to Hypertext Markup Language (HTML) forms. However many components of VSN relied on JavaScript events and AJAX requests instead. We assume their low detection rates are a consequence of that. w3af did not detect any CSRF vulnerabilities at all.

CE vulnerabilities were detected by all scanners. However w3af only detected one parameter that was vulnerable to CE. It missed another parameter that was equally capable to inject shell code. Manual analysis shows that both parameters were first concatenated and then included into an OS command scheme. We assume that the result is related to the order in which the parameters are injected into the OS command.

In our first test row only Acunetix WVS was able to detect one DT vulnerability that could be used to partially disclose the input of a file. The other DT vulnerabilities we included into the application were especially hard to detect because they rather present vulnerabilities for the applications underlying logic. For instance, one vulnerability allows an adversary to craft friend-request messages between arbitrary users. We suppose the scanners did not detect the remaining DT vulnerabilities as a consequence of insufficient detection methods. According to the vulnerability report of Acunetix the scanner detected the DT vulnerability by testing techniques provided by its AcuSensor Technology. Compared to the other vulnerabilities that rather triggered state changes this vulnerability could be exploited to disclose the content of an arbitrary file partially. We assume that the additional information represented by the file content allowed the detection of this vulnerability by Acunetix. Potentially, Acusensor created a known file in the local filesystem and instructed the web application to retrieve the content of this file to detect the vulnerability.

Generally FD weaknesses are very difficult to detect for black-box scanners because they have few means to gather information about the Web application' state. Thus detecting a change in file system is hard. In this case the file deletion vulnerability was also restricted to the folder which contained the images of the users. We assume that this restriction was crucial to prevent Acunetix from detecting this vulnerability in VSNg because on the other side it detected the related DT weakness previously.

6.2 Analysis of VSNp Test Results

VSN$_{POST}$	SQLi	XSS	CSRF	CI	CE	DT	FD
Reference	12	11	10	0	1	6	1
Acunetix	7	0	3	0	0	1	0
Arachni	5	1	1	0	0	0	0
w3af	8	0	0	0	0	0	0

Table 6.3: Numerical results of the vulnerabilities which the scanners detected in VSNp.

Table 6.3 shows the results of our second scan. The bSQLi injection vulnerabilities which we included in VSNp differed from the vulnerabilities injected in VSNg in two points. Firstly they relied on POST-requests. Secondly they were more difficult to detect for the scanner's crawling component. The crawling challenges which the scanners had to cope with are similar to the ones which might have prevented the detection of XSS vulnerabilities in VSNg. Consequently we suppose that the latter argument was primarily responsible for the decreased coverage of all three scanners. The bSQLi that no scanner except for w3af detected was executed by a mouse click on a AJAX component and might be related to nondeterministic fuzzing.

The coverage results for the XSS, CSRF, DT did not differ significantly from the previous scans. Although the method was changed to POST we did not measure another result regarding XSS. We suppose that this observation is another indication that the scanner's crawling components had problems during the tests. The results regarding CSRF were similar to the previous scan too. Although we added additional CSRF vulnerabilities, the scanners did not detect more vulnerable endpoints. Similarly as in the VSNg we suppose that this was due to the scanners tendency to restrict their search to HTML forms. Regarding the DT and FD vulnerabilities we rather assume that the scanner's audit methods were insufficient due to a lack of proper heuristics.

We also increased the difficulty of the CE vulnerability by limiting the set of allowed characters. As a result, none of the scanners was still able to detect the CE weakness although it was still possible to include malicious code.

6.3 Analysis of VSNf Test Results

VSN$_{Filter}$	SQLi	XSS	CSRF	CI	CE	DT	FD
Reference	2	7	15	0	1	2	1
Acunetix	2	0	4	0	0	0	0
Arachni	2	1	1	0	0	0	0
w3af	2	1	0	0	0	0	0

Table 6.4: Numerical results of the vulnerabilities which the scanners detected in VSNf.

The vulnerabilities included in VSNf are designed to be very difficult to detect by WVS. Consequently the low detection rates were expected. Performing attacks on the XSS vulnerabilities would have afforded to bypass filtering mechanisms that filter many tags and escape other JavaScript meta-characters. Nonetheless most of the filtering mechanisms had still a weakness whose detection would have afforded an accurate manual audit. At this point we can not make any conclusions about the performance of the scanners when confronted with these filter mechanisms. This was due to our previous findings which suggests that the scanners were not able to crawl to these vulnerabilities. The vulnerable CSRF tokens were either predictable or fixed. During this test w3af detected an XSS that was undiscovered by the other scanners. We assume that the vulnerability had been missed by them because it is only triggered by XSS strings in a very specific length.

6.4 Analysis of Additional Tests

VSN$_{Filter}$	**SQLi**	**XSS**	**CSRF**	**CI**	**CE**	**DT**	**FD**
Reference	1	6	5	1	1	2	2
Acunetix	1	3	1	0	1	2	2
Arachni	1	3	1	1	0	2	0
w3af	1	0	2	1	1	0	0

Table 6.5: Numerical results of the vulnerabilities which were detected in additional tests.

Acunetix was the only scanner that successfully discovered both FD vulnerabilities by using its AcuSensor technology. This is no surprise because black-box scanners only have few possibilities to detect that an action deleted a file on the server's local file system. Another point worth mentioning are the side-effects of w3af's behavior while testing for the FD vulnerability. The scanner accidentally instructed the vulnerable file to delete itself, thus inducing permanent changes to the web application. As a consequence the web application could not process requests anymore which made further scanning pointless. Huang et al. already addressed the issue of side effects earlier [HTLK04]. They introduced a side-effects aware testing methodology and examined the influence of the side-effect awareness of scanner's on the coverage. Arachni was able to detect the DT weaknesses due to its heuristic.

Moreover this test series yielded further hints which seem to confirm our previous thesis regarding the scanner's approach to detect CSRF weaknesses. We integrated every CSRF vulnerability into a HTML form. As a consequence the alert rates of Acunetix and Arachni increased. However our tests contained several false negative locations which were containing guessable tokens and one false positive test case that included a visible but valid csrf token. Arachni and Acunetix were not able to correctly classify any of these false negative forms. Generally, our findings suggest that the scanners reported a csrf vulnerability if and only if a form did not contain a hidden form field. w3af did not detect a CSRF vulnerability in any platform. Thus we assume that it was not actually able to conclude that the two correctly classified vulnerabilities were false positives, instead it may have ignored CSRF vulnerabilities just as in the tests before.

One of the main reasons for additional tests was to check whether the xss vulnerabilities were missed due to insufficient crawling capabilities. In our final tests the scanners detected almost half of the vulnerabilities we isolated from VSNf. w3af did not detect any XSS weakness which might indicate that it implemented insufficient filter evasion mechanisms. Arachni and Acunetix did not detect weaknesses with that were very restrictive. For instance, to overcome one filter the scanners needed to inject a payload containing a specific tag class. Additionally both scanners had problems to classify code locations that submitted AJAX requests which were triggered by JavaScript events. Thus we conclude that the scanners missed the XSS vulnerabilities in previous tests because of limited crawling capabilities which conforms to the assumption of the Acunetix support.

We have no clue why Acunetix and Arachni did not find the CI respective CE vulnerabilities. We expected the scanners to report them because they were comparable to VSNg. Further tests would have been necessary to examine this behavior.

6.5 Evaluation

Low detection rates of vulnerabilities which required the scanners to correctly handle JavaScript events and AJAX indicate that the crawling capabilities of modern scanners need to be improved. This evaluation is consistent with previous literature [DCV10]. Pellegrino et al. proposed an interesting approach to improve the scan coverage. They introduced a technique that analyses the client-side JavaScript program dynamically [PTBR15]. Using the dynamic analysis they were able to monitor the execution of the JavaScript program and thus were able to detect events and dynamically created URL's. However in our case the scanners were able to extract a part of the URL's which were created at runtime. This implies that the scanners already implement approaches to crawl dynamically generated URLs

Additionally we observed that scanners can not perform a complete fuzzing and thus might miss some vulnerabilities that are not sufficiently protected by application filters. Since black-box web vulnerability scanners only test a limited set of parameter values for efficiency reasons, they can not provide evidence that a web application is sound. This is especially true in practice because the range of potential user inputs grows exponentially by length. Acunetix approaches to overcome this limitation by introducing gray-box scanning techniques which seems promising. Nonetheless the scanner missed several vulnerabilities that it could have detected under ideal conditions. Thus we also state that additional improvement of the integration of black-box and white-box techniques is necessary.

The WVSs were not able to detect vulnerabilities that afforded the tester to consider the web application's architecture because they could only be abused by sequential interaction of two user entities. Although they should be detected easily by a human penetration tester by the creation of another dummy account, in a manual audit. Thus Web application scanners need to be improved regarding vulnerabilities that require crawling that is capable to consider an application's design.

7 Outlook

Currently there are numerous ideas to generalize the concept to test web applications with WVS. Antunes et al. recently examined if recent ideas could be transferred to REST or SOAP based web services [AV17]. Additionally the authors proposed three distinct designs for web service vulnerability testing tools. Their general approaches were based on improved black-box testing, interface monitoring and anomaly detection at runtime. To show the potential of their ideas they conducted comparison tests with state-of-the-art web vulnerability scanners which confirmed that their adapted approaches are more efficient than existing automatic tools.

Another potential generalization point of web vulnerability scanners is to extend their target range by network vulnerabilities. Commercial web vulnerability scanners such as Acunetix WVS already have advanced network scanning functions implemented [AcuNet19]. With these functions they are searching for vulnerabilities in active sockets for various other protocols like the SSH, FTP, SOCKS or Telnet. El et al. recently benchmarked Nessus and Burp Suite in a network environment consisting of Supervisory Control and Data Acquisition (SCADA) devices and scientific instruments [EMS+17]. Burp Suite and Nessus are two state-of-the-art WVS that are also able to detect network vulnerabilities. Their research suggests that automatic scanners also show potential in this domain. Additionally they determined that each scanner has a distinct detection rate regarding different vulnerability classes [EMS+17]. This generalization approach seems especially important as exclusively testing for web vulnerabilities is not an holistic approach to test the security of a system. There are other attacks that affect web applications that are not exploited using non-web channels such as cross channel scripting (XCS) which was further examined by Bojinov et al. [BBB09].

Thirdly Antunes et al. already mentioned that web vulnerability testing should be more interconnected with the development process [AV12]. To prove their claim they refer to Microsoft's Security Development Lifecycle that complements their software development process. According to Microsoft, these additional activities, which are concerned with software security, reduce the number of vulnerabilities in its products [AV12]. Additionally they also reflect about the idea to devise compilers that automatically detect and correct security vulnerabilities in the source code of applications [AV12]. Indeed, WVS that automatically fix security issues would further reduce manual effort in web application development and thus decrease development and maintenance costs too.

8 Conclusion

In this thesis we examined and compared the capabilities and limits of three web application scanners: Acunetix, Arachni and w3af. For this purpose we developed a new web application and scanned it using the tools. We observed that Web vulnerability scanners can aid developers to establish and maintain the security of web applications. Correctly configured, they can assist the software development process by detecting and reporting a wide range of web vulnerabilities and can reduce the costs for web application testing by reducing the time necessary for manual audits.

Furthermore new approaches such as gray-box testing seem promising because they are able solve problems that are currently very intricate for pure black-box scanners such as the detection of server-side state changes. Additionally they provide techniques to verify their findings which saves time that would be needed for a manual examination otherwise.

We have seen that web vulnerability scanners reach their limits when they are instructed to search websites that rely heavily on dynamic technologies such as JavaScript and AJAX. Considering static websites which contain basic vulnerabilities like SQLi or rXSS web application scanners seem to be suited for a coarse grained search under constant monitoring of an security expert. We conclude that Web vulnerability scanners are a suitable tools for security professionals that are intending to test rather static large-scale web applications to reduce the present vulnerabilities in a quick first scan. However they do not replace a manual security audit.

Bibliography

[AAZ+17] A. Alzahrani, A. Alqazzaz, Y. Zhu, H. Fu, N. Almashfi. “Web application security tools analysis”. In: *2017 ieee 3rd international conference on big data security on cloud (bigdatasecurity), ieee international conference on high performance and smart computing (hpsc), and ieee international conference on intelligent data and security (ids)*. IEEE. 2017, pp. 237–242 (cit. on pp. 10, 11).

[Acu19] acunetix.com. *Acunetix - Website*. Online; accessed 3-May-2019. 2019. URL: `https://www.acunetix.com/` (cit. on p. 28).

[AcuCI19] Acunetix. *What is Code Injection | Acunetix*. Online; accessed 13-May-2019. 2019. URL: `https://www.acunetix.com/blog/articles/code-injection/` (cit. on p. 16).

[AcuDeep19] Acunetix. *HTML5 Security with Acunetix DeepScan Technology*. Online; accessed 14-May-2019. 2019. URL: `https://www.acunetix.com/vulnerability-scanner/crawling-html5-javascript-websites/` (cit. on p. 28).

[AcuFD19] acunetix.com. *Acunetix - Website - Arbitrary File Deletion*. Online; accessed 7-May-2019. 2019. URL: `https://www.acunetix.com/vulnerabilities/web/arbitrary-file-deletion//` (cit. on p. 16).

[AcuLsr19] acunetix.com. *VIDEO: Acunetix Login Sequence Recorder*. Online; accessed 14-May-2019. 2019. URL: `https://www.acunetix.com/blog/docs/acunetix-wvs-login-sequence-recorder/` (cit. on p. 28).

[AcuNet19] acunetix.com. *Key Features of the Acunetix Online Network Security Scanner*. Online; accessed 3-May-2019. 2019. URL: `https://www.acunetix.com/vulnerability-scanner/network-security-scanner/` (cit. on p. 37).

[Acurep19] Acunetix. *Types of reports | Acunetix*. Online; accessed 13-May-2019. 2019. URL: `https://www.acunetix.com/support/docs/types-reports/` (cit. on p. 21).

[AcuSen19] Acunetix. *Interactive Application Security Testing (IAST) with Acunetix AcuSensor*. Online; accessed 14-May-2019. 2019. URL: `https://www.acunetix.com/vulnerability-scanner/acusensor-technology/` (cit. on p. 28).

[AcuXSS19] acunetix.com. *Acunetix - Website - XSS*. Online; accessed 3-May-2019. 2019. URL: `https://www.acunetix.com/websitesecurity/cross-site-scripting/` (cit. on p. 17).

[Arach19] Arachni. *Home - Arachni - Web Application Security Scanner Framework*. Online; accessed 14-May-2019. 2019. URL: `https://www.arachni-scanner.com/` (cit. on p. 29).

[ArachLic19] Arachni. *License - Arachni - Web Application Security Scanner Framework*. Online; accessed 14-May-2019. 2019. URL: `https://www.arachni-scanner.com/license/` (cit. on p. 29).

[AV12] N. Antunes, M. Vieira. “Defending against web application vulnerabilities”. In: *Computer* 45.2 (2012), pp. 66–72 (cit. on pp. 11, 19, 22, 37).

[AV17] N. Antunes, M. Vieira. “Designing vulnerability testing tools for web services: approach, components, and tools”. In: *International Journal of Information Security* 16.4 (2017), pp. 435–457 (cit. on p. 37).

[BBB09] H. Bojinov, E. Bursztein, D. Boneh. “XCS: cross channel scripting and its impact on web applications”. In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 420–431 (cit. on p. 37).

[BGBK11] M. Balduzzi, C. T. Gimenez, D. Balzarotti, E. Kirda. “Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications.” In: *NDSS*. 2011 (cit. on p. 21).

[CERN19] CERN. *National Vulnerability Database - Website*. Online; accessed 11-May-2019. 2019. URL: https://home.cern/science/computing/birth-web (cit. on p. 7).

[CWE19] cwe.mitre.org. *CWE - Website*. Online; accessed 3-May-2019. 2019. URL: http://cwe.mitre.org/ (cit. on pp. 13, 14).

[DCKV12] A. Doupe, L. Cavedon, C. Kruegel, G. Vigna. “Enemy of the state: A state-aware black-box web vulnerability scanner”. In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 523–538 (cit. on pp. 23, 24).

[DCV10] A. Doupé, M. Cova, G. Vigna. “Why Johnny can’t pentest: An analysis of black-box web vulnerability scanners”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2010, pp. 111–131 (cit. on pp. 7, 19, 24, 35).

[EMS+17] M. El, E. McMahon, S. Samtani, M. Patton, H. Chen. “Benchmarking vulnerability scanners: An experiment on SCADA devices and scientific instruments”. In: *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*. IEEE. 2017, pp. 83–88 (cit. on p. 37).

[FO07] E. Fong, V. Okun. “Web application scanners: definitions and functions”. In: *2007 40th Annual Hawaii International Conference on System Sciences (HICSS’07)*. IEEE. 2007, 280b–280b (cit. on p. 10).

[FVM07] J. Fonseca, M. Vieira, H. Madeira. “Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks”. In: *13th Pacific Rim international symposium on dependable computing (PRDC 2007)*. IEEE. 2007, pp. 365–372 (cit. on pp. 11, 19).

[GG17] S. Gupta, B. B. Gupta. “Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art”. In: *International Journal of System Assurance Engineering and Management* 8.1 (2017), pp. 512–530 (cit. on pp. 16, 17).

[Gio09] S. Gioria. “Web application security”. In: *Clusif* (2009), pp. 1–20 (cit. on p. 23).

[HTLK04] Y.-W. Huang, C.-H. Tsai, D. Lee, S.-Y. Kuo. “Non-detrimental web application security scanning”. In: *15th International Symposium on Software Reliability Engineering*. IEEE. 2004, pp. 219–230 (cit. on p. 34).

[IG17] B. Idrissi, S. Guerouate. “Performance evaluation of web application security scanners for more effective defense”. In: *International Journal of Applied Engineering Research* (2017) (cit. on pp. 7, 23).

[JKC+06] N. Jovanovic, Kruegel, Christopher, Kirda, Engin. “Pixy: A static analysis tool for detecting web application vulnerabilities”. In: *2006 IEEE Symposium on Security and Privacy (S&P’06)*. IEEE. 2006 (cit. on p. 23).

[MNC16] I. Medeiros, N. Neves, M. Correia. “Detecting and removing web application vulnerabilities with static analysis and data mining”. In: *IEEE Transactions on Reliability* 65.1 (2016), pp. 54–69 (cit. on p. 23).

[NVD19] nvd.nist.gov. *National Vulnerability Database - Website*. Online; accessed 3-May-2019. 2019. URL: https://nvd.nist.gov/ (cit. on p. 14).

[OWASP19] owasp.org. *OWASP - Website*. Online; accessed 3-May-2019. 2019. URL: https://www.owasp.org/ (cit. on pp. 13–15, 17, 28).

[OWASPwvs19] owasp.org. *Category:Vulnerability Scanning Tools*. Online; accessed 9-May-2019. Apr. 26, 2019. URL: https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools (cit. on p. 22).

[PTBR15] G. Pellegrino, C. Tschürtz, E. Bodden, C. Rossow. “jäk: Using dynamic analysis to crawl and test modern web applications”. In: *International Symposium on Recent Advances in Intrusion Detection*. Springer. 2015, pp. 295–316 (cit. on pp. 21, 35).

[PZK15] M. Parvez, P. Zavarsky, N. Khoury. “Analysis of effectiveness of black-box web application scanners in detection of stored SQL injection and stored XSS vulnerabilities”. In: *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE. 2015, pp. 186–191 (cit. on p. 32).

[RM88] J. W. Ratcliff, D. E. Metzener. “Pattern-matching-the gestalt approach”. In: *Dr Dobbs Journal* 13.7 (1988), p. 46 (cit. on p. 21).

[SecMa16] S. Chen. *Price and Feature Comparison of Web Application Scanners*. Online; accessed 13-May-2019. 2016. URL: http://www.sectoolmarket.com/price-and-feature-comparison-of-web-application-scanners-unified-list.html (cit. on p. 22).

[Sym19] Symantec. *ISTR Internet Security Threat Report Volume 24 | February 2019*. Online; accessed 12-May-2019. 2019. URL: https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf (cit. on p. 11).

[SZM09] H. Shahriar, Zulkernine, Mohammad. “Automatic testing of program security vulnerabilities”. In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. Vol. 2. IEEE. 2009, pp. 550–555 (cit. on p. 22).

[VMN15] A. Van Deursen, A. Mesbah, A. Nederlof. “Crawl-based analysis of web applications: Prospects and challenges”. In: *Science of computer programming* 97 (2015), pp. 173–180 (cit. on p. 7).

[Vuln19] nvd.nist.gov. *National Vulnerability Database - NVD, Vulnerabilities*. Online; accessed 12-May-2019. 2019. URL: https://nvd.nist.gov/vuln (cit. on p. 13).

[wp18] secure.wphackedhelp.com. *WordPress Arbitrary File Deletion Vulnerability Exploit*. Online; accessed 7-May-2019. Sept. 20, 2018. URL: https://secure.wphackedhelp.com/blog/fix-wordpress-arbitrary-file-deletion-vulnerability-exploit/ (cit. on p. 16).

[ZD12] J. Zhou, Y. Ding. “An analysis of urls generated from javascript code”. In: *2012 IEEE/ACIS 11th International Conference on Computer and Information Science*. IEEE. 2012, pp. 688–693 (cit. on p. 21).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature