POLITECNICO DI TORINO

Master Degree in Computer Engineering

Graduate Dissertation

Securing Vulnerable Web Applications

Providing a structured procedure and .Net Framework implementation examples

Supervisor
Prof. Antonio Lioy

Ema Srdoc

Academic year 2017-2018

Summary

This work tackles the problem of analysing a web application to detect security vulnerabilities and mitigate them properly with respect to context and requirements.

During the course of the stage attended throughout 4 months in KPMG S.p.A., the candidate was confronted with such task and, as a result of the experience gained in the process and the difficulties encountered, a procedure has been devised, with a structure designed based on what the candidate considers to be an optimised and less error prone approach when securing web applications.

The procedure provides a first analytic phase in which the application is examined based on purpose, functionality and technical structure. This section guides through the detection of the most common vulnerabilities and provides the better part of the information necessary to properly evaluate their severity. This includes the description of the most common mistakes made by developers along with the most relevant exploitation methods and consequences. The aim of this section is to concentrate all the relevant theory aspects of web application security, therefore facilitating to developers the task of properly individuating the vulnerabilities present in the code, and to provide an organised approach in order to optimise the process and guarantee that all the relevant aspect are properly examined.

For each of the vulnerabilities described in the analysis section, one or more implementation examples are proposed in the implementation section of the procedure. The code provided is extracted from the real case study performed during the stage and is therefore specific to the language and frameworks used by the web application, however the purpose of this section is not to provide code ready to be used on other applications but to show the logic behind the implementation of the theory aspects, discussed in the analysis section, and expose the considerations that where made upon selecting the proper approach. Once this concepts are clear the code can easily be translated to apply to other languages and frameworks. In addition this section also aims at minimising the time spent on discriminating the correctness of the different approaches that are proposed online. The implementations proposed in this document are the result of the discrimination performed by the candidate during her experience and have been tested to provide the expected mitigation.

The last section of this work provides guidelines and best practices for the correct maintenance of the secure state of the application throughout it's lifetime.

Contents

Chapter 1

Introduction

This thesis aims at providing the structure of a general procedure to guide the process of securing a web application against the most common risks. The procedure will be subdivided in 3 main phases:

1. Analysis
2. Implementation
3. Review and Maintenance

Such procedure is devised keeping in consideration the current state of the art of web security techniques and the most commonly exploited vulnerabilities.

Alongside the presentation of the general procedure a real case study will be provided, to help understand how context, means and purposes can influence the outcome of the whole process, together with examples of actual vulnerabilities and implementations.

The web application under exam was provided by the company KPMG S.p.A. and is developed under the .NET Framework. Therefore this provided implementation examples concentrates exclusively on the Windows environment.

The arguments are organised as follows:

Introduction: this chapter describes the context for this work, what motivated the author and how and in which cases this document can be useful to the reader.

Analysis of the web application: discusses the assessment process, dividing it in three main phases and for each describes the steps to be taken both from conceptual and practical point of view.

- **Phase 1** : the Black Box analysis, describes the first approach for understanding and testing the application through the provided interactions.
- **Phase 2** : the White Box analysis, takes a more in depth look at the application by examining the source code and configurations. In addition, the most common vulnerabilities, their exploitation and consequences will be explained, in order to allow a better evaluation of the actual risks.
- **Phase 3** : support tools and environment analysis, evaluates the security of the server, support software, libraries, etc. used by the web application, to identify known vulnerabilities, outdated versions and insecure configurations and connections.

Implementation of risk mitigation actions: explains how to secure the application against the evaluated risks, providing best practices and practical implementation examples from the real case study.

Review and Maintenance: describes the necessary actions that need to be performed after the securing process is terminated and discusses the policies that must be devised and implemented in order to guarantee the secure state of the application throughout it's lifetime.

1.1 Present state

The importance of having secure software is steadily increasing, as insecure software is undermining our financial, healthcare, defense, energy, and other critical infrastructure. As of today the situation is alarming; an increasing number of individuals and companies are turning towards the significant profits that derive from the exploitation of such insecure software.

Attackers have at their disposal a wide pool of systems with weak security, especially in developing countries, that offer an ideal testing ground; more and more people are attracted by it's lucrative opportunities, and even those with weak hacking skills can easily find online tools and script for their purposes, often becoming the vector for the real crackers who can thus remain well hidden.

Furthermore often minor cyber-crimes go unpunished as it is not feasible to perform an investigation for every ransomware victim, whereas major crimes are committed by specialists that can be very creative in hiding their tracks. In addition, thanks to cryptocurrencies, also illegal money transfers have become easy to hide.

On the other hand many companies still do not fully grasp the threat posed by this growing community of crackers and often adopt insufficient precautions and do not train their employees properly.

Even when properly adopted, defensive strategies are mainly reactionary, meaning that a system will be protected against well known threats but might be completely vulnerable to new unknown exploits. Whereas if companies were willing to invest more in the security of their systems, new predictive approaches could be implemented.

Furthermore increasingly complex software is developed in haste by sacrificing quality and security and most of the developers have no training in secure coding. It has also been observed that the explosion in the use of internet has resulted less competent system administrators, due to the tendency of pressing non-technical people into service as system administrators [1].

Another great issue nowadays is the persistence of old "legacy" software.

> "In computing, a legacy system is an old method, technology, computer system, or application program, "of, relating to, or being a previous or outdated computer system." Often referencing a system as "legacy" means that it paved the way for the standards that would follow it. This can also imply that the system is out of date or in need of replacement." [2]

When this kind of software remains active and running it can be a critical point for the security of the company.

The candidate, even in her limited experience, was faced with the fact that the majority of companies are constantly struggling with the pressure of keeping up with market demands, competition and the fast pace of technological evolution. This leads to the focus being concentrated in making the development phase as fast as possible in order to meet the demands, neglecting among many things also the security aspects or doing the bare minimum to be compliant with regulations, hence effectively protecting themselves against inspection, but very poorly against real security threats.

Incidentally, it was interesting to witness the effect of the new General Data Protection Regulation(GDPR) (EU) 2016/679, applicable in Italy since the 25th of May 2018. This regulation, that is better known for it's implications in privacy protection, in truth also focuses on the security of the data processing activities, clearly stating in Article 32 (1):

> "Taking into account the state of the art, the costs of implementation and the nature, scope, context and purposes of processing as well as the risk of varying likelihood and severity for the rights and freedoms of natural persons, the controller and the processor shall implement appropriate technical and organisational measures to ensure a level of security appropriate to the risk, . . . "

Faced with these new security requirements, and it's substantial fines, many companies have been forced to acknowledge their security gaps, or rather their new compliance gaps, and have decided to upgrade their systems accordingly. Those companies, lacking the technical expertise, had to turn to consulting firms or external experts in order to achieve compliance, and that is were the candidate had the chance to study this phenomenon. It emerged that many companies were found extremely lacking to even the most basic security requirements.

Ultimately I believe that progress is being made in raising security awareness in the IT community but these efforts are yet to reach the financial administration, which still needs to understand that underestimating or disregarding the security issues not only exposes customers, and employees alike, to the consequences of cyber attacks but also exposes the company itself to the disastrous impact a breach could have, such as, but not limiting to, defacement, intellectual property loss, financial damage and in addition, if found negligent, punitive fines.

In fact the 2018 Clusit Report on ICT security has highlighted that the global costs generated by cyber criminals has increased from 100 billion dollars in 2011 to over 500 billion dollars in 2017, of which 180 billion affected private individuals. It also estimates that by 2019 the cost could reach 2 thousand billion if appropriate countermeasures are not adopted.

1.2 OWASP

To contrast this trend and raise awareness a new security oriented open-source community has been growing in the last decades.

A great contribute to this community has been provided by the Open Web Application Security Project (OWASP) [3], a non-profit organization focused on improving the security of software. OWASP provides unbiased information, best practices and advocates open standards, in order to make security transparent to individuals and organizations alike. In addition the Top 10 project releases every couple of years a document containing the most critical security risks to web applications, estimated by gathering surveys from a large number of companies [4].

Throughout this dissertation the documentation and guidelines provided by OWASP are heavily referenced and implemented as they represent the current publicly available state of the art for security practices.

1.3 Objectives

The purpose of this work is to provide a structured approach to guide the process of securing a web application against the most common risks. This structure has been drafted initially during the securing of the KPMG web application and it has subsequently been refined. Examples drawn from the real case study will be presented, analysing the assumptions that led to insecure configurations and providing the current best practices and solutions.

As a consequence of the analysis performed on the current state of software security, providing an easy to follow structured procedure for discovering and patching the most common vulnerabilities exploited by crackers and in parallel providing an analysis and explanation of the risks involved might encourage more developers and companies to apply it on their own software. It would significantly facilitate the task of securing vulnerable web application and reduce the required man-hours involved. Furthermore it would be easily applicable also by developers untrained in writing secure code and therefore both companies and individuals might more easily decide to dedicate some time in securing their vulnerable software.

It must be highlighted, however, that the the optimum would be to develop software that is security-minded since the first stages of it's conceptual design, more often the priority is to release software as soon as possible, leaving security only as an afterthought. Furthermore in this work it will be assumed that the software that needs to be secured was not developed by the same person or team performing the securing, as it commonly is the case.

1.4 The application: SoD App

1.4.1 Purpose

The application analysed as real-case study is not an example of the previously mentioned "legacy" software, but is instead the product of developers untrained in writing secure code and the absence of security instructions and requirements in the design of the project.

It was regularly used by the company's employees and was available only in the intranet before the company decided to make the tool available also to it's customers. For this purpose however the application had to be exposed to the extranet and therefore it was mandatory to resolve the security issues.

The service provided by the application is the analysis of the segregation of duties in the **Enterprise Resource Planning** (ERP) systems of a customer, with the possibility to calibrate the software's variables to fit better the nature of the industry in which it operates and its particular ERP system. The application will be hereinafter referred to as SoD App.

The increasing reliance of business processes on the IT systems supporting their execution results in the necessity to define a system to properly monitor and manage **segregation of duties** (SoD) to avoid granting employees with excessive system authorizations, inadequate to their official duties, thus violating the **Principle of Least Privilege**. The principle states:

> "In information security, computer science, and other fields, the principle of least privilege (PoLP, also known as the principle of minimal privilege or the principle of least authority) requires that in a particular abstraction layer of a computing environment, every module (such as a process, a user, or a program, depending on the subject) must be able to access only the information and resources that are necessary for its legitimate purpose." [5]

This is a fundamental principle in defining security, as every additional point of access to a system constitutes a vulnerability. In the specific case of SoD, an attacker could exploit an improper management of authorizations through the use of **social engineering**, to gain access to information available to the targeted employee.

> "Social engineering, in the context of information security, refers to psychological manipulation of people into performing actions or divulging confidential information. A type of confidence trick for the purpose of information gathering, fraud, or system access, it differs from a traditional "con" in that it is often one of many steps in a more complex fraud scheme." [6]

Usually an attacker would select multiple targets and gather little pieces of information, seemingly innocuous, from the victims. All these bits of information, however, when put together in the hands of an expert in deception could allow the social engineer to impersonate someone internal to the company so accurately that even the most guarded person might be deceived.

On the contrary if all relevant information is well guarded and provided strictly to those who need to know it in order to perform their job and each employee is therefore charged with protecting a restricted set of information for which he is well aware of the exclusivity of knowledge, an attacker would have a much smaller and cautious pool of targets for each specific information, making his task significantly harder.

This is only one of the risks arising from the improper segregation of duties resulting from granting employees with excessive system authorizations, inadequate to their official duties. Planning for an appropriate division of responsibilities and reflecting it in the access privileges granted to users of IT systems becomes necessary for the proper, efficient and secure execution of the business processes.

In this perspective the web application offers a tool to quickly and efficiently scan the database generated by the ERP software used by the customer to retrieve roles and authorizations of each employee to allow easier identification of excessive roles and potential conflicts.

1.4.2 Functionalities

Home

The home view of the application [Fig. 1.4.2] offers a short introduction to the application and provides:

- Status of the account: running analyses and completed analyses.
- A list of the last analyses performed, along with a link to the report resource.
- A list of links to download helpful resources for the user.

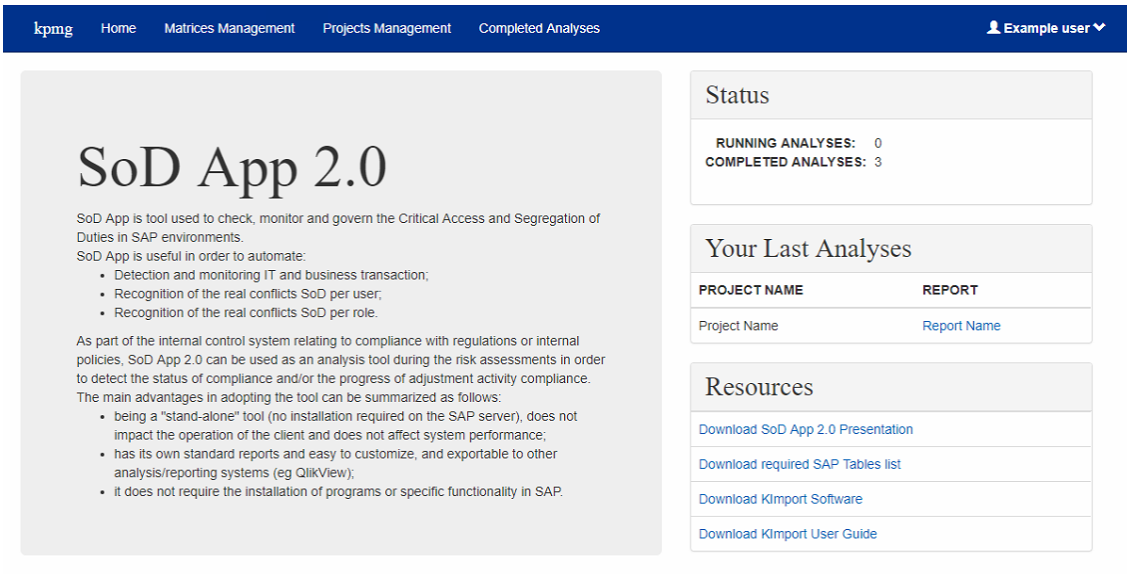

Figure 1.1. SoD App Home view.

Projects

In the Projects Management page [Fig. 1.4.2] it's possible to view a list of all created projects, modify them and create new projects [Fig. 1.4.2]. Each user has a maximum number of projects that can be created and a maximum number of analyses over all the projects. A single project usually is specific to a certain audit activity with respect to a specific client, therefore it might contain multiple databases and for each database multiple analyses with different parameters [Fig. 1.4.2].

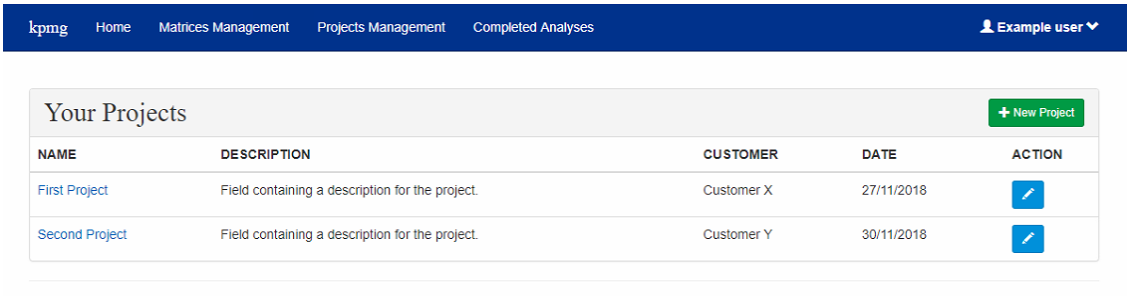

Figure 1.2. SoD App Projects Management view.

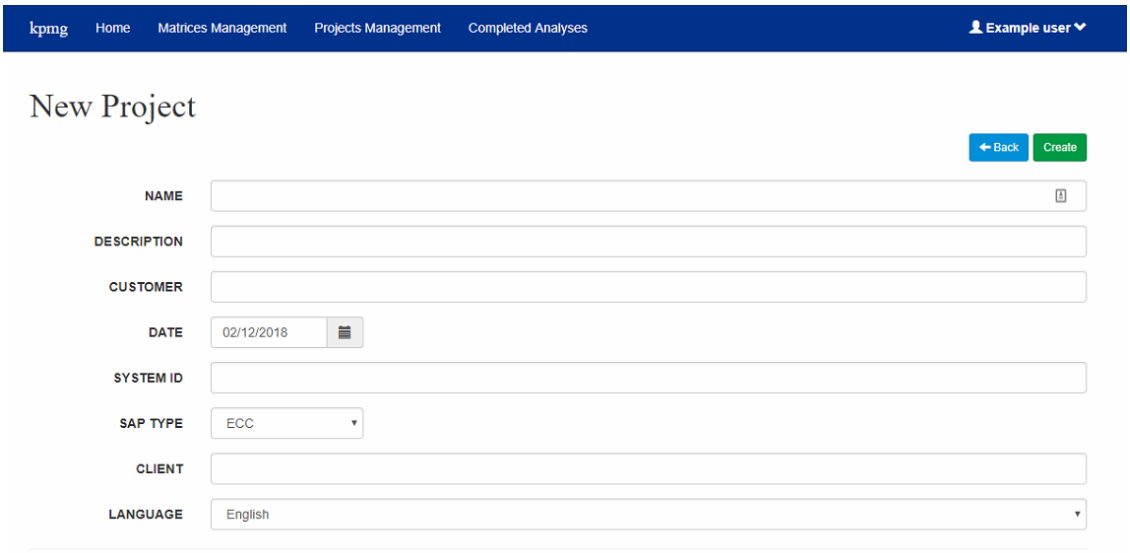

Figure 1.3. SoD App creation of a new project.

Analyses

Different types of analyses can be performed, and the output is returned in the format of a specified predefined excel matrix. Many parameters [Fig. 1.4.2] and filters are available to refine the analyses and when the setup operation is completed the application will delegate the analysis to an asynchronous thread on the server.

1.4.3 Structure

Framework

The application was developed under .NET Framework 4.7.2, using the C# language for the back-end, and was structured in a Model View Controller (MVC) pattern:

Model: defines the data structure, entities and relations.

View: defines the user interfaces through HTML.

Controller: the control methods and classes that define the functioning logic of the application.

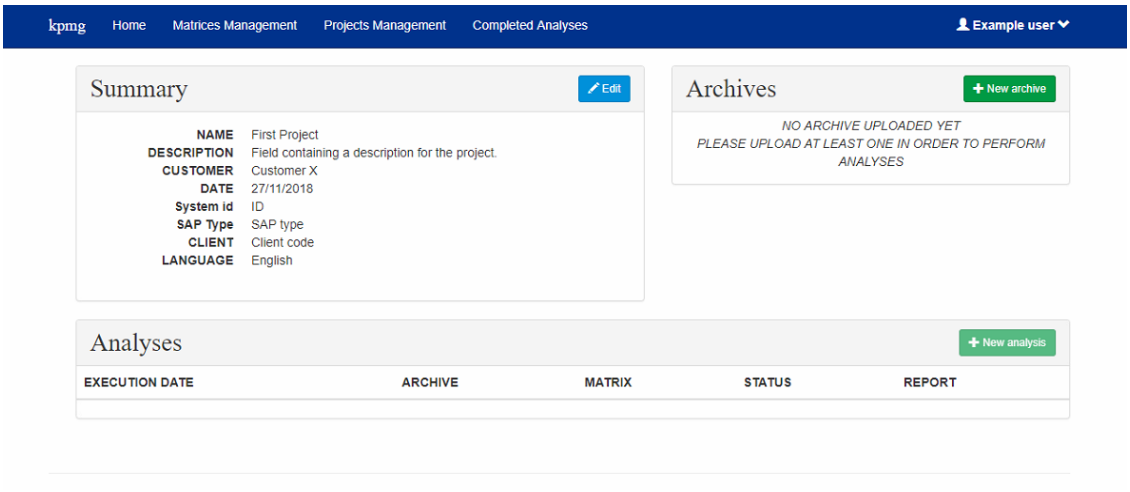

Figure 1.4. SoD App project view.

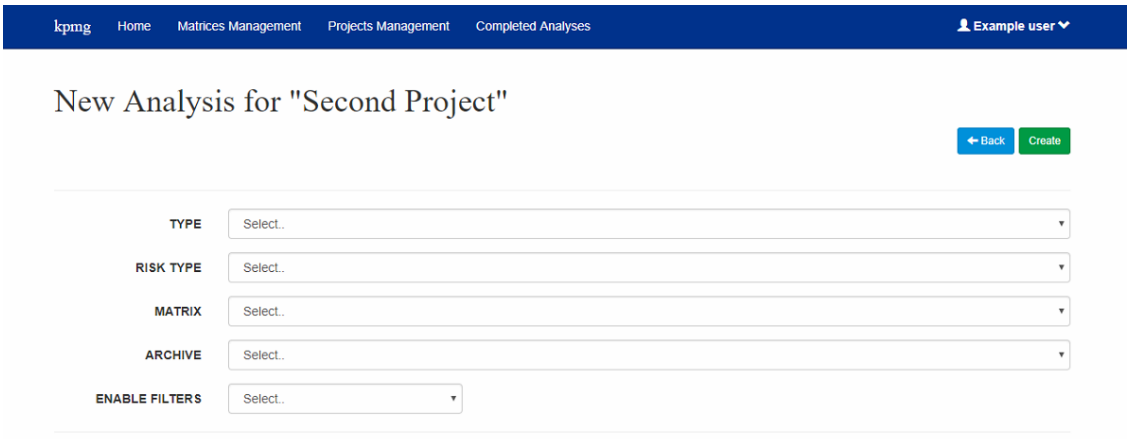

Figure 1.5. SoD App analysis parameters.

The use of a different structure pattern should not excessively influence the actual implementation of the solutions but only how the different elements of the application communicate and the data flows through the code.

The proposed implementation examples were developed for an application that targets .NET Framework 4.7.2, therefore some solutions might not work for older versions, however the examples are meant to help understanding the implementation logic and not to be an applicable solution for all cases. Once the mechanisms are clear it should be easy to reformulate the solution and adapt it to different contexts. The web server application used for SoD App was Internet Information Services (IIS) version 8.5.

Throughout the dissertation two important configuration files will be referenced, they are both named Web.config but are two different files, one used by IIS to manage it's own configuration and the other specifying web configurations used by the Web application, namely SoD App. In order to distinguish them they will be referred to as "IIS Web.config" and "App Web.config".

Database

The application uses both relational and non relational databases. Information related to users, projects and analyses is contained in MongoDB 3.4, a NoSQL database, whereas static data used by the application, such as analysis matrix schemes and analysis filters, are stored using Microsoft SQL Server 2016. The reasons behind these design choices are unknown.

Chapter 2

Analysis of the web application

In this chapter it will be described how to perform a 360-degree evaluation of the web application that must be secured. The procedure will assume that the tester has permission from the web application owner and complete access to it, it's source code, configurations and deployment devices. It will also be assumed that there is no familiarity with the application at the start of the analysis. This assumption is made on the basis that most of the time the person in charge of securing a software is not part of the team that wrote it, and might not be able to communicate with them. If this is not the case subsection 2.1.1 can be skipped.

2.1 Black Box analysis

The Black Box technique is a testing technique that examines the functionalities of a software without having access to it's internal structure and is usually employed for testing that focuses on what the software is supposed to do and not how it does it. In this context it will be useful to gain a global understanding of which actions are available to a user and how is interaction structured before diving into the code.

2.1.1 Understanding the application

In order to establish if the software behaves correctly, it is important to first understand it's purpose, what type of users it targets, the context of deployment and everything there is to know about the software. It is not possible to simply perform a list of standard actions to secure a web application, that is not the purpose of this work, but each software needs to be studied and understood in its own in order to effectively secure it.

The purpose

Understanding the purpose of the software helps defining how should the software behave. The following information should be gathered:

- How many types of user accounts are defined?
- Who are the users?
- How are the users authenticated?
- What actions can each type of user perform?
- What data is necessary to perform each action?
- How do users provide such data?

- What data should the software provide to the users?

Each software usually has at least two types of user accounts: common users and super users. Common users generally represent the end user the software is designed for, whereas super users could be the administrators and developers of the software. In reality there are usually more types of end user accounts, for example premium users, and there might be also more types of super user accounts with different degrees of privilege.

For each type it is necessary to clearly define what are their roles and purposes to be able to verify whether the software manages them correctly. It is also important to create typical profiles that describe who are the people using each type of account. For example it might be interesting to know if a user account is designed to be used only by people internal to the company, to a specific subset of employees, to associates of the company, etc. This information will become highly relevant when examining the actions available to each user and the authentication methods.

Depending on the type of account there might be different types of authentications: for example the website might require a simple password from a common user but digital certificates from super users. It is important to be aware of all of them in order to verify that they are properly managed and it is never possible to circumvent them.

Furthermore the software should be designed to offer to each account type a specific set of functionalities. These are the only actions it should be authorized to perform and it must be thoroughly verified that there is no possible sequence of actions that could lead to gaining access to any other area of the website it is not meant to access. Another reason for which it is important to really understand the purpose of the software is to judge when an account has been designed to have unnecessary privileges.

For example, considering an internet banking application, there might be a *User Administrator* account in charge of providing support to customers having problems with their own accounts. *User Administrator* type accounts should be able to perform all tasks necessary for their job, such as view and edit account configurations in order to help customers solve their problems, delete accounts, reset passwords, etc., but should not be able to perform actions such as creating or removing super user accounts, as these do not fall in the scope of "customers".

This example, however, would have been inconsequential without the initial explanation of the role of the *User Administrator* account. Roles are strictly tied to the context and purposes they are designed for, it is not possible to define a general rule, as in another smaller company the same operator in charge of customer users might have also been in charge of super users.

Hence it is not possible to correctly judge which privileges are strictly necessary unless each user has been correctly profiled and the context examined in detail.

Another important detail to consider is the data managed by the software. It is important to define which data the user is asked to provide in order to make two important checks:

- Is such data really necessary?
- Does the software always check that the provided data is in an acceptable format?

It should also be considered how and where this data is requested, but this will be explored in more detail in the next subsection "Mapping the Interactions".

Finally it should be clearly defined what data should the application provide to the different users, either as already available information or as result of a request.

2.1.2 Mapping the Interactions

Every time the software receives an input of any kind from an untrusted source this input should be handled with extreme care. The definition of **untrusted** source could be simplified as anything that could be potentially under the control of a malicious individual. Basically everything that

comes from the outside of the server itself should be assumed as untrusted, even if it comes from the intranet.

For the sake of this dissertation the term **input** refers to any information that is not directly elaborated by the software but must be provided externally. It could be the text inserted by the user in an input field, a selection from a drop down menu but also data not directly provided by the user, such as information inserted in the HTTP/HTTPS requests by the browser or scripts running on the client side of the application or state information not computed by the back-end of the web application. All this information could be easily tampered by a malicious agent and must not be trusted.

An important step is to thoroughly map all these input sources, completing them with a description of their purpose, what is expected by the software from that field and which account types should be able to access it. Common input sources are:

- input fields
- drop-down menus
- toggle buttons
- upload buttons
- the browser's address bar

These are all obvious inputs, but there are also less obvious inputs that are calculated by scripts running on the client side of the web application and incorporated in the HTTP/HTTPS request sent to the server. These inputs are also not to be trusted because even if the scripts themselves are integral part of the web application and therefore trustworthy, a skilled user could easily perform a number of actions to tamper with those scripts, like rewriting the script itself, disabling it's execution, editing manually the script's input directly in the HTTP/HTTPS request, etc.

Therefore anything implemented to run on the client side of the application should be only used for improving user experience but should in no case be considered safe.

This is a common mistake made by many website developers. Incidentally many wrong assumptions of these kind were found in SoD App. For example validation and sanitization of the inputs was made by a JavaScript script running on client side. The software would perform no additional check in the back-end C# classes. A non malicious user would be promptly notified if the format of the input was incorrect, however it was sufficient to edit the settings of the browser to disable JavaScript, effectively voiding any control. This could result in a SQL Injection attack or in runtime exceptions being triggered by the unexpected input. Both could pose a great security risk.

Validating an input means verifying if the input provided respects the required length, format, range, and allowable characters. For example if an input field is supposed to contain the name of a person, there shouldn't be any numbers in the string and it should have a realistic maximum length.

Sanitizing the input instead means that the input is modified in order to conform it with respect to software and security standards. It may include the neutralisation of dangerous characters from the input by means of removing, replacing, encoding, or escaping the characters.

These are two fundamental actions that must be performed by the server in order to prevent malformed data from persisting in the database and triggering malfunction of various downstream components. Input validation and sanitization should happen as early as possible in the data flow, preferably as soon as the data is received from the external party. In the next section, under "Input management" [2.2.2] it will be analysed in detail how they are implemented and what are the most common exploits against improper input handling.

Another error that was committed in SoD App was assuming that the user would only use buttons provided by the view in order to navigate between different pages of the website. This

assumption led to the belief that certain web pages were reachable by the user only through a controlled sequence of steps that were available only if the user had proved authorization to reach that particular page. Therefore controls had been done throughout the process of normally reaching the page, but where not done on the actual GET request for the specific page. This way a user could easily fool the authorisation system by requesting the interesting page directly typing it's URL in the browser's address bar.

In SoD App's specific case in order to reach the page containing the list of SoD analyses performed by user A on a specific database, and the relative links to download the results, the expected sequence of actions A should have performed was to login, select "Projects" button from the home page, then select the name of the project from a list, that would have provided the parameter for the URL query string to obtain the specific project and would have loaded it. However the control was made only during login, so if user B, after authenticating to the website with his own credentials then typed the URL to the project page using A's project identifier as parameter the server would not perform any authorization check and would provide the requested page.

The developer naively assumed that user B would not know the parameter for A's project, as it was available only after a sequence of actions that only A could make.

This kind of assumptions are extremely dangerous as they expose the website to *Web Parameter Tampering* attacks, based on the manipulation of parameters exchanged between client and server in order to modify application data. Usually, this information is stored in cookies, hidden form fields, or URL Query Strings, and is used to increase application functionality and control. In Sod App's specific case a malicious user can both try to guess randomly URLs trying to find interesting resources, or deduce the URL patterns to retrieve specific resources.

2.1.3 Testing the application with unexpected behaviours

Once the mapping has been performed, in order to establish if the application is able to respond correctly to malformed and tampered inputs, each of the mapped input sources must be tested with different types of inputs.

The correct behaviour for the back-end application would be to accept only the required format, implementing also some restrictions with regard to length and allowed characters, and refuse anything else.

In this part of the analysis the testing will concentrate only on the response of the application, however in section [2.2.2] it will be further verified how correctly is the validation performed, regardless of the success of the current tests. Here the main goal is to have an overview of the application, as it is easier to understand how the application works and what the developers expect from the user. It is extremely helpful to first familiarise in this way with the software before diving into the code. Code is often messy, poorly commented, full of leftover classes and methods that are not even used, and diving directly into it's examination without having a good idea of what it's supposed to do and what it actually does will most likely result in an inefficient and messy workflow, that is more prone to error.

Some examples of unexpected behaviours that can be tested are:

- Malformed input: testing characters and values that are not those clearly expected by the application.
- URL manipulation: trying to reach resources that should not be reachable by typing their URL directly in the browser's address bar.
- Unexpected values from drop-down menus: a common mistake made by developers is assume that since the drop-down presents a limited set of values then the input will necessarily belong to that set. In truth a malicious user could change the value of the parameter in many ways and the final request would contain an unexpected value for the given parameter. If the back-end does not correctly validate this value the attacker could exploit it for example by injecting code. Injection attacks will be explained in detail in section 2.2.2.

- Length of the input: test if there are maximum and minimum length rules and how does the software behave for extremely long or too short inputs.
- Wrong content upload: if the application expects a specific type of data, such as an image or a document, it should not be possible to upload a different type of file, even if the file extension has been removed or changed.
- Unexpected values from client-side scripts: as explained earlier since it is possible to disable the execution of scripts or tamper both with the script itself and the result, the back-end should validate also those inputs.
- Tampered parameters: modifying the parameters directly in the HTTP/HTTPS requests.

It is also necessary to verify that the application fails safely. If an unexpected input triggers an error this is a clear sign that the validation was not performed correctly, but how did the software react? In the worst case scenario the whole application crashes if the exception is not caught correctly. But even managed exceptions could result in security risks if not handled properly. This topic will be discussed in more detail in section 2.2.7.

In this step it is also possible to try to perform some known attacks, such as SQL Injection, Code Injection, XSS, etc. There are many examples online on how to perform such attacks and some will also be provided in section [2.2.2], together with a detailed description and explanation for these attacks.

It is however necessary to bear in mind that if the test attack fails it does not mean that the application is safe with respect to that type of attack. There are many reasons why an attack could fail, for example tester is inexperienced in penetration testing or the software is safe with respect to that specific input but might be exploited with a differently designed input. Some simple examples will be provided later in order to help understanding the attacks, however they are only meant to help in conveying the explained concepts and are not to be intended as sufficient test proofs against the relative attacks.

2.1.4 Testing with cyber attack tools

There are many interesting tools available online designed to find vulnerabilities in web applications. These tools should not be completely relied on, however they can help speeding the analysis project by scanning the application and providing a first set of vulnerabilities. Once the tester has evaluated the scan results he should proceed with a more in depth analysis of the application and performing all steps described in the previous sections. The advantage is that the tool will have performed part of the tester's initial work.

Analysis tools can be categorised based on 3 main parameters:

Approach: some tools are designed to use the Black Box approach and basically perform the steps described in this section, others use instead the White Box approach, analysing how each input flows through the code.

Focus: some are designed to perform general testing, others instead focus on testing the application with respect to a specific type of attack.

Cost: there are both open-source tools and proprietary tools.

Here a few interesting tools will be provided that follow the Black Box approach. For tools using the White Box approach see section 2.2.8.

OWASP ZAP

The OWASP Zed Attack Proxy (ZAP) [7] is one of the main OWASP projects and is a penetration testing tool for finding vulnerabilities in web applications that categorises as Black Box, general, open-source testing tool.

Advantages:

- free
- easy to use for beginners
- developers can configure it for automated security testing
- cross-platform
- it is possible to implement extensions

Main features:

Intercepting Proxy: it is possible to configure the browser to proxy trough ZAP so it can see all the requests and responses and edit them through it's interface.

Passive Scanners: when the passive scanner is activated ZAP performs some vulnerability evaluations by looking at the requests and responses. It is safe to use on any website as it does not perform any attack but it can nevertheless spot problems, such as exploitable content in headers and issues with HTTP modes.

Active Scanners: the active scanners perform a wide range of attacks and analyse the responses of the server in order to gauge vulnerabilities. These scanners should be used only on websites for which the tester has permission.

Spider: this feature implements a bot that navigates the website with the purpose to create an index of all the contents available. This can be useful to see what pages and resources the spider has been able to reach that should have been off-limits or if there are leftover contents from the development phase.

URL Brute-force: this tool uses OWASP DirBuster to find additional resources, even if there are no links to them, by brute-forcing the URLs.

> "In cryptography, a **brute-force** attack consists of an attacker submitting many passwords or passphrases with the hope of eventually guessing correctly. The attacker systematically checks all possible passwords and passphrases until the correct one is found. Alternatively, the attacker can attempt to guess the key which is typically created from the password using a key derivation function. This is known as an exhaustive key search." [8]

In this case the brute-forced value is the URL string.

Fuzzing: fuzzing is a term that refers to the testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. In this case the tool uses FuzzDB, an open dictionary of fault injection patterns, predictable resource locations, and regex for matching server responses. The provided pattern are able to cause issues like OS command injection, directory listings, directory traversals, source exposure, file upload bypass, authentication bypass, XSS, HTTP header CRLF injections, SQL injection, NoSQL injection and more, providing an interesting tool for testing common attacks. The fuzzing tool also uses OWASP JBroFuzz's library to test HTTP and HTTPS web protocol parameters. The HTTPS traffic can be processed thanks to dynamic SSL certificates that are generated by ZAP and can be added between the browser's trusted certificates.

Report generation: ZAP can produce reports describing the vulnerabilities found and providing links where to find additional information. This is a very useful tool as it gives more insight on the problems and how to solve them.

Port Scanner: scans the server to see what ports are open on the machine.

Parameter Analysis: by looking at requests and responses creates a list of all the parameters used by the application.

Anti CSRF token handling: if the application uses tokens that prevent Cross-Site Request Forgery attacks ZAP will use this tool to actively regenerate the tokens when performing active scanning or fuzzing.

Usage:

A penetration testing session could be structured in the following sequence of steps:

1. Configure the browser to proxy via ZAP.
2. Explore the application manually, ZAP will list all the resources explored for later use during scanning and fuzzing. This step could be skipped performing the exploration directly with the Spider tool but the result is more reliable if a good set of contents from which to perform the scan has been provided manually.
3. Run the Spider to find additional content that was not found manually.
4. Check the issues found by the passive scanner. The passive scan is run by default, so unless explicitly disabled the traffic generated by the previous steps has been analysed.
5. Run the active scanner.

This tool is also convenient for performing manual attacks. When the requests and responses are intercepted by the proxy feature their content is displayed and can be analysed and modified at will.

Burp Proxy

Burp Proxy is a tool very similar to OWASP ZAP but is a proprietary solution. The free version provides a very limited number of features, but the licensed version has more features than ZAP and is better maintained.

Firefox tools

Firefox offers an interesting set of add-ons that can be used to inspect requests and responses directly from the browser:

- Firefox LiveHTTPHeaders: allows to view HTTP headers of a page while browsing.
- Firefox Tamper Data: implements an interface to easily view and modify HTTP/HTTPS headers and post parameters.
- Firefox Web Developer Tools: native solution that implements various web developer tools to the browser.
- Firefox Firebug: a non native solution to to edit, debug, and monitor CSS, HTML, and JavaScript.

Acunetix Vulnerability Scanner

This tool performs automated web application security testing and as a result outputs a report containing the vulnerabilities detected, rated based on severity, along with the relative CVE (Common Vulnerabilities and Exposures) identification, information regarding impact, exploitability, associated malware and possible solutions. This tool was used on SoD App, however it detected only a minor set of vulnerabilities.

2.1.5 Analysis results on SoD App

Different types of analysis approaches where adopted, in the Black Box phase, returning different results. Before starting with a manual analysis a first evaluation was performed using the Acunetix Vulnerability Scanner. This tool reported the following vulnerabilities:

- Use of HTTP: resulting in exposed connections.
- Remote code execution: due to the use of an unpatched version of Windows.
- Information disclosure through application error message: application error massages and warnings that displayed to the user sensitive information about the logic and structure of the application.
- Use of RC4 cipher suites detected: due the existence of attacks against TLS when using RC4 encryption.
- SSL certificate public key shorter than 2048.
- Use of SSLv3 enabled: exposes to POODLE attacks.
- Various information disclosed in response headers.
- Various security headers missing.
- Various dangerous HTTP methods enabled.

Subsequently during the manual Black Box analysis the following additional vulnerabilities where found:

- Input validation performed only via javascript on client side.
- No input sanitisation.
- Weak file upload validation.
- Broken authentication on certain resources.
- Weak password policies.

All these vulnerabilities where successfully mitigated and the implementations can be found in the Implementation chapter.

2.2 White Box analysis

Now that the application has been properly analysed from an external point of view, the tester should have, in addition to the input mapping, a list of issues found, errors triggered and concerns. If the tester did not manage to find anything with the Black Box testing it is not a problem as this phase will cover the analysis of all the critical aspects of the application, even if no issue was detected. However if vulnerabilities were detected, the details of their manifestation will help understand the error committed and highlight specific issues that otherwise might have been ignored.

2.2.1 Application configuration

Before examining the code there are some general aspects of the application that must be considered.

HTTPS

One thing that can be noticed right away is whether the application uses the HTTP protocol, HTTPS or a combination of both. HTTP does not provide any security feature, requests and responses sent with this protocol are not encrypted and no integrity or authentication actions are performed. An attacker could easily intercept the packets, read the data and modify it without the client and server noticing the attack. Therefore passwords, usernames, cookies, files and anything else contained in the communication can be easily seen by anybody. It is clear that this is an unacceptable solution.

HTTP should never be used even with web pages that do not require authentication or accept any input from the client. Furthermore even if the client and server are not communicating sensitive data, due to the lack of integrity controls the server might still have it's responses to the client modified, posing a threat for the user that could receive malicious data as if it were coming from the legitimate server.

For the same reason mixed solutions should be avoided. Mixed solutions happen when some pages are protected by HTTPS, but some resources available in the page or other pages are loaded using HTTP. The unprotected resources and pages weaken the security of the website and might be used to crack it.

Some developers make the mistake of protecting with HTTPS only pages performing authentication or submitting sensitive data, but forget that they are making use of session tokens in all connections. This approach leaves the session token completely vulnerable when browsing pages with HTTP, effectively voiding any protection offered during authentication, as an attacker that intercepts the session token can then use it to authenticate as the user without having to provide the credentials.

HTTP Methods

HTTP offers a number of methods that can be used to perform actions on the web server. Many of theses methods are designed to aid developers in deploying and testing HTTP applications, however some could be exploited by crackers if the web server is misconfigured, as they allow an attacker to modify the files stored on the web server and, in some scenarios, steal the credentials of legitimate users.

If not used the following methods should be disabled:

PUT: allows a client to upload new files on the web server. An attacker can exploit it by uploading malicious files, or by simply using the victim's server as a file repository.

DELETE: allows a client to delete a file on the web server. An attacker can exploit it as a very simple and direct way to deface a website or to mount a DoS attack. Denial of Service (DoS) attacks will be described in section 2.2.2.

CONNECT: could allow a client to use the web server as a proxy.

TRACE: echoes back to the client whatever string has been sent to the server, and is used mainly for debugging purposes. This method, originally assumed harmless, can be used to mount an attack known as Cross Site Tracing.

Cross Site Tracing (XST) exploits the TRACE method to circumvent the protection of user cookies implemented by the HttpOnly flag. The method allows the client to see what is being received by the server therefore the attacker, by combining it with a XSS attack, would be able to see the cookies incorporated in the request when it is returned by the TRACE method, even if he is not able to access them directly using the document.cookie object [9], therefore invalidating the security offered by the HttpOnly flag, that will be described in detail in Session Management section[2.2.3].

In addition it was discovered in 2008 that many web application frameworks allowed well chosen or arbitrary HTTP methods to bypass an environment level access control check:

> "Many web environments allow verb-based authentication and access control (VBAAC). The rules for these security controls involve using HTTP verb (also called method), such as GET or POST, as part of the security decision. ... Unfortunately, almost all the implementations of this mechanism work in an unexpected and insecure way. Rather than denying methods not specified in the rule, they allow any method not listed. ... Some web platforms, including both Java EE and PHP, allow the use of arbitrary HTTP verbs. These requests execute similarly to GET requests, making it possible for attacker to use these verbs to bypass flawed VBAAC implementations. Even worse, the response is not stripped off as it is in HEAD request, so the attacker can see the unauthorized pages as if there were no protection."
>
> Arshan Dabirsiaghi, "Bypassing Web Authentication and Authorization with HTTP Verb Tampering", 28 May 2008 [10]

A consequence of what is described by the quoted text is that, depending on the framework, the method HEAD, for example, could be treated as a GET request, albeit one without any body in the response. If a security constraint was set on GET requests such that only authenticated users could access GET requests for a particular servlet or resource, it would be bypassed for the HEAD version. In addition some frameworks allowed arbitrary HTTP methods such as JEFF or CATS to be used without limitation. These were treated as if a GET method was issued, and were found not to be subject to method role based access control checks, again allowing unauthorized blind submission of privileged GET requests.

Therefore an additional check that should be implemented in code is the explicit verification of the method of the request. The web application should respond only if the method is the expected one. This would provide significant protection against misconfigurations or unexpected mechanisms.

If an application needs one or more of these methods, such as REST Web Services (which may require PUT or DELETE), it is important to check that their usage is properly limited to trusted users and safe conditions.

HTTP Response Headers

By default many web servers send with the responses information regarding the web application, such as version, framework used, etc. It is necessary to verify which information is sent and remove anything that gives too specific information of the website's architecture and software versions, or an attacker could use this information to exploit well known vulnerabilities or launch attacks refined to target the specific server configuration.

2.2.2 Input Management

In this section it will be explained what is the correct way of validating and sanitizing the most common types of inputs, and how should they be stored in the database, complete with an explanation of some of the most common attacks that make the proposed security measures mandatory. For each input the tester should thoroughly analyse how it flows through the code to verify that not only validation an serialization are performed correctly, but also that the values are stored and managed appropriately.

Managing Text Input

Throughout the website there will be many input fields and other mechanisms that will produce a parameter that is eventually handled as string. This kind of input is often subject to injection attacks of different kind, where the attacker tries to inject some code instead of the expected input.

> "Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization." [4]

Injection attacks are rated number 1 in the OWASP Top 10 list of most critical security risks to web applications. This kind of attacks are simple to perform, can be easily automated also through the use of the many tools available online and could have disastrous consequences for the attacked website.

Running trough the list of inputs mapped during the Black Box analysis, for each input source or web parameter that accepts text input it is necessary to find the method or methods that process it and examine whether validation and sanitization are correctly implemented as soon as the input is received.

As a general rule, for all input strings, it should be validated the minimum and maximum lengths, the character set and the logical validity of the value. Each validation function should additionally consider also if there are other requirements for the input that are assumed by the subsequently executed code. Characters should be managed using the **white-list** method, which means considering valid only those belonging to a predefined list of safe characters and refusing the whole input string if there are invalid ones. This method is opposed to the **black-list** method in which the input is considered valid if none of the characters that compose it belongs to a predefined list of dangerous characters. The second approach is strongly discouraged, however, as it is relies on the assumption that only the listed characters are dangerous, when in truth there are often loopholes that allow crackers to pass the validation with special character combinations.

Dangerous characters are those that have a special meaning in the language of the interpreter.

In general the safest choice is to allow only alphanumeric characters and deny all symbols and punctuation. Sometimes however it might be necessary to allow a larger character set than those strictly regarded as "safe", as a consequence an additional step of sanitization is necessary. After validation has been performed the sanitization function should either escape or encode the dangerous characters. It is strongly discouraged to craft the sanitization function oneself, as it is very easy to implement it incorrectly. There are specific libraries available that should be used instead and that have been designed by experts in the field and thoroughly tested.

In addition for some fields such as descriptions, comments, etc., the website might want to avoid restricting at all the character set. This choice removes an important level of protection, leaving the security of the input reliant only on the sanitization process. Therefore in such situations it is extremely important to analyse how will be the specific input used, in order to correctly sanitize it.

For ranged values such as integer, float, dates, etc., it should also be implemented a maximum and minimum value range check.

From the previous analysis phase it might be understood that some input fields are only meant to help the user and improve usability. These fields usually do not contain any information that is elaborated by the application, but are just used to be displayed in some part of the website, such fields are for example the Biography of the user, the description of an item, comments and reviews, etc. Usually they are not key attributes for database entries or parameters used for database access or parameters.

Following this assumption it must be verified, by analysing the code, that they actually conform to the expected usage. If the verification is positive these fields cannot be exploited to perform SQL Injections, but could instead be exploited to perform other types of attacks, depending on the use. If the fields are only retrieved to be displayed to the user, they are at risk of XSS attacks, therefore performing validation and sanitization against characters that are dangerous for HTML should be sufficient to avoid injection attacks.

Other attributes, on the contrary, have a functional role and are used by the web application to take decisions, they are more likely to be vulnerable to SQL Injection and if not properly validated result in Exceptions being raised. It is therefore necessary to verify that they belong to an acceptable range of values in order to avoid errors.

SQL injection attacks target the SQL database of the application and are executed by injecting, via web page input, strings that contain SQL commands and that are properly crafted in order to trick the interpreter into considering what should have been the parameter of a statement as actual part of the command. A successful SQL injection exploit could read sensitive data from the database, modify it, execute administrative operations (such as shutdown the DBMS) and in some cases issue commands to the operating system.

The breach could result in loss of **confidentiality** and **integrity**. Depending on the value or sensibility of the exposed/tampered data the company could face great financial losses, reputational damage and even fines. There could also be issues with **authentication**. If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password. Furthermore if **authorization** information is held in the database, it may be possible to change this information and permanently gain additional privileges.

To provide an example the following back-end code can be considered:

```
string querySQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

If the UserId is taken directly from input without proper validation and serialization the user might provide a malicious input such as "123 OR 1=1", which would result in the following statement being constructed:

```
SELECT * FROM Users WHERE UserId = 123 OR 1=1
```

The result of the WHERE clause would always be true, independently from the UserId value provided. This is just a simple example and the format of the SQL Injection input varies based on how is the back-end code structured. The defense against this type of attack should not rely on trying to formulate appropriately the statement to make it unexploitable, because an expert might still be able to find a fault, instead it should concentrate on neutralizing the input itself on different levels:

1. Validation: if possible the use of dangerous characters should be prohibited by using, for example, regular expressions.
2. Sanitization: for additional protection the input should also be encoded using appropriate encoding libraries.
3. Using **prepared statements**: this step offers the highest level of security. Prepared statements use a structure that forces the developer to first define all the SQL code, and then pass in each parameter to the query later. Since different objects are used to contain the query and the parameters the database can distinguish between code and data, regardless of what user input is supplied.

Each step offers a certain degree of security, that however is not fail-proof due to implementation errors and bugs in the libraries used.

Even if some of the security features implemented might seem to overlap in utility, implementing multiple levels of security guarantees that as long as all do not fail the input is safely processed. This approach is fundamental when implementing security in all aspects. There is never redundancy of security.

After it has been established that an input is used to construct queries, it is not important whether the result of the query is displayed directly to the user or used for internal processing, whether it contains data that seems relevant or not, etc. It should be always and without exception managed safely using prepared statements. A cracker might be able to exploit such opening to generate errors or other kinds of unexpected and unwelcome behaviours.

Prepared statements, however, are a concept that is present for relational databases. If the application uses NoSQL databases there is not an equivalent concept, as the queries are not constructed as strings. However non relational databases are also subject to injection attacks if the input is crafted properly. Given the following query string:

```
database.users.find({username: username, password: password});
```

If this is a MongoDB query string, an attack is possible if for example the parameters come from deserialised JSON objects:

```
{
    "username": {"$gt": ""},
    "password": {"$gt": ""}
}
```

In MongoDB, the field $gt has a special meaning, which is used as the "greater than" comparator. If the username and password fields are not validated the username and the password from the database will be compared to the empty string and as a result return a positive outcome.

Therefore to protect against NoSQL injections the developer should take care to properly sanitize the input against characters that have a special value for the used database.

Sensitive fields, such as those used for identification and authorisation, should not be taken from user input or client-side scripts but should be retrieved or defined by the server.

Managing File Uploads

Uploaded files represent a significant risk to applications. The first step in many attacks is to get some code to the system to be attacked. Then the attack only needs to find a way to get the code executed. Using a file upload helps the attacker accomplish the first step. The consequences of unrestricted file upload can vary:

- Complete system takeover: the file could be designed to execute a web-shell which can run commands. Based on the level of privilege the attacker is able to obtain the consequences could be disastrous. If the uploaded file can be accessed by entering a specific URL path, it could be especially dangerous because the file could be executed immediately after uploading.
- An overloaded file system or database: if the user is allowed to upload an unlimited number of files or the size of files uploaded is not restricted an attacker could exploit it to perform a Denial of Service attack.
- Forwarding attacks to back-end systems: if the web-server is not isolated properly form the rest of the company network, or if the attacker is able to exploit local vulnerabilities, the implanted file could be used to reach other machines in the network.
- Client-side attacks: the file name or content could be used to perform XSS attacks or, if downloaded by the user, to execute malicious commands on the user's machine. In addition the attacker might be able to put a phishing page into the website.
- Breach of privacy: the code executed by the malicious file could browse system files and local resources and gain access to sensitive information regarding users.
- Defacement and other risks: the file storage server might be abused to host troublesome files including malwares, illegal software, or adult contents. Uploaded files might also contain malwares' command and control data, violence and harassment messages, or steganographic data that can be used by criminal organisations. These contents would not only cause damage to the user but also to the company's reputation.

"The **Denial of Service (DoS)** attack is focused on making a resource (site, application, server) unavailable for the purpose it was designed. There are many ways

> to make a service unavailable for legitimate users by manipulating network packets, programming, logical, or resources handling vulnerabilities, among others. If a service receives a very large number of requests, it may cease to be available to legitimate users. In the same way, a service may stop if a programming vulnerability is exploited, or the way the service handles resources it uses."[11]

It is virtually impossible to completely protect against a DoS attack, as it is often not easy to distinguish between legitimate traffic and malicious traffic, however there are ways to significantly mitigate the consequences. It is straightforward that if a service relies on inadequate resources and has trouble sustaining even the traffic generated by legitimate users, an attacker would easily generate enough traffic to exhaust the website's capacity, that as a consequence would not be able to handle new requests. Therefore a first step is to redesign the system architecture in order to be able to handle amounts of traffic sufficiently larger than the maximum entity of legitimate traffic that is daily registered. Once such measure has been implemented it will be possible to set up a monitoring systems that could detect if the server is under DoS attack and block the suspected traffic.

This measure however can offer little protection if there are bugs or vulnerable implementations in the code. An example of vulnerable implementation, as mentioned earlier, is allowing the upload of files without restricting the size of the file or the quantity of data a user can store on the server. In this situation an attacker could try to load an extremely large file or as many smaller files as necessary to saturate the database capacity. Huge files once loaded could also be exploited to saturate the RAM of the server if the application tries to load the whole file.

Some other examples are:

- Allocating a number of objects based on user input: if users can supply, directly or indirectly, a value that will be used to specify how many of an object to instantiate on the application server, and if the server does not enforce a hard upper limit on that value, it is possible to cause the environment to run out of available memory. This type of attack would also pass unnoticed under any monitoring system as it would require only one seemingly legitimate request to effectively fill the server's available memory. An example of vulnerable code:

```
int elementsNumber= Int32.Parse(userInput); //no check on elementsNumber value
ComplexObject[] objects= new ComplexObject[elementsNumber];
```

  The proposed code should be modified to implement a range check for the elementsNumber value.

- Using user input as loop counter: if the user can directly or indirectly assign a value that will be used as a counter in a loop function an attacker could create an extremely long loop, locking the used memory resources for the whole duration. An example of vulnerable code:

```
string[] values = GetValuesProvidedByUser(Request); //no check to verify if
    number of values is acceptable
for ( int i=0; i<values.length; i++) {
  // lots of logic to process the request
}
```

  The proposed code should be modified to implement a range check for the length of the values array.

- Improper release of resources: if an error occurs in the application that prevents the release of an in-use resource, it can become unavailable for further use, occupying the memory indefinitely and resulting in memory leakage.

```
OleDbConnection connection = new OleDbConnection(connectionString);
connection.Open();
OleDbCommand cmd = new OleDbCommand(query, connection);
cmd.Parameters.Add(new OleDbParameter("someParameter", parameterValue));
OleDbDataReader reader;
```

```
reader = cmd.ExecuteReader();

//retrieving column headers
List<Field> head = new List<Field>();
for(int i = 0; i<reader.FieldCount; i++)
{
  Field f = new Field(reader.GetName(i), 0);
  head.Add(f);
}
db.mapTable(tableName, head);

int recordNum = 0;

// reading the rows
while (reader.Read())
{
  List<string> kReadRow = new List<string>();
  for (int i = 0; i < reader.FieldCount; i++)
  {
    kReadRow.Add(reader.GetValue(i).ToString());
  }

  db.insertLine(tableName, kReadRow, SE16RowsBatch);
  recordNum++;

  if ((recordNum % 100000) == 0)
  {
    outputMessage(tableName + ": " + recordNum + " records read");
  }
}

db.flushTable(tableName);
reader.Close();
connection.Close();
connection=null;
```

In the provided example taken from the original Sod App's code, if at any point an exception occurs both the reader and the connection are not disposed. When using resources the code should be designed to guarantee proper resource release in any situation. In .NET this could be implemented with Using statement and Try-Catch-Finally statement, for which a more detailed description of usage is provided in section 3.7.

- Buffer Overflow: in languages where the developer has direct responsibility for managing memory allocation, such as C and C++, there is potential for a Buffer Overflow. While the most serious risk related to a buffer overflow is the ability to execute arbitrary code on the server, the first risk comes from the denial of service that can happen if the application crashes. In order to avoid overflow the data size should be checked before storing it in memory.

- Too much data in session objects: storing too much information in the user's session, such as large quantities of data retrieved from the database, can cause denial of service issues. An attacker could exploit this mismanagement of memory by creating a number of sessions and increasing their size by requesting large amounts of data. This problem is amplified if session data is also tracked prior to a login, as a user can launch the attack without the need of an account. In order to mitigate the risk a maximum amount of data that can be loaded in memory should be defined for every session. This amount should be fixed and appropriate to the service provided, and the current state should be verified before retrieving new data.

- Account access denial: a common defense to prevent brute-force discovery of user passwords is to lock an account from use for a certain time after a number of failed attempts to login. This means that even if a legitimate users were to provide their valid password, they would be

unable to login to the system until their account has been unlocked. This defense mechanism can be turned into a DoS attack against an application if there is a way to predict valid login accounts. In this case an attacker can lock all the discovered valid accounts by exhausting the available login attempt with random passwords. Even if the accounts are unlocked after a set amount of time the attacker can repeat the process each time the lock timer is up, effectively denying access to the legitimate user indefinitely.

This attack can be mitigated by making it difficult for the attacker to establish whether an account is valid. If after a failed login attempt the server replies with an error message that states which one between the password and the username was incorrect, the attacker will know whether to continue launching the DoS attack on the username or trying another one. Also notifying that the account has been locked only for valid accounts hints information about which accounts are valid, and the attacker will be able to concentrate only on those. An additional way to mitigate the attack is to implement a monitoring system that blocks suspicious traffic.

As a general guideline, when asking the user to upload files there are a set of key steps that must be performed in order to properly secure the web application:

Verify the format: For each file upload function mapped in the previous phase it must be verified which data type is acceptable. A correctly implemented upload function should accept only specific file types and extensions, using a white-list approach. The set of allowed formats should be as restrictive as possible and allow only those pertinent to the specific purpose of the file and properly tested by the developers.

Sometimes however it is necessary to allow a broad set of file types, for example when providing a repository feature to the user. Even in this case the file types should be restricted, investing some time in carefully considering which types to allow.

The most common file types used to transmit malicious code into file upload features are the following:

- Web executable script files: such as aspx, css, swf, xhtml, rhtml, shtml, jsp, js, pl, php, cgi. It is advised to deny those files.
- Microsoft Office document: Word/Excel/Powerpoint using VBA Macro and OLE package. The risk could be mitigated by verifying if the document contains "code"/OLE package, if that's the case then the upload process should be blocked.
- PDF document: malicious code inserted as attachment.
- Images: malicious code embedded into the file or use of binary file with image file extension. In the first case the file will still appear as a normal image but when opened it will also execute the code hidden. It could contain scripts or tags that exploit well-known web application vulnerabilities, such as cross-site scripting (XSS). A possible solution could be to sanitize the incoming image using a re-writing approach and then disable/remove any code present (this approach also handles the case in which the file sent is not an image). The re-writing approach consists into creating a new image from the uploaded image to remove metadata.

It's technically possible to perform sanitizing on Office or PDF documents but it is not advised since the sanitization could be performed incorrectly, missing evasion techniques and allowing a corrupted file to enter the file system, or the resulting file could present document structure/rendering issues.

When verifying the format of the file it is not sufficient to check the extension from the name of the file or the contents of the Content-Type HTTP header, as they are both controlled by the user and might have been tampered to not match the actual file content. Instead the signature header in the file should also be verified.

Check/Change the name: The name of the saved file, even if temporary, should not contain any user controlled input.

It is good practice to rename the uploaded file, to make it more difficult for the attacker to locate it after the upload but also since a malicious file could overwrite another that already exists with the exact same name on the server. Furthermore this approach offers additional security with respect to a standard validation and sanitization approach as it ensures that, even if evasion techniques have been used to bypass the white-list filter, the risk will be mitigated. These evasion techniques could include appending a second file type to the file name (for example image.jpg.php) or using trailing space or dots in the file name.

If it is necessary to keep the original file name it is advised to use a look-up table or a similar approach to link the validated and sanitized user supplied file name to the server created name. In addition this approach could be also used to avoid disclosing to the user the name of the file on the file system, displaying instead only the alias from the look-up table, that could be user supplied or specifically crafted by the application.

Depending on the type of web server, the OS used, etc., there are a number of important files, such as configuration files, that an attacker could try to overwrite. Such attempts should be recognized and denied and there should be a reporting system to notify the suspicious activity from the user.

Examples of such files are the "Web.config" and "App.config" files for .NET, "crossdomain.xml" for Flash, Java and Silverlight, ".htaccess" and ".htpasswd" for Apache, and so on. It is up to the developer to detect the critical files, however the better solution still remains to rename the uploaded files, and use the recognition system only for threat reporting.

Verify the size of the file: Depending on the purpose of the file to upload, the size constraint might differ. However it is advised to choose a maximum upload size sustainable by the server and appropriate to the type of file to be uploaded. In some cases it might be necessary to upload very large files, as is the case for SoD App's sqlite database upload. The size of such databases could go up to several GB, nevertheless a maximum allowed size should always be implemented.

When dealing with these situations there are a number of approaches that can be taken in order to mitigate the risk of DoS attacks. First of all a maximum number of contemporary uploads that a user can perform should be defined. The upload process should then be monitored and when the cached file exceeds the allowed threshold the upload should be interrupted.

Limit the quantity of data a user can store: The website should allow each user to store on the server a limited amount of data in order to mitigate the risk of a DoS attacks in which the attacker saturates the server's file system by uploading a great number of files. This can be achieved in many ways, a simple approach is to define an attribute for the User object that stores the amount of data currently in the database for the specified user, and is updated every time a new file is uploaded or deleted.

In addition to limiting the data on disk also session data cached by the server should be limited by properly designing the caching system. If data is explicitly inserted by the developer in cache then the oldest data related to the specific user could be removed if the user reaches a set limit.

Another way to mitigate the deterioration of the service due to an overburdened memory is to forbid the excessive growth of session objects. These objects, if correctly implemented, should not hold large amount of data but code or design errors might lead to their uncontrolled growth.

Examine file content: Upon validating the file the developer could also try to sanitize a file or detect if it contains code. For some files there are well known header and trailer signatures that enclose the actual data of the file, anything outside of these signatures could be removed or treated as suspicious. In addition often code is hidden as metadata, therefore another sanitization action could be to remove the metadata from the file.

However these actions offer very limited reliability, a more secure approach is to have a valid anti-virus scan the uploaded files before storing them and to configure the server to deny the execution of files in the destination folder.

If the website supports Zip file upload, it is important to do a validation check before unzipping the file. The check should verify the target path specified by the Zip file, the level of compression and the estimated unzip size.

A Zip file may incorporate a malicious file that specifies as unzip target a path such as "..\..\..\..\malicious.php" trying to gain unauthorised access to specific locations of the file system. This type of attack is called Directory Traversal. Furthermore the Zip algorithm can be recursively applied on a file, therefore a properly crafted file that has multiple levels of Zip compression could go from several petabytes in the uncompressed file to a dozen kilobytes in the final .zip.

This type of compression is called a Zip Bomb and is a DoS attack that aims at exhausting the memory of the program or system reading it in order to render it inoperable or crush it.

Secure the save path: The directory to which files are uploaded should be outside of the website root. The website root directory is public and reachable through the domain name, hence if files are uploaded in one of its sub-directories the contents might be reachable by an attacker. On the contrary files located outside of the rood directory cannot be directly reached, therefore they are less vulnerable.

Check correctness of Content-Type header when serving the file: It should be verified that the uploaded files are served to the users with the expected MIME type in the Content-Type HTTP header, to avoid client-side execution of malicious code.

Management of Variables Storing the Input

When dealing with user input that might contain or contains sensitive data, such as passwords and personal information, it is good practice to clean the variables containing this information as soon as it is processed. The same should also be applied to data retrieved from the database.

In most managed languages, when using managed objects, if no explicit action is taken by the developer the garbage collector will clean the object from memory when it falls out of scope. However it is a non-deterministic process, it will not happen immediately but it will be simply scheduled and executed when the garbage collector sees fit.

In addition there is the common misconception that rewriting the value effectively removes from memory the previous content. However depending on the language used rewriting the value might not provide any security at all, resulting instead in uncontrolled multiplication of the sensitive information in memory. Such is the case for C#, for which a possible alternative in provided in the Implementation chapter 3.2.2. Therefore before taking this approach the actual management of strings should be verified for the used language, to define the correct approach.

As long as data remains in memory it is vulnerable to attacks that target the RAM of the server, such as buffer over-read, RAM scraping, memory dumps theft, etc.

Often these types of attacks are difficult to spot as they exploit software and hardware bugs to gain access to the memory, which usually contains some amount of sensitive information at any given time. In order to mitigate the consequences of such leaks it is necessary to pay special attention in disposing of sensitive data as soon as possible, without waiting for the garbage collector.

2.2.3 Authentication and Session Management

User ID

An user identifier is a string that unequivocally identifies the user. As such the application should guarantee it's uniqueness when registering a new account, by verifying that the username chosen by the user is not already in use.

When acquiring the user ID from user input the same validation and sanitization rules already described should apply. In this case it is advised to limit the character set to only safe character as the user ID is used for many purposes and is likely to be one of the first targets of attackers.

When using an email as user identifier it is important to verify that the format of the email is properly managed.

Many developers implement complex regular expressions to validate the email that often, relying on incorrect assumptions based on the most common email formats, allow a set of formats an characters more restrictive than those actually allowed by the RFC standards[12].

A regular expression that correctly models the allowed character sets and combinations is complex to define, error prone and requires to be maintained up to standards. Furthermore such expression would have to allow also characters considered dangerous for the scope of the web application. Hence this step can be avoided, relying instead on other types of controls to guarantee security and validity.

In order to validate the email it is sufficient to perform the following controls [13], however it is advised to use email validation libraries instead of regular expressions.

- Check for the presence of at least one "@" symbol in the provided address.
- Verify that the local-part is not longer than 64 bytes.
- Ensure the domain is not longer than 255 bytes.
- Verify that the address is legitimate.

The first three checks perform a first skimming of invalid addresses provided, however it's the last step that actually validates the email. The website will send an email to the provided address and require confirmation from the user. In this way it is possible both to verify the validity of the address and to ensure that the user actually owns the email provided.

In order to secure the input related to the email address against injection attacks the email should be properly encoded for use with the website and decoded when used to send emails.

To further improve the security of the user identifier it is possible to define two separate identifiers: the first that is the email or username chosen by the user and would be the only identifier of the two visible to the user, the other that is a server generated string and is the actual identifier used by the application. The purpose of creating two identifiers is to hide the actual value used for database access and user identification, making it more difficult for an attacker to target data relative to a specific user, as he would most likely not know the real identifier. This solution however requires the management of two separate identifiers that must be both unique.

An intermediate solution is to use only one identifier that, however, is defined by the server and is more difficult to guess than an identifier defined form public data (email) or easily predictable data (user input).

Session Management

Another important aspect that must be evaluated is the correct management of sessions. HTTP is a stateless protocol therefore it is completely in the hands of the developer to take care of session management, and the relative aspects of authentication and access control. Once an authenticated session has been established, the session ID (or token) is temporarily equivalent to the strongest authentication method used by the application, binding the user authentication credentials (in the form of a user session) to the user HTTP traffic. This means that if not correctly designed the tokens can allow an attacker to authenticate directly as the user for which the token was created. This places these tokens at high risk of exploitation, in fact the OWASP Top 10 places at second position of most critical security risks those related to broken authentication and session management.

There are two main approaches to session verification: Permissive and Strict. The permissive mechanism allows the web application to initially accept any session ID value set by the user as valid, creating a new session for it, while the strict mechanism enforces that the web application

will only accept session ID values that have been previously generated by the web application. It is important to verify that the web application does not implement the Permissive mechanism, for instance PHP uses the Permissive approach by default , as it leaves it vulnerable to session fixation attacks, that will be explained in detail in the next paragraphs.

In order to establish the security of the implemented sessions the following should be evaluated:

- **Session name**: it should be as generic as possible in order to give the least information about it's purpose and the technologies and programming languages used by the web application. For example session ID names used by most frameworks, if left unchanged, have a standard value that can easily disclose the framework used. Some examples are PHPSESSID (PHP), JSESSIONID (J2EE), ASP.NET_SessionId (ASP .NET), etc. It is clear that any attacker inspecting the packets can easily deduce the purpose and format of the token. If instead the name is changed to a generic "ID" the attacker will not be able to extract the same information as easily.

- **Token length**: the session ID must be long enough to prevent brute-force attacks, where an attacker can go through the whole range of ID values and verify the existence of valid sessions. This type of brute-force attack is also known as **Search Attack** and consist of exploring all possible combinations of a given character set for the given token length. This kind of attack is slower the bigger is the space of possible candidates. For example the total number of values to try for a token long 20 characters of numeric value (0-9) is 20ˆ10, which means over 10 trillion tries.

  The current best practice requires tokens of at least 128 bits, as they still provide sufficient protection. The length however depends also on the *entropy* of the token, if the attacker is able to predict the distribution of values he can greatly reduce the number of tries necessary to find the actual value.

  SoD App uses an ASP.NET_SessionId that has maximum length set to 120. Currently it is considered sufficient thanks to the entropy of the token so no modifications where applied [14]. There is no official support for longer tokens therefore in this situation if a higher level of security is required a different solution should be designed.

- **Token value**: the value of the token should not contain any information but should be a meaningless identifier. If part of the ID is composed by details specific to the user it might both result in information disclosure and the token being predictable. Information regarding each session should be stored on the server in specific objects that can be retrieved using the session identifier. Hence the token would constitute the identifier used to access the object and could be designed to be a meaningless and random string. In addition being able to generate random identifiers increases the **entropy** of the token. Session ID tokens whose value is random are more secure since it is impossible for an attacker to guess or predict the ID of a valid session through statistical analysis techniques. For this purpose, a good PRNG (Pseudo Random Number Generator) should be used to generate at least 64 of the bits of the token. This lower threshold is estimated based on the following equation [15]:

$$T = \frac{(2^E) + 1}{2G \times S}$$

  Where:

  T is the expected number of seconds required to guess a valid session identifier.

  E is the number of bits of entropy in the session identifier.

  G is the number of guesses an attacker can try each second.

  S is the mean value number of valid session identifiers that are valid and available to be guessed at any given time.

  Assuming a 64 bit session identifier, 32 bits of entropy, an attacking power of 1000 guesses per second and approximately 10 000 valid session identifiers at any given moment, the expected time for an attacker to successfully guess a valid session identifier is about 7 minutes.

T however grows exponentially if considering a 128 bit session identifier that provides 64 bits of entropy. Even considering a stronger attack, for example 10 000 guesses per second, and a larger pool of active session IDs available to be guessed, for instance 100 000 , the expected time for an attacker to successfully guess a valid session identifier is greater than 292 years.

- **Session Implementation**: session tokens can be managed in different ways, such as cookies, URL parameters, URL arguments on GET requests, body arguments on POST requests, hidden form fields (HTML forms), or proprietary HTTP headers. However it is advised to choose a solution that allows defining advanced token properties, such as the token expiration date and time, or granular usage constraints. This is one of the reasons why cookies are one of the most extensively used session ID exchange mechanisms, offering advanced capabilities not available in other methods. Solutions that incorporate the token in the URL should be avoided as they facilitate a number of attacks such as the manipulation of the ID, session hijacking or session fixation attacks.

  > "The disclosure, capture, prediction, brute-force, or fixation of the session ID will lead to session hijacking (or sidejacking) attacks, where an attacker is able to fully impersonate a victim user in the web application. Attackers can perform two types of session hijacking attacks, targeted or generic. In a targeted attack, the attacker's goal is to impersonate a specific (or privileged) web application victim user. For generic attacks, the attacker's goal is to impersonate (or get access as) any valid or legitimate user in the web application." [14]

  In **session hijacking** the attacker steals the established session between the client and the Web Server after the user authenticates. In the **session fixation** attack instead the user is induced to authenticate using a session ID provided by the attacker, then the session is hijacked thanks to the attacker's knowledge of the ID. A website is vulnerable to this type of attack if after the authentication the ID is not regenerated.

  An example attack could work out as follows: The attacker obtains a valid session ID, for example by connecting to the website, and then terminates his session to free the ID. In order to induce the victim to establish a connection using the known session ID there are different options that the attacker could exploit. He could insert the token in the URL argument and then send it to the victim in a hyperlink that would be then used to access the website. Alternatively the attacker could put the session ID in a hidden form field and trick the victim in authenticating with the malicious login form, for example by hosting it on a website or directly in an HTML formatted e-mail. In case the website uses cookies there are yet different ways to trick the victim into using the malicious session ID and they include client-side scripts, injection attacks or the use of HTTP header responses that are sent to the victim by the attacker, impersonating the website and exploiting the Set-Cookie parameter. The victim will therefore establish an unauthenticated session with the server using the session ID provided by the attacker and will then proceed with the authentication. If upon performing authentication the web server does not close the previous session and open a new one with a different session ID the attacker will be able to hijack the authenticated session and will have successfully performed a session fixation attack.

  In general the session ID must be renewed or regenerated by the web application after any privilege level change within the associated user session, and is especially important during the authentication process, as the privilege level of the user changes from the unauthenticated (or anonymous) state to the authenticated state. However other common scenarios must also be considered, such as password changes, permission changes or switching from a regular user role to an administrator role within the web application.

  The objects containing information relative to the active sessions must be stored in a secure way, to avoid accidental or unauthorized access.

  The application should not accept session tokens provided with any mechanism other then the secured one. Even if a web application makes use of cookies as its default session ID exchange mechanism, it might accept other exchange mechanisms too. It is therefore required to confirm via thorough testing all the different mechanisms currently accepted by the web application and properly disable them.

- **Cookies Implementation**: cookies should be the preferential mechanism to establish sessions as they provide a set of useful features:

  Secure attribute: instructs web browsers to only send the specified cookie through HTTPS. This feature is especially helpful when leaving the HTTP port active even if redirect is enabled, as the server could still be tricked to send the cookie trough an unencrypted connection.

  HttpOnly attribute: instructs web browsers to deny to scripts access to cookies through the DOM document.cookie object, to protect them from XSS attacks. **Cross-site Scripting (XSS)** attacks are a type of injection attack in which malicious code, meant to be run on client side, is injected in a website. This attack is very common and is ranked at position 7 of the OWASP Top 10 of most critical security risks. It usually exploits websites that use user input to generate an output to the user without properly validating and sanitizing it.

  SameSite attribute: prevents the browser from sending this cookie along with cross-site requests, avoiding information leakage and CSRF attacks. **Cross-Site Request Forgery (CSRF)** is a type of attack similar to session fixation, in this case the victim is induced to submit a request, crafted by the attacker, to the web server. The request will be designed to execute malicious actions such as executing payments, changing user information, etc.
  As for the session fixation attack, after the attacker has crafted the request for a target website, he will trick the victim into sending it to the server with the help of social engineering (sending the link/form via email, chat, etc.). If the user has an account for the targeted website there are two cases in which the attack will be successful:

  1. If the user currently has an active authenticated session with the website the request will be automatically submitted using that session. This is actually quite common considering for example an employee that uses a specific company web application often enough during the day that he has always a tab open in the browser for that website.
  2. The browser is configured to automatically authenticate the user for the specific website. This is a commonly used configuration as it removes the hassle of having to login each time. Browser requests will automatically include any credentials associated with the site, such as the user's session cookie, IP address, domain credentials, and so forth. The site will have no way to distinguish between the forged request sent by the victim and a legitimate request sent by the victim.

  This specific attack is not aimed at retrieving data, as the attacker would not be able to see the response of the server, but it is meant primarily for performing actions and changing the state of the user's account.
  The SameSite attribute offers two possible values: lax or strict.
  The **strict** value is the more secure option and will prevent the cookie from being sent by the browser to the target site in all cross-site browsing context, even when following a regular link. However this option might not be always appropriate, for example for a GitHub-like website this would mean that if a logged-in user follows a link to a private GitHub project posted on a corporate discussion forum or email, GitHub would not receive the session cookie and the user would not be able to access the project.
  In a banking website context, however, the strict flag would be the most appropriate, since it is unlikely that the website would want to allow any transactional pages to be linked from external sites.
  The **lax** on the other hands trades a bit of security for improving usability for websites that want to maintain user's logged-in session after the user arrives from an external link. In the above GitHub scenario, the session cookie would be allowed when following a regular link from an external website while blocking it in CSRF-prone request methods (for example POST).

Usually the default value is lax, however in some frameworks it might be set to none, disabling the use of the attribute, so it would be best to explicitly set it to the required value.

Domain attribute: this attribute specifies to the browser to which domains can the cookie be sent. If it is not set by default the cookie will only be sent to the origin server, however it can be set to specify additional domains or subdomains. For this attribute the most secure choice is to leave the attribute unset, as permitting additional domains allows an attacker to launch attacks on the session IDs between different hosts and web applications belonging to the same domain.

Path attribute: this attribute further restricts the destination of sent cookies to a specific directory, subdirectories or resource within the web application. This attribute should be set as restrictive as possible by specifying the application path that makes use of the session ID. By default the path is set to "/" which is the server root. When setting the attribute the Path property extends the Domain property to completely describe the specific URL to which the cookie applies. For example, in the URL http:/www.domain.com/home, the domain is www.domain.com and the path is /home.

Expire and Max-Age attributes: The Expire attribute is an older implementation and sets the expiry date for when a cookie gets deleted, using the format "Expires: Tue, 15 May 2007 07:19:00 GMT".
The Max-Age attribute is a more recent implementation and sets the time in seconds for when a cookie will be deleted, using the format "Cache-Control: max-age=3600".
It is advised to use the Max-Age attribute as it is less error prone and simpler to manage. If the Expire attribute is used it should be tested that there are no errors and cache misinterpretations derived from the format, the zone conversions, etc. Both attributes however are used to implement the distinction between non-persistent and persistent cookies.

Non-persistent cookies: this type of cookies, also referred to as session cookies, do not present the Max-Age or Expiration attribute and are therefore deleted when the web browser instance is closed, forcing the session to disappear from the client. It is highly recommended to use non-persistent cookies for session management purposes, so that the session ID does not remain on the web client cache for long periods of time, from where an attacker could obtain it.

Persistent cookies: on the other hand when the Max-Age or Expiration attribute is set the cookie will be stored by the browser for as long as specified and will be available the next time the browser is launched. These cookies should be used for user preferences and other less sensitive information.

Therefore Expire and Max-Age must not be set for session cookies.

All these cookie attributes, however, are not supported by all browsers. In addition not even all web servers support them, even if properly configured. It is therefore advised to always test these flags after having implemented them, in order to verify that they are correctly incorporated in the server responses.

- **Session Expiration:** Sessions should not be kept active until the user logs out or the browser session is terminated, instead there should be a set expiration time imposed by the web application. Reducing the lifetime of session IDs results in also reducing the time an attacker has to launch attacks on active sessions. When a session is regenerated it is most likely that all progress obtained by the attacker is lost, and in case the attacker had managed to obtain the token he will have a short window of time to exploit it. Hence the smaller is the validity window the more secure is the session, however the session expiration timeout values must be set accordingly with the purpose and nature of the web application, and balance security and usability, so that the user can comfortably complete the operations

within the web application without his session frequently expiring. A trade-off is to set shorter expiration times if the session is idle.

Idle timeout values could range between 2-5 minutes for applications that require high levels of security or 15-30 minutes for low risk applications. Absolute expiration times, for when the user is active, should be defined based on the mean time estimated to be necessary for users to perform the main operations on the website.

Idle timeout should be calculated from the last HTTP/HTTPS request received by the server for the specific session. This timeout reduces significantly the window of exposure to attacks but is irrelevant if the attacker has already managed to hijack the session.

Absolute timeout defines the maximum amount of time a session can be active, regardless from user activity. When the absolute timeout expires, the session ID should be properly invalidated and the user should re-authenticate. This value effectively reduces the amount of time an attacker could use an hijacked session.

Renewal timeout is an additional approach in which the session ID is automatically renewed and provided to the client. For a short amount of time the web server will keep both session IDs to give enough time to the client to become aware of the renewal. This approach does not require re-authentication and therefore should be used to complement the previous timeouts, but it has the advantage that it minimises the amount of time a session ID is valid even when the user is active, and therefore the time an attacker would have to perform attacks on a specific ID or use it. Depending of the implementation, there could be a race condition in which the attacker with a still valid previous session ID sends a request before the victim user, right after the renewal timeout has just expired, and obtains first the value for the renewed session ID. In this scenario, however, the victim user might be aware of the attack as his session would be suddenly terminated because the associated session ID is not valid anymore.

Timeouts must be enforced completely by the server and should rely only on values contained in the session object stored on the server. Using the session token itself or other client parameters received through the client requests is completely insecure as they could be easily tampered by an attacker.

When the session expires or the user logs out the ID must be actively invalidated both for the server and the client. A good approach is to set an invalid or empty value for the session ID, set the Expires or Max-Age attribute to a date from the past (or 0 for Max-Age) if a persistent cookie is used, and call the appropriate termination functions (for example Session.Abandon for .NET).

It should be verified that the web application provides a logout button that is easy to reach for the user and that effectively terminates the active session, correctly invalidating the ID.

It is also possible to use client-side JavaScript code to implement additional security features, that should be not relied on, but that can offer and additional layer of security. An example of such an implementation might be a script that performs the logout when the browser window is closed, as if it was done manually by the user, immediately terminating the session also for the server.

Session IDs should be treated as any other input and properly sanitized and validated, filtering out invalid IDs before processing them and implementing Strict session management, perhaps even recognising when a user provides a session ID that was not generated by the server and raising an alert.

2.2.4 Password Management and Policies

In order to guarantee stronger security for the website accounts, proper password rules and policies should be implemented.

Password Strength

Password strength is defined by length and complexity, the application should impose a minimum values for these two parameters.

Standards define as weak those passwords that are shorter than 7 (PCI DSS [16], 2018) to 10 (NIST SP800-132 [17], 2010) characters. It can be noticed, however, that NIST defined 10 characters weak already in 2010, therefore even if more recent standards might define even weaker lower limits it is clear that nowadays these numbers cannot guarantee a high level of security. In addition these values require the password to have an acceptable entropy, for example containing both alphabetic and numeric characters in case of the PCI DSS standard.

In fact the minimum length for passwords composed of only letters, as is the common case for passphrases, is 20 characters, far more than the 7-10 previously specified. Therefore when allowing the user to choose passwords shorter that 20 characters it is necessary to impose an acceptable level of entropy.

In the following list, the rules applicable on a password are ordered from lowest entropy to highest, based on the pool of possible values that a character can assume:

- Only numbers.
- Only letters.
- Numbers and letters required.
- Upper and lower case letters required.
- Numbers, upper and lower case letters required.
- Numbers, upper and lower case letters and symbols required.

When allowing short passwords, such as 10 characters long, higher levels of entropy should be required. It must be however kept in mind that simply requiring longer passwords and more entropy is not sufficient to guarantee security. The password authentication method is intrinsically insecure, as it has many weak points, such as storage of the password, reuse, guessable values. The avarage user, even if forced to provide a strong password, will most likely do some of the following:

- Choose an easy to remember password, making it easier to guess.
- Reuse the same password for multiple websites, making it more exposed. The other websites with which the password is shared might not manage it securely.
- Write it down somewhere, save it in browser or use a password manager, making it more exposed to attacks.

For applications that require high security, such as banking applications, or to authenticate administrator accounts, the authentication should not rely solely on a password but use multi-factor authentication. This method will be discussed in section 2.2.4.

Additionally the upper limitations should be as lax as possible, allowing the user to choose long and complex passwords. An acceptable upper length limit is of 160 characters. It should be avoided to perform character set or encoding restrictions.

The web application should never truncate or normalise the password in order to conform it to requirements.

When the user submits an unacceptable password the website, along with the refusal notification, should also provide a detailed description of all the rules that are applied for password validation, to avoid unnecessarily stressing the user.

Password Storage

Passwords should not be stored on the server in clear text, as it would expose them in case the database gets stolen. At no point should the password be written unencrypted anywhere on disk.

If this approach is implemented the input of the user is only processed to generate the hash, therefore it is not necessary to perform any sanitization, only validation.

In addition, upon registration, the user provided password should be associated with a Salt with which the hash of the password should be calculated. A different Salt should be generated for each credential, if the user has multiple credentials or the password is renewed the Salt must also change.

The Salt is a random string whose main purpose is to mitigate the effects of brute-force attacks that make use of precomputed dictionaries of hashed words in case the database of passwords is stolen. This type of brute-force attack is know as **Dictionary Attack** and consists in preparing a list of commonly used passwords and precomputing their hashed value. When an attacker manages to breach the defenses of a server and steal the database containing the passwords, what he finds are only the hashes of the passwords. Since hashing algorithms are not reversible, as they perform a lossy operation, in order to exploit the stolen data the cracker will first need to find a string that when hashed generates the same hash of the password. In this way, since the website performs a check on the hashes and not on the actual password, providing the crafted value will be completely equivalent to providing the real password. However, if a good algorithm is used, finding this type of collision is extremely difficult, therefore the attacker creates the aforementioned dictionary and confronts the stolen hashes with the hashes in his dictionary, hoping to find a match. In order for this operation to work the attacker must only be careful to craft a dictionary that uses the same hashing algorithm or key derivative function and parameters as the website, however the pool of possibilities can be easily reduced by looking at the length of the stolen hashes and by considering the most popular options.

This attack however relies on the fact that the input string for the hashing algorithm is exclusively composed by the password. If a Salt is used this assumption fails and the attacker would have to recompute the dictionary considering also the Salt. It is clear that if every password uses a different Salt the attacker has to generate a dictionary for each password, which makes the cracking of the whole database extremely expensive for the attacker.

On the contrary if the same Salt is used for all the users it's purpose is completely defeated.

For a correct storage of passwords the tester should verify that the following steps are performed each time a user creates a new credential or changes the password:

1. A new Salt is generated. It is not necessary to verify whether it's value is unique, as long as it is different for the majority of users a few collisions for the Salt values are acceptable.
2. The Salt string is appended or prepended to the password.
3. The concatenated Salt and password are hashed.
4. The variable storing the password is properly disposed.

As already mentioned in the Input Management section 2.2.2, when managing sensitive information the used variables should be correctly disposed as soon as they have been processed, to shorten as much as possible the persistence of such data on memory.

Not all hashing algorithms are appropriate for this task, it's important to use good cryptographic hashing algorithms, that guarantee the least amount of collision, increasing the complexity of finding a value that generates the same hash. If a certain algorithm has different versions providing different hash lengths, as a general rule the longer is the hash generated the more secure is the algorithm.

Good choices include: RIPEMD-160, SHA-2, SHA-3.

Bad choices include: MD2, MD4, MD5, SHA-1.

SHA-3 is the choice that offers the most security among the proposed.

However when hashing passwords the use of key derivative functions is preferred over direct use of the algorithms, as they are specifically designed to introduce complexity in the computation.

The Salt and hash are the only values that should be stored on the database and when a user authenticates the following steps should be performed:

1. The password is validated.
2. The hash stored in database is retrieved together with the associated Salt.
3. The provided password is concatenated to the retrieved Salt in the same order performed during the creation of the password.
4. The concatenated password and Salt are hashed with the same hashing function used during the creation of the password.
5. The calculated hash and the retrieved hash are confronted. If they are equal the user is authenticated.
6. The variable storing the password is properly disposed or rewritten.

When analysing the code it must be verified that the Salt is unique for each credential (if a user has multiple credentials each should have a different hash), generated at the time of registration and regenerated for each password renewal. It is not necessary to keep the Salt secret as it's only purpose is to make dictionary attacks more complex.

The role of the Salt is a common cause of misconception for many developers, that implement it's use without fully understanding it's purpose, resulting in incorrect and insecure implementations. Such an example was found during the analysis of SoD App, where the Salt was a hard-coded value that was never renewed and was the same for all users. This approach completely defeats the purpose. The developer most likely believed that if the Salt is kept secret then it is equally secure to reuse the same value. This is a very common misconception. Many people believe that "security through obscurity" is able to provide an additional layer of protection, when in fact it only gives a false sense of security.

In this specific case a fixed value, that being hard-coded does not have any renewal policy, is a single point of failure for the implemented protection system, and is bound to be discovered. Returning to the example of the cracker managing to steal the database of passwords, looking at the database the attacker will see that no Salt is stored for each password. The attacker could make different hypoteses, among which the most likely would be that either no Salt is used or all the passwords share the same value. The attacker could already have an account created by him in the database to confirm the hypotheses or if the breach has not been detected yet and the website is still running the attacker can easily verify his assumptions by creating a new account.

In case a common Salt is used there are a number of different approaches he can use to obtain it:

- Social engineering: trick or bribe an employee to reveal it.
- Brute-force: if the attacker had created an account previous to the breach he would be able to brute-force the Salt by knowing the actual password to the hash belonging to his account.
- Code reverse engineering: if during the breach he managed to obtain also the executables and libraries of the web application he could try to reverse them to the source code.
- Internal repositories: if the attacker is able to gain access to the source code he can easily retrieve the value.

This list obviously does not cover all the possibilities available to the attacker, however it should be sufficient to highlight that relying on the secrecy of a single value is not secure.

The correct implementation on the other hand does not rely on guaranteeing secrecy, but on the practical unfeasibility of the attack under the given circumstances.

Password Recovery

When verifying the password recovery procedure the tester should check whether the following applies:

- The website does not send the password to the user, independently from the channel used. As already explained the website should not even store the password on the database, only the hash. Furthermore if email is used it constitutes a very insecure channel and if the user does not delete the email it will remain in the provider's database and email application stored in clear text indefinitely.
- The website should generate a unique reset token to be sent to the user along with the reset URL. The token can either be manually inserted by the user in the pointed page or the URL can already contain it as parameter. The website should avoid solutions in which a new password is automatically generated and sent to the user, because in addition to security concerns detailed in the previous point, an attacker could exploit this by launching a DoS attack in which he continuously regenerates the user's password denying him access.
- The reset token should have an expiration time. The window of time should be set as narrow as possible, allowing enough time for the user to receive the email but reducing the amount of time an attacker would have to obtain the token and use it. A possible range could be between 30 minutes and 1 hour.
- After sending the token the website should not give any information regarding the success of the procedure. The user should be notified with a message such as "An email has been sent to the provided address." regardless if the email provided is associated to any account, otherwise an attacker could try to use use this function to enumerate email addresses to find one that is connected to an account. If there is no user associated with the provided address the website could send an email anyway notifying the recipient that someone has tried to access with their address.

  Email enumeration can be specifically dangerous because it is usually easy to associate an email to an identity, therefore an attacker could exploit the information in social engineering attacks towards the email possessor and it could additionally result in privacy issues, for example with websites that offer services that could damage the user's reputation.
- After authentication the reset token associated to the user is deleted. Tokens should be usable only once and within their expiration window.

It is also possible to improve security by implementing an additional verification step. The website, after the user has provided the renewal token and before allowing the user to provide a new password, could ask for an additional information, such as answers to a set of personal questions defined upon registration. This additional step is recommended as it is not implausible that the user's email has been compromised. In addition the website should lock the reset process for a set time if wrong answers are repeatedly provided.

Password Policies

The web application should force the users to change the password regularly. The window of validity is decided depending on the level of security required by the application and the complexity forced on passwords. If frequent changes are required the time-frame in which an attacker can exploit a cracked password is reduced to the time remaining until expiration. In addition brute-force attacks have less time to find matches and if the password is of appropriate length it could become unfeasible for most attackers. Therefore the website implements weak password requirements the password renewal should be more frequent. However usability should also be considered, in order not to stress the users excessively.

In addition an history of old passwords, in the form of Salt+hash, should be kept to prevent the user from reusing the same passwords. The history should be maintained for a window of time that takes into consideration the required level of security and the medium lifetime of accounts.

Alternative Authentication Mechanisms

The factors that can be used for authentication can be condensed to 3 key elements:

1. Knowledge: something the user knows, such as a password.
2. Ownership: something the user possesses, such as a badge, a smart card, a mobile phone, a token, etc.
3. Inherence: some physical characteristic of the user, biometric data such as fingerprints, retina, etc.

Multi Factor Authentication is a method of authentication that relies on the use of two or all three types of factors.

Unless very high levels of security are required, it is advised to avoid collecting biometric data as it is categorised as sensitive data and requires a high level of protection when managed by the company. In addition if biometric data is compromised it is not possible to simply change it.

In general a sufficient level of security can be reached through the use of the first two factors, for example adding to the use of passwords the use of OTP tokens.

2.2.5 Data Management

The application should avoid as much as possible storing in the database users' personal data that is not of public domain. Storing this type of information constitutes a great responsibility for the company, and if it is breached the reputation damage can be considerable. In addition if the personal data is also categorised as sensitive, depending on local privacy laws, the company must provide adequate security measures and transparent processing procedures. The consequences of a breach involving this type of data could be not only reputational but also pecuniary.

As for Europe, the General Data Protection Regulation(GDPR) (EU) 2016/679 in Article 32 (1) states:

> "Taking into account the state of the art, the costs of implementation and the nature, scope, context and purposes of processing as well as the risk of varying likelihood and severity for the rights and freedoms of natural persons, the controller and the processor shall implement appropriate technical and organisational measures to ensure a level of security appropriate to the risk, ..."

In addition the GDPR comes with severe fines, where infringements of the regulation can lead to fines up to 20 000 000 EUR, or in the case of an undertaking, up to 4 % of the total worldwide annual turnover of the preceding financial year (Art.83 (4-5-6)). Infringements however are not only related to technical issues, but also to organisational issues, such as insufficient training of the employees, improper data management documentation and procedures, improper data subject support, etc. It is a process that requires a lot of attention therefore it is advised to avoid storing sensitive data as much as possible and when it is unavoidable to remove it from the databases as soon as possible.

The risks related to sensitive data can be mitigated by using processing techniques such as tokenisation, truncation and pseudonymisation.

If the website needs to implement a transaction feature to allow credit card payments it is highly recommended to avoid custom solutions and utilize third party payment providers to perform payments and store credit card information for recurring billing. Developing this type of service is extremely difficult and at high risk of attack, therefore delegate this responsibility to a specialised third party not only guarantees more security but unburdens the company from the management of the related sensitive data. In addition users are more likely to trust a well known provider rather than disclosing credit card information to the website.

In the Black Box analysis phase the tester should have mapped all the types of data collected and their purpose. Analysing the results it is necessary to ascertain that the different types of data are being correctly managed.

As already explained in the Password Management and Policies section, passwords are protected employing the hashing process, however there are other types of data that must be protected but cannot be stored hashed, such as keys used for multi-factor authentication, credit card information, etc., as their value cannot be lost. For these types of data encryption should be employed.

Depending on the purpose of the encrypted data, incorrect uses of encryption algorithms may result in sensitive data exposure, key leakage, broken authentication, insecure sessions and spoofing attacks.

When testing the correct implementation of encryption the following should be verified:

Used cyphers :

The web application should avoid using weak encryption algorithms. Based on NIST specifications [18], at the time of redaction of this dissertation, the approved symmetric key algorithms for encryption/decryption are: Advanced Encryption Standard (AES) and Triple Data Encryption Standard (TDES). In addition to the right choices of algorithm, it is also important to choose an appropriate mode of operation. It is recommended to avoid ECB (Electronic Code Book) mode for use with asymmetric encryption, as it does not consider any positional information regarding the blocks and therefore an attack that swaps the order of encrypted blocks could go unnoticed.

Acceptable modes are: Cipher Block Chaining (CBC), Cipher Feedback(CFB), Output Feedback (OFB), Counter (CTR), Galois Counter Mode (GCM), Counter with CBC-MAC (CCM) and Offset Codebook Mode (OCM). Where possible, modes offering authenticated encryption should be preferred.

Whereas the approved asymmetric key algorithms are: Digital Signature Algorithm (DSA), RSA, Elliptic Curve Digital Signature Algorithm (ECDSA).

Symmetric algorithms should be used for direct encryption of data, as they are simpler and faster algorithms. On the other hand asymmetric algorithms are better suited for key exchange, authentication (digital signature) and integrity (digital signature + digest).

Key length :

The tester should establish what the application's minimum computational resistance to attack should be, taking into consideration how appealing is the information managed by the application to an attacker, how long data needs to be protected, where it is stored and if it is exposed. Identifying the required resistance to attacks helps defining the minimum length of the cryptographic keys required to protect data over the life of that data.

According to OWASP the minimum key lengths acceptable, in spite of the evaluations, are:

Algorithm	*Minimum Key Length*
AES	128
TDES	56 x 3 indipendent keys
DSA	2048
RSA	2048
ECDSA	224-255

Key storage :

The keys used to encrypt data should be stored in a secure location that is different from where the encrypted data is stored. This means keys shouldn't be stored on web servers, database servers etc., but in specific key-storage devices.

Keys must be protected on both volatile and persistent memory, ideally processed within secure cryptographic modules, minimising the time in which they are not in encrypted form.

In addition when encrypting keys for storage or distribution, always encrypt a cryptographic key with another key of equal or greater cryptographic strength.

The tester should ensure that keys have integrity protections applied while in storage, and preferably use Authenticated Encryption (AE) modes under an uniform API. AE has the advantage of performing both encryption and authentication operations using one algorithm and one key, providing more speed and security with respect to the combination of different algorithms.

Recommended modes include OCB 2.0, CCM and GCM.

Encryption at rest :

Data at rest, as opposed to data in use and data in transit, means inactive data that is stored physically in any digital form (e.g. databases, data warehouses, spreadsheets, archives, tapes, off-site backups, mobile devices etc.).

However defining the boundary between inactive data and data in use is complex and there are many different definitions. In the scope of this dissertation it will be considered inactive all data in computer storage, while excluding data that is traversing a network or temporarily residing in computer memory to be read or updated.

Data at rest should always be encrypted. Some database manager applications offer integrated data at rest encryption, however if this feature is not available the encryption shoul be performed manually by the developer.

Encryption can be applied at disk level, file level, column level, etc. It is advised to use multiple layers of encryption by applying a first encryption for all files and columns that contain sensitive data, using a different key for each item, and then applying a disk wide encryption including all data. Keys should not be reused.

Hardware cryptographic modules are preferred over software cryptographic modules for protection. This includes key generators, key-transport devices, key loaders, cryptographic modules, and key-storage devices. Caching should be disabled for responses that contain sensitive data.

2.2.6 Keys Management

Keys should be changed periodically, both for current data and old backups. In order to do so the application should provide to administrators an integrated feature to easily change keys. Such feature should consider two main scenarios:

- Periodic maintenance: the web application should allow administrators to periodically change all the keys with new ones and data should be re-encrypted. In order to facilitate this process there should be also support for scheduling and logging of applied changes, in addition the application should have a mechanism to distinguish key versions.

- Emergency: the application should provide the possibility to select different levels of granularity for data re-encryption. In case of emergency it might be necessary to act fast and on specific sets of data, therefore having this feature supported might significantly improve the rapidity of remediation. In addition the web application should provide multiple algorithm options when changing keys, in order to be able to rapidly secure the data in case a new vulnerability is discovered for the version of the used algorithm.

Administrators should have all the information they need, readily available, when rotation of encryption keys must be performed. Rotating keys should not require changes to source code or other risky deployment measures, since doing this in the middle of an incident would unnecessarily burden the mitigation team.

Additionally the tester should verify that the following applies:

- The key rotation procedure defines an appropriate cryptoperiod, meaning the time span during which a particular cryptographic key can be used for its defined purpose. Considerations for defining the cryptoperiod include, but are not limited to, the strength of the algorithm used, length of the key, risk of key compromise and the sensitivity of the data being encrypted. It is advised to perform key rotation at least annually.
- The key rotation process should remove an old key from the encryption/decryption process and replace it with a new key. All new data entering the system must encrypted with the new key. It is recommended also to decrypt the existing data and re-encrypt it with the new key.
- Encryption keys are changed anytime anyone with knowledge of the keys leaves the company or changes positions.
- For high security systems "Split knowledge" and "Dual control" of keys should be implemented. Split knowledge means that key components are under the control of at least two people who only have knowledge of their own key components. Dual control means that at least two people are required to perform any key management operations. The system will require both credentials to allow key management.

  These two policies are used to eliminate the possibility of one person having complete control over the keys.
- Access to keys is restricted to the fewer number of people necessary, additionally the interfaces for key management should avoid displaying to administrators the value of the keys, minimising as much as possible the exposure of keys.

2.2.7 Error handling

Errors are an inevitable occurrence during the lifetime of a software, therefore it should be verified that they are properly managed. There are two key aspects to consider:

- Whether the application fails safely: when an exception is raised it should be always caught and properly handled by the developer. An uncaught exception will propagate upwards in the stack until it finds a method that manages it, however if this does not happen the application will crash.
- What information is disclosed by error messages: the application should avoid disclosing to the user any specific information regarding the type of error, providing instead generic messages and suggestions. Giving detailed information about the cause of the error is certainly more helpful for a legitimate user, however it discloses to malicious users information that could directly or indirectly hint at the internal structure and mechanisms of the software.

The most common causes of exception being raised are improper input validation, improper release of resources, errors in establishing connections with underlying services, etc. In all situations where a variable state is influenced by factors external to the application the developer should wrap the code using proper statements and should explicitly implement a reaction mechanism, also defining what is displayed to the user.

A popular scenario in which error information is commonly exploited is during authentication. There are two very common cases of failed authentication: there is no account linked to the provided username or the password provided is wrong. While it might be convenient for a honest user to know which one happened, the application should not give this information. Instead it should respond with the same generic error message, for example "Authentication error", for both cases. The reason is that a malicious user could exploit it in order to learn if a certain username exists in the database and try to brute-force the password. On the other hand if the attacker does not know if the username is even legitimate, he could be brute-forcing a non-existent account, therefore wasting time and resources.

Other common scenarios include developers leaving debug code in the release versions, exception messages being used directly as output, etc.

A good example was found during the testing of SoD App. The developers had left in the release version of the application a feature that should have been clearly intended only for the development version. When an error occurred, and it was correctly caught, the application would display to the user a page containing an error warning. The warning page however contained the error message and stack trace of the exception. This can be very convenient for the developers during testing but it's the kind of information that should be never provided to the user. A malicious individual could infer from this information what exactly caused the error and how is the internal workflow of the application designed and could learn how to fabricate an input that would take full advantage of the newly found error.

In addition to correctly managing the errors, the application should also keep a log of their occurrences. These logs should be periodically inspected by an intrusion detection system or by a service of the web application itself, in order to detect anomalies or suspicious activities and report them to administrators.

It is also possible to implement an automatic system to notify the user or lock temporarily the account/source address if an excessive amount of anomalies verify. This can both slow down an attacker and discourage him from proceeding with the attack if he knows that his actions are being monitored.

2.2.8 Code analysis tools

Even if meticulously going through the source code is the most reliable approach, it is undeniable that it is a highly time consuming task and is often an unfeasible operation. Therefore there are a set of useful tools that can be used to perform a first evaluation of the code, allowing the tester to individuate faster at lest the problems reported by the tool's analysis.

The software behind these tools is quite complex, therefore most of the more reliable and multi-language tools are available only under paid licence. Among the many available solutions, KPMG already owned a licence for the static code analysis tool Checkmarx [19], therefore that has been the software used to test SoD App.

Checkmarx, as most of the available alternatives, requires the whole project to be uploaded and as a result of the analysis provides a report. In the report the various vulnerabilities found are cathegorised based on the OWASP classifications, and for each vulnerability a description of the impact and a possible solution are provided. It is a very helpful tool for detecting the most common vulnerabilities as the analysis is quite thorough, and it considerably speeds up the analysis process, however it must be kept in mind that these tools have many limitations and it is still necessary for the tester to examine the code for additional missed risks. Furthermore these tools often find false positives, especially if the developer has designed some functions with an uncommon approach. Fig. 2.2.8 provides an example of report.

2.2.9 Analysis results on SoD App

The White Box analysis using automated tools was performed only at the end of the securing process, as final check. Only some minor vulnerabilities where detected, however they have been purposefully ignored as the exploitation difficulty was high and impact entity was limited. It was decided to focus on more critical vulnerabilities.

As for the manual analysis the following vulnerabilities were found, in addition to those already found during the Black Box analysis:

- No data encryption.
- No legitimacy verification for provided email addresses.
- Use of a hard-coded global Salt for the hashing of the passwords of all users.

CHECKMARX

Scan Results Details

SQL Injection

Query Path:
CSharp\Cx\CSharp High Risk\SQL Injection Version:0

Categories

PCI DSS v3.2: PCI DSS (3.2) - 6.5.1 - Injection flaws - particularly SQL injection
OWASP Top 10 2013: A1-Injection
FISMA 2014: System And Information Integrity
NIST SP 800-53: SI-10 Information Input Validation (P1)

Description

SQL Injection\Path 1:

Severity	High
Result State	To Verify
Online Results	http://
Status	New

Method LoadOnMongo at line 49 of /Service.cs gets user input from the dataDirectory element. This element's value then flows through the code without being properly sanitized or validated, and is eventually used in a database query in method mapTable at line 311 of /SQLiteDBConnector.cs. This may enable an SQL Injection attack.

	Source	Destination
File	/Service.cs	/SQLiteDBConnector.cs
Line	49	339
Object	dataDirectory	cmd

Code Snippet
File Name /Service.cs
Method public bool LoadOnMongo(String dataDirectory, MemoryStream stream)

```
....
49.          public bool LoadOnMongo(String dataDirectory, MemoryStream
stream)
```

File Name /SQLiteDBConnector.cs
Method public void mapTable(string _name, List<Field> fields)

```
....
339.                 SQLiteDataReader reader = cmd.ExecuteReader();
```

SQL Injection\Path 2:

Severity	High
Result State	To Verify

PAGE 12 OF 308

Figure 2.1. Checkmarx report extract.

- Use of hard-coded connection keys.
- No key renewal mechanisms.

- Improper resources release in multiple methods.
- No size verification on uploaded files.
- Uploaded files saved with the user provided name.
- Uploaded files saved in website root directory.
- Outdated and unpatched components with known vulnerabilities.

All these vulnerabilities where successfully mitigated and the implementations can be found in the Implementation chapter, with the exception of the data encryption and email legitimacy verification.

The free version of MongoDB used did not provide encryption at rest functionalities and the manually implemented encryption was affecting too heavily performance when applied to the large databases. Due to insufficient time the task was set aside to be implemented in the future.

Email verification was also set aside in favour of more critical vulnerabilities, as user creation is seldom and currently performed manually by administrators and therefore this verification can be also done manually.

2.3 Analysis of support tools and environment

The securing of a web application should take into consideration not only the web application itself but also the environment's configuration and the security of other components used by the application, such as services, database application, libraries, etc.

The application should not use components or protocols with known vulnerabilities or unsafe configurations.

The popular figure of speech "A chain is only as strong as its weakest link" is very accurate also with respect to the security of a system. Investing time and energy in the hardening of the web application ignoring the underlying system can have very unpleasant consequences. In this regard the tester should also analyse the server's configuration. Some common causes of vulnerability are:

- Insecure OS version.
- Weak physical protection.
- Weak OS user authentication system.
- Unnecessary services installed. Some examples are: printer service, FTP, SMTP, NNTP, etc.
- Remote administration enabled.
- Telnet enabled.
- Services running with over-privileged accounts.
- SSL allowed.
- Weak ciphers for TLS allowed.

Only the actually used services and applications should be left on the server machine, and each should be properly configured to minimize the risk of exploitation. Updates should be applied regularly and unsupported software should be replaced with newer maintained versions.

Also the libraries used by the web application should be periodically updated, and the developer should define a list easily accessible to administrators and containing all the libraries used,

directly and indirectly, and their versions. The administrators should periodically verify that these libraries are up to date and that there are no know vulnerabilities discovered.

It is important to pay attention not only to the release server, when performing environment security evaluation, but also to all the components used for maintenance and development, such as development servers, private repositories, logs, backups etc.

Chapter 3

Implementation of risk mitigation actions

This chapter will explain how to secure the application against the evaluated risks, showing best practices and how the mitigation was implemented in the real case example. All the proposed code refers to a .NET environment using .NET Framework 4.7.2 and IIS 8 as web server.

3.1 Application configuration

3.1.1 HTTPS

In order to migrate the whole application to HTTPS there are a few steps that must be performed. The actual implementation depends on the structure of the web application.

1. **Create SSL public key certificate**: the certificate can be bough from a Certificate Authority (CA) or self signed by the company's internal CA. There are many CAs that provide certificates and when choosing one it should be verified that all the most used browsers list it among their trusted CAs, otherwise the browser will not recognize the certificate as trusted and will display a warning advising the user against navigating on the untrusted website. The same would happen for a self signed certificate.

   When working on a web application that is exposed on Internet it is advised to use self signed certificates only during the testing phase and replace them with certificates purchased from trusted public CAs upon release, otherwise users would be scared by the browser's danger notification and it would have a negative impact on the company's image.

   If the website is to be used only in the intranet by employees it is perfectly acceptable to use self signed certificates, as long as the employees' browsers have the internal CA in their trusted list. This should not be overlooked because if users learn to ignore the browser's notification they will ignore also when a real threat arises. For the same reasons certificates that have expired should never be used. When implementing a self signed certificate there is more flexibility in the configuration and it is possible to tailor the certificate to fit specific needs.

   There are many tools available to create certificates and often the web server's control panel offers functionalities to easily manage certificates and also create new self signed ones. It is also possible to use OpenSSL , an open source, cross-platform cryptography library that provides many interesting functions.

   An example of command that can be used to create a certificate:

   ```
   openssl req -newkey rsa:2048 -nodes -keyout key.pem -x509 -days 365
   -out cert_request.pem
   ```

This command generates a x509 certificate, using an RSA key pair of 2048 bits created on the fly and saved unencrypted in key.pem. The certificate is saved under the name of cert_request.pem and has an expiration time of 360 days. The recommended key length is of 2048 bits or more, as it still provides sufficient security and is estimated to be sufficient until 2030, it is not recommended however to use shorter keys [20]. Before producing the certificate OpenSSL will ask for some additional parameters, they should match those specified for the CA. This is however not the final certificate, it must be first sent to the CA to be approved and signed.

To have the CA sign the certificate:

```
openssl ca -in cert_request.pem -out signed_cert.pem
```

The signed_cert.pem file is the actual certificate, signed by the internal Certificate Authority.

When purchasing certificates the cost can vary considerably depending on the type of certificate and other additional services offered by the CA, such as warranty, etc. The important thing however is what extensions are contained in the certificate. When buying a certificate for public use it is best to choose certificates that conform to the X.509 standard, that defines the format of public key certificates in order to provide sufficient information and avoid compatibility issues.

The last step is to install the certificate on the web server.There will either be a configuration file in which to insert the path to the certificate and key or the web server control panel might provide some specific tool. The IIS Manager allows to add certificates, already signed by a CA, under Server Certificates > Complete Certificate Request.

2. **Force the use of HTTPS**: in order to allow only HTTPS it is possible to disable the ports listening for HTTP connections and enable port 443 for HTTPS, providing also the acquired certificate [Fig. 3.1.1]. However this solution can create some problems for users, especially if the website was previously available through HTTP, as they will try connecting to the website using the old HTTP link but will not be able to find the site not knowing that it was migrated to HTTPS. Therefore a more user friendly solution is to implement HTTPS binding but also leave HTTP, permanently redirecting all request coming to it to HTTPS. This solution however must be carefully implemented to avoid leaving any possibility to connect through HTTP.

   To implement redirection rules some solutions provide management panels for the application in which it is possible to specify the rule and it will be automatically implemented. In other cases it might be necessary to manually implement redirect rules from HTTP to HTTPS. This step varies based on the structure of the web application but is usually very easy to implement and details for the specific case at hand can be easily found online, in general there will be a configuration file in which to implement the redirection rules. For SoD App it was sufficient to add the following lines to the IIS Web.config file:

```
<rewrite>
    <rules>
        <rule name="SoDApp_ssl_redirect" enabled="true"
            patternSyntax="Wildcard" stopProcessing="true">
            <match url="*sodapp*" />
            <conditions logicalGrouping="MatchAny">
                <add input="{HTTPS}" pattern="off" />
            </conditions>
            <action type="Redirect"
                url="https://domain/{HTTPS_HOST}{REQUEST_URI}"
                redirectType="Permanent" />
        </rule>
    </rules>
</rewrite>
```

   This example would result in any URL matching the pattern “http://domain/sodapp/*”, where the asterisk represents the path to a specific resource such as for example

"http://domain/sodapp/homepage", being redirected to "https://domain/sodapp/*" that in the provided example would be "https://domain/sodapp/homepage".

3. **Update hard-coded links**: check the code in order to find if there are any hard-coded links using HTTP. All these links will need to be updated to HTTPS or if possible to relative URLs, otherwise the redirection will return a non-existing resource.

4. **Update Custom JS and AJAX Libraries to HTTPS**: update any custom scripts used by the application so that they point to the HTTPS versions. This also includes 3rd party hosted scripts.

5. **Enable HTTP Strict Transport Security (HSTS)**: Using redirection alone to force HTTPS can still leave the system vulnerable to *downgrade attacks* where crackers force the site to load an insecure version. HTTP Strict Transport Security (HSTS) is a web server directive that instructs web browsers to only use secure connections for all future requests when communicating with the website. The implementation depends on the application structure. In general there should be a configuration file in which to insert the indication to use HSTS. For SoD App it was implemented by adding a rule to the IIS Web.config file already edited with the redirect rule:

```
<rewrite>
    <rules>
        <rule name="SoDApp_ssl_redirect" enabled="true" patternSyntax="Wildcard"
            stopProcessing="true">
            <match url="*sodapp*" />
            <conditions logicalGrouping="MatchAny">
                <add input="{HTTPS}" pattern="off" />
            </conditions>
            <action type="Redirect" url="https://domain/{HTTPS_HOST}{REQUEST_URI}"
                redirectType="Permanent" />
        </rule>
    </rules>
    <outboundRules>
        <rule name="Add_Strict-Transport-Security_when_HTTPS" enabled="true">
            <match serverVariable="RESPONSE_Strict_Transport_Security"
                pattern=".*" />
            <conditions>
                <add input="{HTTPS}" pattern="on" ignoreCase="true" />
            </conditions>
            <action type="Rewrite" value="max-age=31536000" />
        </rule>
    </outboundRules>
</rewrite>
```

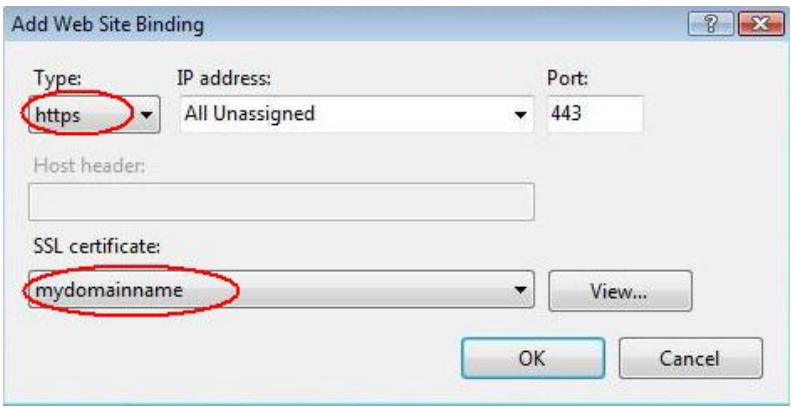

Figure 3.1. Add HTTPS example with IIS

3.1.2 HTTP Methods

In .NET Framework there are different ways these options can be disabled, however following the preference for white-list approaches the following has been deemed as the most secure. Only the managed options are allowed by adding to the App Web.config file the following lines:

```
<configuration>
 <system.webServer>
  <security>
   <requestFiltering>
    <verbs allowUnlisted="false">
     <add verb="GET" allowed="true" />
     <add verb="POST" allowed="true" />
    </verbs>
   </requestFiltering>
  </security>
 </system.webServer>
</configuration>
```

3.1.3 HTTP Response Headers

Removing the headers can be done for .NET by implementing the following function in the Global.asax file:

```
protected void Application_PreSendRequestHeaders()
{
  Response.Headers.Remove("Server");
  Response.Headers.Remove("X-Powered-By");
  Response.Headers.Remove("X-AspNet-Version");
  Response.Headers.Remove("X-AspNetMvc-Version");
}
```

Alternatively is is possible to implement the header removal using rewrite rules added to the IIS Web.config file, an example for the server header:

```
<rewrite>
  <outboundRules rewriteBeforeCache="true">
    <rule name="Remove Server header">
      <match serverVariable="RESPONSE_Server" pattern=".+" />
      <action type="Rewrite" value="" />
    </rule>
  </outboundRules>
</rewrite>
```

If using MVC it is possible to use method attributes in order to restrict an Action method so that it handles only specific requests:

```
public class ProjectController : Controllers
{
  //...
  [HttpPost]
  public ActionResult Create(ProjectModel proj)
  {
    //...
  }
  //...
}
```

In the provided example the element [HttpPost] is the HttpPostAttribute attribute, that restrict the execution of the method Create only in response to POST request.

3.2 Input management

3.2.1 Managing Text Input

In order to provide a practical example the approach used for Sod App will be analysed. In the MVC model the input is received by the back-end Controller method as a Model object, a class defining the parameters contained in the submitted Form, filled in by the user. Validation is performed using method IsValid [21] of the System.Web.Mvc.ModelStateDictionary object defined by the Controller class, evaluating the specifications described in the model using System.ComponentModel.DataAnnotations [22], that define restrictions applied by default to the properties of the class. After validation, the data from the model is used to construct the appropriate Entity Framework [23] object. The Entity Framework is a object-relational mapper (O/RM) and is very useful to reduce the mismatch between the relational and object-oriented elements, enabling developers to write applications that interact with data stored in relational databases using strongly-typed .NET objects that represent the application's domain. It is highly recommended to use O/RM mappers instead of developing ad hoc solutions as the latter would be more error prone and therefore vulnerable. In case an ad hoc solution is used it is necessary to thoroughly test and verify that there is no mismatch between object and relational entities.

If a different model is used the same operations described in this example can be implemented by defining a method that incorporates the actions performed by the DataAnnotations and the ModelState.IsValid method.

```
public class ProjectModel
{
   [Required(ErrorMessage = "Please insert a Name for the Project.")]
   [MaxLength(25)]
   [RegularExpression("^[A-Za-z0-9_]+[A-Za-z0-9 _]*[A-Za-z0-9]+$", ErrorMessage =
       "Please insert a valid project name. Special characters are not allowed.")]
   public string Name { get; set; }

   [Required(ErrorMessage = "Please provide a Description for the Project.")]
   [MaxLength(50)]
   public string Description { get; set; }

   [Required(ErrorMessage = "Please insert a Customer name for the Project")]
   [MaxLength(25)]
   [RegularExpression("^[A-Za-z0-9_]+[A-Za-z0-9 _]*[A-Za-z0-9]+$", ErrorMessage =
       "Please insert a valid customer name. Special characters are not allowed.")]
   public string CustomerName { get; set; }

   [DataType(DataType.Date)]
   [DisplayFormat(DataFormatString = "{0:dd/MM/yyyy}")]
   public DateTime? Date { get; set; }

   [Required(ErrorMessage = "Please insert a System ID for the Project")]
   [MaxLength(3)]
   [RegularExpression("^[A-Za-z0-9]$", ErrorMessage = "System ID must be of 3
       alphanumeric characters.")]
   public string SystemId { get; set; }

   [Required(ErrorMessage = "Please select a SAP Type.")]
   public SystemType SystemType { get; set; }

   [Required(ErrorMessage = "Please insert a Client for the Project.")]
```

```
    [MaxLength(3)]
    [RegularExpression("^[0-9]}$", ErrorMessage = "Client must be 3 digits.")]
    public string Client { get; set; }

    [Required(ErrorMessage = "Please insert the Language of the Project.")]
    [MaxLength(2)]
    [RegularExpression("^[a-zA-Z ]+$")]
    public string Language { get; set; }

    [System.Web.Mvc.HiddenInput(DisplayValue = false)]
    public long ID { get; set; }
}
```

The MaxLength annotation validates the length of the Name string, while the regular expression provides a white-list of allowed characters. If the values are invalid an error message is sent to the client. In the case of the Name attribute it was decided to allow only a very restricted set of characters as more variety was not necessary. DataAnnotations also allow to specify the Required annotation to refuse empty values for certain attributes.

However it is not always possible to restrict the set of character to use, for some fields such as descriptions, comments, etc., restrictions might result in usability problems for the users. In SoD App it was difficult to gauge the correct set of characters to allow for the description field, so it was chosen to not apply any during the validation process. However this removes an important level of protection, leaving the security of the input reliant only on the sanitization process. Therefore in such situations it is extremely important to analyse how will be the specific input used, in order to correctly sanitize it.

In the original version of the proposed class the MaxLength and RegularExpression annotations where not implemented. Looking at the ProjectModel class it is clear why the previous phase that concentrated on understanding the application is fundamental. Without knowledge of the context and meaning of the fields it would not be possible to appropriately define neither validation nor sanitization.

For ranged values such as integer, float, dates, etc., it should also be implemented a maximum and minimum value range check. In this case the Date check was performed by the Controller class, along with the actual validation and serialization, as soon as the client request is received:

```
public class ProjectController : Controllers
{
...
  [HttpPost]
  public ActionResult Create(ProjectModel proj)
  {
    try
    {
      if (proj.Date == null) proj.Date = DateTime.Now;

      if (!ModelState.IsValid || !validateDate(proj.Date) ||
          !validateSystemID(proj.SystemId)
          ||!validateLanguage(proj.Language))
      {
        return View("Denied");
      }

      PROJECT project = new PROJECT()
      {
        NAME = AntiXssEncoder.HtmlEncode(proj.Name, false),
        DESCRIPTION = AntiXssEncoder.HtmlEncode(proj.Description, false),
        CUSTOMER = AntiXssEncoder.HtmlEncode(proj.CustomerName, false),
        DATE = proj.Date,
        SYSTEM_ID = proj.SystemId,
        SAP_SYSTEM_TYPE = AntiXssEncoder.HtmlEncode(proj.SystemType.ToString(), false),
        CLIENT= AntiXssEncoder.HtmlEncode(proj.Client, false),
```

```
        LANGUAGE= AntiXssEncoder.HtmlEncode(proj.Language, false),
        USER_ID= _database.USERS.Where(u =>
            u.USERNAME.Equals(User.Identity.Name)).FirstOrDefault(),
        ID= GenerateProjID();
      };
    }
...
}
```

In SoD App the client request for a new Project was handled by the ActionResult class Create of the ProjectController, therefore this is where the validation and sanitization must be performed, as soon as the input is received. The ModelState.IsValid method validates the ProjectModel associated to the ProjectController by analysing the declared DataAnnotations and verifying if the restrictions are respected. Furthermore for enum types it also checks whether the value is one of the defined options. The additional validation functions validateDate(), validateSystemID and validateLanguage were manually implemented to verify the logical constraints and range constraints on the values. If the model passes validation the inputs contained are sanitized and the Project entity is created.

The sanitization is performed using the System.Web.Security.AntiXss.AntiXssEncoder class [24], that provides different types of encoders and uses a white-list approach. The correct encoder should be selected based on the usage of the input and purpose of the encoding. In this case the encoding was used to prevent XSS attacks therefore HtmlEncoder was used.

If a model different than MVC is used the same type of control can be achieved by manually implementing the check on regular expressions, enums, and type using TryParse or lookup values to assure that the data coming from the user is as expected.

In the analysis phase it has been confirmed that the Description field, CustomerName, Client and Date, are only meant to help the user and improve usability by providing useful information regarding the project. They are not key attributes used for database access. As a consequence these fields could not be exploited to perform SQL Injections, but could instead be exploited to perform XSS attacks and performing validation and sanitization against characters that are dangerous for HTML is sufficient to guarantee proper security.

Other attributes such as the SystemType, SystemID and Language have a functional role and are used during the analysis performed by SoD App on the ERP database, they are more likely to be vulnerable to SQL Injection and if not properly validated result in Exceptions being raised. It is therefore necessary to verify that they belong to an acceptable range of values in order to avoid errors. For SystemType, being an enum, the check is already performed by ModelState, however SystemID and Date are only validated syntactically, not semantically. Hence why it was necessary to implement specific functions for the purpose.

When using prepared statements to construct database query, they would be structured in a similar way:

```
string querySQL= "SELECT * FROM Users WHERE UserId = ?";
try {
  OleDbCommand command = new OleDbCommand(querySQL, connection);
  command.Parameters.Add(new OleDbParameter("@userId", txtUserId));
  OleDbDataReader reader = command.ExecuteReader();
catch{...}
```

The OLE DB.NET Framework Data Provider [25] was used to access the database. Instead of named parameters, positional parameters that are marked with a question mark "?" are used, therefore the parameters added to the Parameters collection must be properly ordered.

What was actually found during the analysis of SoDApp was a set of insecure methods managing database access, one of which is the following:

```
public IDataReader getReaderSelect(string table, string selectClause, string
    whereClause)
{
  string q = "SELECT " + selectClause + " FROM " + table + " WHERE " + whereClause;
  OleDbSQLiteCommand cmd = new OleDbCommand(q, sqlite);
  OleDbDataReader reader = cmd.ExecuteReader();
  return reader;
}
```

An example of code calling the method:

```
IDataReader reader= getReaderSelect("Projects", "*", "SystemID = " + project.SystemID
    );
```

Resulting in:

```
SELECT * FROM Projects WHERE SystemID = 123
```

This implementation clearly suffers from SQL Injection vulnerability. However securing this code is not as straightforward as securing the previous example, it will be necessary to reformulate the whole method, as part of the query is assembled by the getReaderSelect class but part is given as input. In this case it was necessary to distinguish between variables that were received from input and parts of the query that were manually defined by the developer and properly parameterise those from input.

```
public IDataReader getReaderSelect(string table, string selectClause, string
    whereClause, string whereInput)
{
  string q = "SELECT " + selectClause + " FROM " + table + " WHERE " + whereClause+ "
      ?";
  try{
    OleDbCommand cmd = new OleDbCommand(q, connection);
    command.Parameters.Add(new OleDbParameter(whereItem, whereInput));
    OleDbDataReader reader = command.ExecuteReader();
  }
  catch{...}
}

...

IDataReader reader= getReaderSelect("Projects", "*", "SystemID =" , project.SystemID
    );
```

For multiple parameters it is sufficient to state them with a question mark and add the values to the Parameters collection in order.

Sensitive fields such as User_ID and ID of the Project entity, that are used for identification and authorisation, should not be taken from user input or client-side scripts but should be retrieved or defined by the server. In the proposed code the User_ID is retrieved from the authenticated session and the project's ID is generated using a pseudo-random function, however in the original code the project's ID was retrieved from the received ProjectModel, therefore it could have been tampered by the user.

3.2.2 Managing File Uploads

Verify the format: In the following example an implementation of such control is provided, as it was performed for SoD App. The application has only two upload functions, the first

allows the user to load a matrix model for the database analysis and must be either .xls or .xlsx excel formats, the second is used by the user to load the database file itself in the format of an .sqlite file. The following example will analyse the first scenario, however the second is managed in the same fashion.

```
public class ExcelMatrixModel
{
  [Required(ErrorMessage = "Please insert a name for your matrix")]
  [RegularExpression(@"^[A-Za-z0-9_]+[A-Za-z0-9 _]*[A-Za-z0-9]+$", ErrorMessage
      = "Matrix's name cannot include special characters!")]
  [MaxLength(30)]
  public string Name { get; set; }

  [Required(ErrorMessage = "Please insert a description for your matrix")]
  [MaxLength(50)]
  [RegularExpression(@"^[A-Za-z0-9_]+[A-Za-z0-9 _,.!?]*[A-Za-z0-9.!?]+$",
      ErrorMessage = "Please insert a valid description. Special characters are
      not allowed.")]
  public string Description { get; set; }

  [Required(ErrorMessage = "Please select a template")]
  public Template Template { get; set; }

  [DataType(DataType.Upload)]
  [Required(ErrorMessage = "Please select a file")]
  [FileType("xls|xlsx", ErrorMessage = "Please select an Excel file with
      \".xls\" or \".xlsx\" extension.")]
  public HttpPostedFileBase ExcelFile { get; set; }
}

internal class FileType : ValidationAttribute
{
  private string[] _fileExtension;

  public FileType(string fileExtension)
  {
    _fileExtension = fileExtension.Split('|');
  }

  public override bool IsValid(object value)
  {
    // Avoid duplicating the DataAnnotation "[Required]"
    if (value == null)
    {
      return true;
    }

    //Get the declared file extension
    string postedFileName = ((HttpPostedFileBase)value).FileName.ToLower();
    string postedExtension = Path.GetExtension(postedFileName);

    //Get the actual mime type
    string mimeType=
        FileTypeResolver.GetMimeFromFileBase((HttpPostedFileBase)value);

    foreach (var item in _fileExtension)
    {
      if ((postedExtension).Equals(item) && (mimeType).Equals(item))
      {
        return true;
      }
```

```
    }
    return false;
  }
}

public class FileTypeResolver
{
  private static Dictionary<string, string> additionalMimeTypes = new
      Dictionary<string, string>()
  {
      { "504b0304","xlsx" },
      { "d0cf11e0a1b11ae1","xls" },
      { "53514c69746520666f726d6174203300","sqlite" },
  };

  [DllImport(@"urlmon.dll", CharSet = CharSet.Auto)]
  private extern static System.UInt32 FindMimeFromData(
  System.UInt32 pBC,
  [MarshalAs(UnmanagedType.LPStr)] System.String pwzUrl,
  [MarshalAs(UnmanagedType.LPArray)] byte[] pBuffer,
  System.UInt32 cbSize,
  [MarshalAs(UnmanagedType.LPStr)] System.String pwzMimeProposed,
  System.UInt32 dwMimeFlags,
  out System.UInt32 ppwzMimeOut,
  System.UInt32 dwReserverd
  );

  public static string GetMimeFromFileBase(HttpPostedFileBase uploadedFile)
  {
    try
    {
      byte[] buffer = new byte[256];

      byte[] fileBytes = new byte[uploadedFile.InputStream.Length];
      uploadedFile.InputStream.Read(fileBytes, 0,
          (int)uploadedFile.InputStream.Length);

      if(fileBytes.Length<256) return "unknown/unknown";

      for (int i = 0; i < 256; i++) {
        buffer[i] = fileBytes[i];
      }

      System.UInt32 mimetype;
      FindMimeFromData(0, null, buffer, 256, null, 0, out mimetype, 0);
      System.IntPtr mimeTypePtr = new IntPtr(mimetype);
      string mime = Marshal.PtrToStringUni(mimeTypePtr);
      Marshal.FreeCoTaskMem(mimeTypePtr);

      //xls and xlsx files are interpreted by FindMimeFromData as zip, whereas
          sqlite as binary
      if (string.IsNullOrEmpty(mime) ||
          mime.Equals("application/x-zip-compressed" ||
          mime.Equals("application/octet-stream"))
      {
        CheckAdditionalTypes(4);
        CheckAdditionalTypes(8);
        CheckAdditionalTypes(16);
      }
      return mime;
    }
    catch (Exception e)
    {
```

```
      return "unknown/unknown";
    }
  }

  //confronts the first bytes of the file with the signatures in the dictionary
  private void CheckAdditionalTypes(int signatureLength){
    byte[] buffer2 = new byte[signatureLength];
    for (int i = 0; i < signatureLength; i++)
    {
      buffer2[i] = buffer[i];
    }
    string str = Utilities.ByteArrayToHexString(buffer2);
    if (additionalMimeTypes.ContainsKey(str))
    {
      return additionalMimeTypes[str];
    }
  }
}

public class Utilities(){
  ...
    public static string ByteArrayToHexString(byte[] ba)
  {
    StringBuilder hex = new StringBuilder(ba.Length * 2);
    foreach (byte b in ba)
      hex.AppendFormat("{0:x2}", b);
    return hex.ToString();
  }
}
```

The ExcelMatrixModel class models the user's file input, and has the same functions already described for the ProjectModel, validating all the fields submitted by the user. In order to validate the file a custom ValidationAttribute is created and named FileType. ValidationAttribute is the base class for all attributes used by DataAnnotations, hence when ModelState.IsValid will be called by the controller the code implemented in FileType will be executed.

The FileType attribute performs a first check on the extension declared by the filename, confronting it with the allowed ones. The used function Path.GetExtension() retrieves the extension after the last occurrence of the dot character, this is important in order to manage evasion techniques such as providing "image.jpg.php" as filename. If different methods are used it is advised to verify that the extension retrieving function does not return the first occurrence.

As previously explained, however, this check is not sufficient as a malicious user could have easily changed the extension in the file name, hence a second check is performed using the GetMimeFromFileBase() method of the FileTypeResolver class.

Multipurpose Internet Mail Extensions (MIME) is an Internet standard that extends the format of email to support various functions, however the content types defined by MIME standards are used also by HTTP to describe content types.

The GetMimeFromFileBase() method takes inspiration from Internet Explorer's MIME type detection implementation [26]. Using the FindMimeFromData() method, implemented by the Urlmon library, the first 256 bytes of the file are analysed to define the MIME type. However this method contains hard-coded tests for currently only 26 separate MIME types, therefore the developer should verify that the used file types are supported, otherwise the result is unexpected. In this specific case it was later discovered that none of the required formats were supported, therefore it was necessary to manually provide the signatures for the acceptable files by implementing the CheckAdditionalType() method. The FindMimeFromData was however retained for future file upload features.

If this solution is adopted, when manually adding new MIME types it is necessary to verify all the possible signatures for a certain file extension.

Check/Change the name: The following example shows the file name management implemented in SoD App.

```
[HttpPost]
public ActionResult CreateFromExcel(ExcelMatrixModel excelModel)
{
  if (!ModelState.IsValid)
  {
    ViewBag.Templates = templates;
    return View(excelModel);
  }
  try
  {
    //encode the user provided name to secure from XSS
    excelModel.Name = AntiXssEncoder.HtmlEncode(excelModel.Name, false);
    excelModel.Description = AntiXssEncoder.HtmlEncode(excelModel.Description,
        false);

    //validate template
    IExcelImport e = null;
    switch (excelModel.Template)
    {
      case Template.LINEAR:
        e = new ExcelLinearImportService();
        break;
      case Template.MATRIX:
        e = new ExcelMatrixImportService();
        break;
      default:
        return View("Denied");
    }

    //generate a random hexadecimal string
    RNGCryptoServiceProvider rngProvider = new RNGCryptoServiceProvider();
    var byteArray = new byte[4];
    rngProvider.GetBytes(byteArray);
    string serverFilename = Utilities.ByteArrayToHexString(byteArray);

    //compose the path using the random string and store file
    string extension= Path.GetExtension(excelModel.ExcelFile.Filename);
    string path = Path.Combine(App_Data, serverFilename+"_"+
        DateTime.Now.ToString()+"_"+this.User.Identity+ extension);
    excelModel.ExcelFile.SaveAs(path);

    MatrixService user_mService = new MatrixService("Matrices",
        User.Identity.Name);
    Matrix m = e.Import(path, excelModel.Name, excelModel.Description);
    user_mService.Create(m);

    TempData["Message"] = "Matrix successfully created!";

  }
  catch (Exception e)
  {
    ...
  }
  return RedirectToAction("Details");
}
```

In this example the file is retrieved from database and displayed using the user provided name, however the actual save path is created using a random number generator that makes the filename hard to predict.

In the original version of the proposed code, in addition to the lack of renaming, the file was saved before validating the Template, and was inserted in the database afterwards. If the validation failed the file was not correctly removed, therefore it remained in the App_Data folder occupying space but the web application would have not retained any reference to it, leaving a ghost file in the file system.

It is always necessary to verify that all check are performed before saving the file and that the file, or temporary versions of it, are correctly removed if any exception arises.

Verify the size of the file: ASP.NET automatically implements a maximum request length, set by default at 4MB [27]. The default value can be edited by modifying the following lines in the App Web.config:

```
<configuration>
  <system.web>
    <httpRuntime maxRequestLength="2048" timeout="3600" /> <!--sets the maximum
        at 2MB-->
  </system.web>
</configuration>
```

If using IIS version 7+ in order for the change to be effective it is also necessary to edit the maxAllowedContentLength [28] property:

```
<system.webServer>
  <security>
    <requestFiltering>
      <requestLimits maxAllowedContentLength="2048" />
    </requestFiltering>
  </security>
</system.webServer>
```

Both properties are defined as Int32, therefore the maximum upload size allowable by ASP.NET is 4GB. Additionally it is possible to specify the timeout for the request to be completely received. This can be a useful setting against DoS attacks that aim at keeping the server busy with extremely slow clients.

Limit the quantity of data a user can store: The following piece of C# code allows to verify the size of a Session object.

```
long totalSessionBytes = 0;
BinaryFormatter b = new BinaryFormatter();
MemoryStream m;
foreach(var obj in Session)
{
  m = new MemoryStream();
  b.Serialize(m, obj);
  totalSessionBytes += m.Length;
}
```

Examine file content: There are several approaches possible to inspect a Zip file:

- Using System.IO.Compression.ZipArchive Class [29]: this class allows to inspect the elements contained in the file without decompressing the .zip.

```
[HttpPost]
public ActionResult UploadZip(HttpPostedFileBase zipFile)
{
  ...
  using (ZipArchive archive = new ZipArchive(zipFile.InputStream))
  {
    long predictedTotSize=0;
    foreach (ZipArchiveEntry entry in archive.Entries)
    {
```

```
      //if it is not a folder
      if (!string.IsNullOrEmpty(Path.GetExtension(entry.FullName)))
      {
        //deny Zip files if they contain nested Zip files
        if (Path.GetExtension(entry.FullName) == ".zip")
        {
          //file not accepted
        }
        predictedTotSize+=entry.Length;
      }

      //check the path of zipped files to spot directory traversal attempts
      if (entry.FullName.Contains(".."))
      {
        //file not accepted
        //report suspicious activity
      }
    }

    //deny if the decompressed length prevision is greater than a set value,
        for example 4GB
    if (predictedTotSize > 4294967296)
    {
         //file not accepted
    }
  }
  ...
}
```

The ZipArchive object represents the package of compressed files and the ZipArchiveEntry objects are the compressed files and folders within a zip archive. The Entries collection retains all the files and folders in the archive three, including nested nodes.

By cycling through the collection it is possible to evaluate the predicted sizes for the contained files using the Length attribute and the target paths and extensions using the FullName attribute.

In the provided example recursive application of the compression algorithm is not allowed, however it is possible to allow a couple of levels of recursion if necessary. In this case the .zip entry will have to be decompressed and it will be possible to perform the same checks already described. A single entry can be decompressed without decompressing the whole file using the ZipArchiveEntry.ExtractToFile() method.

- Using a controlled process: this is the most reliable method and consists in creating a process with the specific task of decompressing the uploaded file. However this process should have a hard limit of memory it can use, if the extraction exceeds the limit an exception is raised, and the file is refused. This option however has the drawback that the bigger is the allowed data size the more resource intensive it is. For large file this control could be easily exploited to launch DoS attacks on the server by only uploading many files. Therefore this approach is viable if small files are allowed and the amount of memory granted to the process is very limited.
- Using an antivirus: if the contents of uploaded files are analysed by an antivirus it should already perform these verification.

Secure the save path: Applying this modification simply requires to change the target directory for saving uploaded files to be outside of root.

Check correctness of Content-Type header when serving the file: Before serving an uploaded file to the user it should be made sure that the Content-Type HTTP header contains the expected value.

```
public ActionResult Download(string requestedFile)
```

```
{
  try
  {
    //verify authorisation
    //get MIME type
    //get file
    return File(fileBytes, MIME_FileType, fileName);
  }
  catch (Exception e)
  {
    //...
  }
}
```

Management of variables storing the input

In .NET it is possible to use the SecureString class [30] to store sensitive information. This class offers three interesting features:

- Implements the IDisposable interface: instances of the System.String class are both immutable and, when no longer needed, cannot be programmatically scheduled for garbage collection, however the SecureString can be cleaned using the Clean() method and immediately disposed with Dispose() thanks to the implementation of the System.IDisposable interface. If using a String the only solution is to overwrite the value of the variable, however because String instances are immutable, operations that appear to modify an existing instance actually create a copy of it to manipulate. These additional copies remain in memory with the sensitive data until they are managed by the garbage collector, and the developer has no power over them.
- Encryption: instances of the class use the protection mechanism provided by the underlying operating system. This means that on Windows it's contents are guaranteed to be stored in RAM encrypted but on other platforms encryption might be unavailable. However even if the SecureString implementation is able to take advantage of encryption, the plain text assigned to the SecureString instance is be exposed at various times: whenever the value of the secure string is modified by methods such as AppendChar or RemoveAt, it must be decrypted, modified, and then encrypted again. As a result the time interval for which the SecureString instance's value is exposed is merely shortened in comparison to a String instance.
- Memory pinning: when using String, because its memory is not pinned, the garbage collector will make additional copies of String values when moving and compacting memory, that will add to the copies already created when modifying the value. In contrast, the memory allocated to a SecureString object is pinned and when removed it guarantees to leave no copies behind.

3.3 Authentication and Session Management

3.3.1 User ID

The following example shows the controls implemented in SoD App for username and email validation:

```
public class UserModel
{
  [Required(ErrorMessage = "Please insert a valid Username!")]
```

```
  [MaxLength(40)]
  [RegularExpression("^[A-Za-z0-9_]+[A-Za-z0-9]+$", ErrorMessage = "Please insert a
      valid Username, symbols are not allowed.")]
  public string USERNAME { get; set; }

  [Required(ErrorMessage = "Please insert an E-Mail!")]
  [MaxLength(330)]
  [EmailAddress(ErrorMessage = "Please insert a valid E-Mail")]
  public string E_MAIL { get; set; }

 //...

  internal class Username: ValidationAttribute
  {
    public string InvalidMessage = "Please insert a valid username, symbols are not
        allowed.";
    public string RequireMessage = "Please insert a username.";
    public int MaxLength = 40;

    protected override ValidationResult IsValid(object value, ValidationContext
        validationContext)
    {
      var input = (string)value;
      //verify that it is not empty
      if(string.IsNullOrEmpty(input))
        return new ValidationResult(RequireMessage);
      //verify that it is valid
      Regex r = new Regex("^[A-Za-z0-9_]+[A-Za-z0-9]+$");
      bool match=r.IsMatch(input);
      if (!match || input.Length > MaxLength)
        return new ValidationResult(InvalidMessage);
      //verify that it is available
      try
      {
        using(var _database = new SOD_APPLICATION()){
          USER user = _database.USERS.Where(u =>
              u.USERNAME.Equals(input)).FirstOrDefault();
          if (user == null)
            return ValidationResult.Success;
          else
            return new ValidationResult(ErrorMessage);
        }
      }
      catch
      {
        return new ValidationResult(ErrorMessage);
      }
    }
  }
}
```

SoD App does not require high levels of security therefore it was chosen to use as identifier the username provided by the user. In the proposed code during the validation process the web application verifies also that the database does not already contain the proposed username. It is important however that, before the value is used for database access, it has been properly validated. When using Data Annotations, if more attributes are defined (such as Required, MaxLength, etc.) the order in which the validation of the attributes will be executed is unknown, therefore the proper way to assure that the input is valid is to perform verification directly into the custom ValidationAttribute class, before accessing the database.

In this case the System.Text.RegularExpressions.Regex class has been used to implement the regular expression.

For the email validation the EmailAddress ValidationAttribute provided by the Data Annotations namespace was used, the verfication of legitimacy is performed by the controller class:

```
[HttpPost]
public ActionResult Create(UserModel user)
{
  if (!ModelState.IsValid)
      return View(user);

  try
  {
    RNGCryptoServiceProvider rngProvider = new RNGCryptoServiceProvider();
    var byteArray = new byte[20];
    rngProvider.GetBytes(byteArray);
    string activationToken = Utils.ByteArrayToString(byteArray);

    bool sent= UserModel.SendVerificationMail(newUser.EMAIL, "activationURL",
        activationToken);
    if(!sent){
      TempData["Message"] = "An error occured while sending the verification email.
          Please contact support.";
      return View(user);
    }

    USER newUser = new USER();
    newUser.USERNAME = AntiXssEncoder.HtmlEncode(user.USERNAME, false);
    newUser.EMAIL = AntiXssEncoder.HtmlEncode(user.EMAIL,false);
    newUser.VALID = ValidityState.inactive+"|" + DateTime.Now.ToString("yyyyMMdd")
        +"|" +activationToken;

    //...

    TempData["Message"] = "User succesfully created!";

  } catch (Exception e){
      //...
  }
  return RedirectToAction("Index");
}
```

In the proposed example a unique token is generated, and saved in the VALID string attribute of USER, along with the time of generation and validity value. The ValidityState is an enum that offers three states: *inactive* if user has not validated the email, *expired* if the password has expired and *active*. When the user activates the account trough the link provided the activation function checks whether the token has expired, counting 20 minutes from the moment of generation, and if it is still valid the user state transitions from ValidityState.inactive to ValidityState.active. The VALID attribute will retain the date of creation, that will be used for password expiration, while the token will be removed.

The System.Security.Cryptography.RNGCryptoServiceProvider [31] class is used in place of the standard System.Random class for the purpose of generating the activation token. Random relies on the computer system clock, therefore if multiple objects were created at the exact same time, they could contain the same sequence of random numbers. For standard applications this eventuality is remote enough to be acceptable, however for security applications more reliable algorithms could be necessary.

Secure random number generators, such as RNGCryptoServiceProvider, guarantee a lower collision rate and are important when creating a very high volume of numbers or to ensure proper uniqueness over a long period of time. In the provided example, if an attacker managed to learn the logic behind the creation of file names to be stored on the server, he could easily predict the values for the date and username but it would be significantly more difficult to predict the random part of the filename.

3.3.2 Session Management

With respect to the evaluated session vulnerabilities the following implementations are proposed:

- **Session name**:

  For SoD App in order to change the ASP.NET_SessionId name the following lines were added to the App Web.config file:

```
<configuration>
  <system.web>
    <sessionState cookieName="ID" />
  </system.web>
</configuration>
```

  For different frameworks there might be also a specific configuration file to edit or it might have to be set programmatically.

- **Token length**:

  SoD App uses an ASP.NET_SessionId that has maximum length set to 120. Currently it is considered sufficient thanks to the entropy of the token so no modifications where applied [14]. There is no official support for longer tokens therefore in this situation if a higher level of security is required a different solution should be designed.

- **Token value**: If using the ASP.NET_SessionId the value is already generated randomly, otherwise the System.Security.Cryptography.RNGCryptoServiceProvider class [31] could be used to generate the token:

```
//generate a random hexadecimal string
RNGCryptoServiceProvider rngProvider = new RNGCryptoServiceProvider();
var byteArray = new byte[4];
rngProvider.GetBytes(byteArray);
string token = Utilities.ByteArrayToHexString(byteArray);
```

- **Session Implementation**:

  In .NET Framework in order to renew the session the "Session.Abandon()" and "Response.Cookies.Add()" methods are used. It is essential ti verify that all older sessions are correctly terminated before generating a new session ID, otherwise an attacker could exploit old session IDs to access an authenticated session. This is especially dangerous if the website allows also HTTP sessions.

  It is best to use already existing implementations and libraries offered by the web development framework used rather than implementing home-made solutions to manage session tokens. These libraries are thoroughly tested and maintained and are likely to be less exploitable. It is very important however to always use the latest libraries as older ones have well known vulnerabilities and bugs that would introduce significant risks. In addition, as already highlighted when discussing the name of the token, when using well known implementations it is good practice to modify some default values in order to make it more difficult for a cracker to guess the values used by the web application.

- **Cookies Implementation**: The offered attributes can be set using the HttpCookie class [32] either programmatically or using the App Web.config file. With the first approach it is possible to set attributes only for specific cookies whereas with the second approach the attribute is set for all cookies. Examples will be provided for both approaches when using .NET Framework equal or superior to 4.7.2.

  Secure attribute: This attribute can be implemented by adding the following lines to App Web.config:

```
<configuration>
  <system.web>
    <httpCookies requireSSL="true" />
  <system.web>
</configuration>
```

However if there is a <forms> element in the <system.web>/<authentication> block then the default setting of false will override the previously added line so it should be edited as follows:

```
<system.web>
  <authentication mode="Forms">
    <forms requireSSL="true" />
  </authentication>
</system.web>
```

HttpOnly attribute: In .NET the HttpOnly attribute is set by default for Session ID and Forms Authentication cookie, however it is possible to protect all custom application cookies by adding the following line to the App Web.config file:

```
<system.web>
  <httpCookies requireSSL="true" httpOnlyCookies="true" >
<system.web>
```

The programmatical approach:

```
HttpCookie customCookie = new HttpCookie("customCookie");
customCookie.HttpOnly = true;
Response.AppendCookie(customCookie);
```

SameSite attribute: Usually the default value is lax, however in some frameworks it might be set to none, disabling the use of the attribute, so it would be best to explicitly set it to the required value.

As for the other attributes this one can also be set through the App Web.config file:

```
<configuration>
  <system.web>
    <httpCookies ... sameSite="strict" />
  </system.web>
</configuration>
```

Domain attribute: Implementing this attribute however introduces vulnerability and is therefore not advised.

For .NET Framework the programmatical approach is:

```
HttpCookie customCookie = new HttpCookie("customCookie");
customCookie.Domain = ".domain.com";
```

or using the App Web.config file:

```
<system.web>
  <httpCookies ... domain=".domain.com"/>
</system.web>
```

Path attribute: In order to set the attribute in the App Web.config file:

```
<system.web>
  <httpCookies ... path="/home"/>
</system.web>
```

Expire and Max-Age attributes: Expire and Max-Age must not be set for session cookies.

All these cookie attributes, however, are not supported by all browsers. In addition not even all web servers support them, even if properly configured. It is therefore advised to always test these flags after having implemented them, in order to verify that they are correctly incorporated in the server responses.

Sometimes the web server could ignore the cookie setting contained in the App Web.config file. In this case the flags will have to be set through code. In case the attributes are ignored altogether it is still possible to implement a work-around by creating mechanisms that append the attributes directly to the Set-Cookie HTTP response header, such as for instance redirection rules. As an example, adding the SameSite attribute can be implemented by adding the following rule to the IIS Web.config file:

```
<rewrite>
  <outboundRules>
    ...
    <rule name="Add SameSite" preCondition="No SameSite">
      <match serverVariable="RESPONSE_Set_Cookie" pattern=".*" negate="false" />
      <action type="Rewrite" value="{R:0};SameSite=strict" />
      <conditions>
      </conditions>
    </rule>
    <preConditions>
      <preCondition name="No SameSite">
        <add input="{RESPONSE_Set_Cookie}" pattern="." />
        <add input="{RESPONSE_Set_Cookie}" pattern=";SameSite=strict"
           negate="true" />
      </preCondition>
    </preConditions>
  </outboundRules>
</rewrite>
```

- **Session Expiration:**

  Using .NET an idle timeout of 5 minutes can be set in the App Web.config file, using the HttpSessionState class [33], adding the following lines:

```
<configuration>
  <system.web>
    <sessionState mode="InProc" timeout="5" />
  </system.web>
</configuration>
```

  When a session times out, the Abandon method is called, or the ASP.NET application is shut down, and the session End event is raised if the mode is set to InProc. The End method can be used to perform the proper invalidation of the ID.

  The other two types of timeout do not have a default implementation and should be defined manually by the developer.

  For SoD App the Absolute timeout was implemented by modifying two methods in the Global.asax file: Session_Start and Application_AcquireRequestState. In the Session_Start method a session value is set on the HttpSessionState object, containing the absolute expiry time calculated adding the chosen timeframe of 30 minutes to the current time. Then in the Application_AcquireRequestState each time a request is received if the current time is later than the expiry time set then the ID is invalidated, the session terminated and the user is redirected to the login page.

```
void Session_Start(Object sender, EventArgs e)
{
  Session["AbsoluteExpiryTimeout"] = DateTime.Now.AddMinutes(30);
}
```

```
void Application_AcquireRequestState(object sender, EventArgs e)
{
  if (HttpContext.Current.Session != null)
  {
    DateTime expiry = (DateTime)base.Context.Session["AbsoluteExpiryTimeout"];

    if (expiry <= DateTime.Now)
   {
      Session["ID"]= null ;
      Session.Abandon();
      //redirect to login page
    }
  }
}
```

3.4 Password management and Policies

3.4.1 Password Strength

In SoD App, in order to define what was the an acceptable minimum complexity requirement, both the pool of users and the application purpose had to be considered.

During the Black Box analysis phase it was defined that the target of the application were the employees of the company in charge of performing auditing operations for the clients of the company, and that it would soon be extended to include client employees, with a similar competence profile. Since the employees of the company were colleagues it was easy to asses their level of familiarity with computers and management of multiple accounts. It was highlighted that the majority of the users where young, used the computer as part of their job, and were accustomed to managing many different accounts in their daily routine, therefore it was evaluated that they should be able to handle correctly a medium complexity password.

Concerning the application, it was evaluated that since it's purpose was to manage databases containing sensitive information regarding structure and roles of the clients it was necessary to require at least medium level of security.

Based on these evaluations it was chosen to opt for a minimum length of 12 characters, coupled with high entropy, applying all of the rules previously described.

```
public class UserModel
{
  //...

  [MaxLength(160)]
  [MinLength(12)]
  [Required(ErrorMessage = "Please insert a password")]
  [RegularExpression("^((?=.*?[A-Z]) (?=.*?[a-z]) (?=.*?[0-9])
      (?=.*?[!£$%&?^@§#])).{12,}$" , ErrorMessage = "Passwords must be at least 12
      characters long and contain one for each of the following character sets: upper
      case (A-Z), lower case (a-z), number (0-9), symbol (!£$%&?^@§#)")]
  public SecureString PASSWORD { get; set; }

  //...
}
```

3.4.2 Password Storage

The following portion of code is the continuation of the method that manages the creation of a new user, and was previously proposed for the validation of user ID and email 3.3.1.The model

used for validation is displayed in the above code snippet, whereas this portion of code implements the validation and storage of the password:

```
[HttpPost]
public ActionResult Create(UserModel user)
{
  if (!ModelState.IsValid)
  {
    return View(user);
  }
  try
  {
    //... ID and email validation

    //generate salt:
    var salt = new byte[64];

    rngProvider.GetBytes(salt);

    var pwdBytes = Encoding.ASCII.GetBytes(user.PASSWORD.ToString());

    user.PASSWORD.Clear();
    user.PASSWORD.Dispose();

    var hash = Hash(salt, pwdBytes);

    newUser.PASSWORD = salt+hashString;

    //...additional validation

    _database.USERS.Add(newUser);
    _database.SaveChanges();

    TempData["Message"] = "User succesfully created!";

  } catch (Exception e)
  {
    //...
  }
  return RedirectToAction("Index");
}

public string Hash(byte[] salt, byte[] pwd){
  //set up Argon2 hasher
    Argon2i argon2hasher = new Argon2i(pwd);
    argon2hasher.Iterations = 15;
    argon2hasher.DegreeOfParallelism = Environment.ProcessorCount;
    argon2hasher.MemorySize = 65536; //number of 1kB blocks used while processing the
        hash
    argon2hasher.Salt = salt;

    var hash = argon2hasher.GetBytes(64);
    string hashString = string.Empty;
    foreach (byte b in hash)
    {
      hashString += String.Format("{0:x2}", b);
    }
    return hashString;
}
```

In the proposed example first the Salt was generated using the already described RNGCryptoServiceProvider class. In the specific case of SoD App a restricted pool of users and the storage capacities permitted the use of long Salts therefore it was decided to use a 64 bytes Salt, as it

provides high enough collision resistance with respect to the pool of generated Salts.

After generating the Salt the password was hashed using the Argon2 [34] implementation for .NET provided by the Konscious.Security.Cryptography library[35]. Simply hashing the password and Salt with a hashing function does not provide the same level of security that a key derivation function provides.

Among the available options the most relevant functions were:

Argon2 : winner of the Password Hashing Competition in July 2015, uses BLAKE2 hash function.

Scrypt : published in 2009, uses PBKDF2 and HMAC-SHA256.

PBKDF2 : published in 2000, uses HMAC-SHA1

Bcrypt : published in 1999, based on the Blowfish cipher.

It was decided to use Argon2 as it resolves vulnerabilities found in the other functions and uses a stronger hashing algorithm, however since it is recent, and therefore less tested, the tester must be aware that this implementation is at higher risk of new exploitation being found, hence requires more frequent monitoring of the cryptoanalysis state.

There were three main .NET implementation available for Argon2, however only the selected library provided the possibility to set the number of iterations of the function, the other libraries based their implementation on the time cost of the function. In other words it was only possible to define the duration of execution of the function and not the actual iterations. This is not a reliable approach, as the number of iterations for a specific password depends on the current workload of the server. An attacker could exploit this by launching a DoS attack to slow the server which would result in hashes being computed differently throughout the duration of the attack.

There are three versions of the Argon2 function: Argon2i, Argon2d and Argon2id.

> "Argon2 has one primary variant: Argon2id, and two supplementary variants: Argon2d and Argon2i. Argon2d uses data-depending memory access, which makes it suitable for cryptocurrencies and proof-of-work applications with no threats from side-channel timing attacks. Argon2i uses data-independent memory access, which is preferred for password hashing and password-based key derivation. Argon2id works as Argon2i for the first half of the first iteration over the memory, and as Argon2d for the rest, thus providing both side-channel attack protection and brute-force cost savings due to time-memory tradeoffs. Argon2i makes more passes over the memory to protect from tradeoff attacks."
>
> A. Biryukov, D. Dinu, D. Khovratovich, Internet-Draft, "The memory-hard Argon2 password hash and proof-of-work function", 03 August 2017[36].

Therefore the preferential choice would have been Argon2id, however none of the previously cited libraries provided an implementation for it. Therefore the Argon2i variant was selected, as more appropriate for password hashing.

The logic behind key derivation functions is to make the computation of the hash more resource consuming than using directly the hashing functions, therefore imposing the same workload on at attacker trying to brute-force the hash. Argon2 implements this logic using 3 main parameters:

- Iterations count: the number of iterations has a direct influence on the time cost.
- Memory size: the memory used while processing the hash, affecting the memory cost.
- Level of parallelism: how many threads will be used during the process, affecting memory and CPU cost.

Higher values of these parameters result in hashes more complex to brute-force, however they reflect directly also on the web server's workload. As already mentioned the SoD deals with a restricted pool of users that id addition use the application infrequently, therefore it was possible to select higher parameter values for the Argon2 implementation.

The iteration count of 15 was chosen based on recent demonstrations that if using less than 10 iterations the hash can be computed by an algorithm which has complexity that is independent from the chosen parameters [37].

The length of the final hash has been set to 64 bytes to mitigate risk of collision.

The following formula was used for password storage, as per OWASP recommendations [38], since it was deemed sufficient:

$$[\text{protected form}] = [\text{salt}] + \text{protect}([\text{protection func}], [\text{salt}] + [\text{credential}]);$$

However the Argon2 function allows to use additional data to increase the complexity of the algorithm.

Concerning the variable management, from the code snippet in the previous section, reporting the UserModel, it can be seen that the user input is stored in a SecureString object, however from Create method code it is clear that the protection offered by the SecureString is very limited, as it has to be converted to bytes for the Argon2 hasher. Nevertheless it still offers an improvement and is easy to implement, therefore it is advised to take advantage of the SecureString class.

3.4.3 Password Recovery

The implementation is similar to that of email validation 3.3.1, generating a unique token sent over email to allow the user to reset the password. Also in this case an expiration time has been set.

3.4.4 Password Policies

For SoD App the password expiration was set at three months, an acceptable timeframe both for the security and usability requirements.

Upon login the web application extracts the creation date information from the USER.VALID field and checks whether the date is older than the allowed period. A week before the expiration time the application notifies the user in the home page that the password must be renewed within a week. When the expiration date is reached the user will be forced to provide a new password to be able to continue using the application.

```
public ActionResult Login(UserModel model, string returnUrl)
{
  //..authenticate
  using(var _database = new KSOD_APPLICATION()){
    USER user = _database.USERS.Where(u => u.USERNAME.Equals(input)).FirstOrDefault();

        string[] items= user.VALIS.Split('|');
        switch(items[0]){
          case ValidityState.inactive.ToString():
            //activation token management
                break;
          case ValidityState.expired.ToString():
            TempData["Message"] = "Your password is expired, please set a new one.";
                return RedirectToAction("Renew Password");
                break;
          case ValidityState.valid.ToString():
```

```
            DateTime creationDate = DateTime.ParseExact(items[1], "yyyyMMdd",
                System.Globalization.CultureInfo.InvariantCulture);
            var elapsedTime= (DateTime.Now - creationDate).TotalDays;
                if(elapsedTime > 83){
                  TempData["Message"] = "Your password will expire soon, please set a
                      new one.";
                  return RedirectToAction(returnUrl);
                }else if(elapsedTime>90){
                  TempData["Message"] = "Your password is expired, please set a new
                      one.";
                  user.VALID=ValidityState.expired.ToString()+"|"+items[1]+"|"+
                      items[2];
                  _database.SaveChanges();
                  return RedirectToAction("Renew Password");
                }
                break;
        }
    }
    //..
}
```

Upon password renewal the history of old hashes is retrieved and verified against the new password, then the new Salt+hash string is updated along with the date in the USER.VALID field. In addition the old Salt+hash value is added to the history database.

```
[HttpPost]
public ActionResult Edit(UserModel user)
{
  if (!ModelState.IsValid)
  {
    return View(user);
  }
  try
  {
    //...

    USER_PASSWORD_HISTORY history=_database.PASSWORD_HISTORY.Find(User.Identity.Name);

    var hash= String.Empty;

    foreach(string h in history.HASHES){
      string[] items= history.Split('|');
      hash=Hash( Encoding.ASCII.GetBytes(items[0]),
                Encoding.ASCII.GetBytes(user.PASSWORD.ToString()) );
      if(hash.Equals(items[1])){
        TempData["Message"] ="The password has already been used, please chose a
            different password";
        return View();
      }
    }
    //generate salt:
    var salt = new byte[64];
    rngProvider.GetBytes(salt);

   //compute hash
    var hash = Hash(salt, Encoding.ASCII.GetBytes(user.PASSWORD.ToString()));
    user.PASSWORD.Clear();
    user.PASSWORD.Dispose();

    newUser.PASSWORD = salt+hash;
```

```
    //...additional validation

    _database.USERS.Add(newUser);
    _database.SaveChanges();

    TempData["Message"] = "User succesfully created!";

  } catch (Exception e)
  {
    //...
  }
  return RedirectToAction("Index");
}

public partial class USER_PASSWORD_HISTORY
{
  [DataMember]
  [JsonRequired]
  public string USERNAME { get; set; }

  [DataMember]
  [JsonRequired]
  public string[] HASHES { get; set; } //format: salt | hash | date_of_expiration
}
```

3.4.5 Alternative Authentication Mechanisms

In SoD App for administrator type of accounts a two factor authentication was implemented using Google Authenticator. For the purpose the a .NET GoogleAuthenticator library [39] was used to manage the token verification. The version proposed below uses the library, however it is possible to download the project sample provided by the author, that contains the source code used by the library. This allows to use the provided code directly and customise it as needed.

```
private FileContentResult GenerateQRCode(string userIdentifier, string
    accountSecretKey)
{
  TwoFactorAuthenticator tfA = new TwoFactorAuthenticator();
  var setupCode = tfA.GenerateSetupCode("SoD App", userIdentifier, accountSecretKey,
      300, 300);

  WebClient wc = new WebClient();
  MemoryStream ms = new MemoryStream(wc.DownloadData(setupCode.QrCodeSetupImageUrl));
  var bytes= ms.ToArray();

  return new FileContentResult(bytes, "image/png");
}
```

The TwoFactorAuthentication class manages the creation of the QR code that can be scanned by the user using the android Google Authenticator App to instantiate a token generator in the android App that is synchronised with the website.

The GenerateSetupCode method generates the QR code using five parameters:

- Issuer: describes the application to which the authenticator is synchronised.
- User identifier: this should be a unique user identifier such as the user ID, the email, or the username (if it's unique). It will be displayed in the android App to help the user identify which account is connected to the token generator. It must contain no spaces.

- Account Secret Key: it is a secret key used for the token generation by both authenticator and application. It should be different for each credential and the application should store it encrypted in the database as it is the only secret element of the token generation.
- QR image height.
- QR image width.

The SetUpCode object also provides an entry key to use in the android application, as an alternative to the QR code, to instantiate the authenticator.

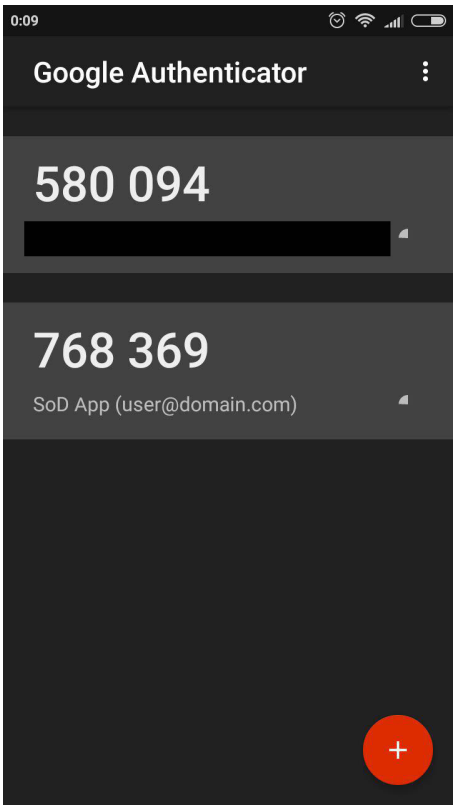

Figure 3.2. Google Authenticator android view

In order to verify the token provided by the user the following function is used:

```
private bool VerifyToken(string accountSecretKey, string providedCode)
{
 TwoFactorAuthenticator tfA = new TwoFactorAuthenticator();
 return result = tfA.ValidateTwoFactorPIN(accountSecretKey, providedCode);
}
```

The ValidateTwoFactorPIN method generates a set of tokens, allowing a backwards tolerance of 30 seconds, and if the provided token is among the generated tokens the authentication is successful.

3.5 Data Management

The following example shows the classes implementing encryption and decryption of files in SoD App, using the System.Security.Cryptography.Aes class:

```
static bool AESEncryptStream(Stream plainText, byte[] Key, byte[] IV, string savePath)
{
  // Check arguments.
  if (plainText == null || plainText.Length <= 0)
      throw new ArgumentNullException("plainText");
  if (Key == null || Key.Length <= 0)
      throw new ArgumentNullException("Key");
  if (IV == null || IV.Length <= 0)
      throw new ArgumentNullException("IV");

  // Create an Aes object
  // with the specified key and IV.
  using (Aes aesAlg = Aes.Create())
  {
    aesAlg.Key = Key;
    aesAlg.IV = IV;

    // Create an encryptor to perform the stream transform.
    ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);

    try{
      using (FileStream encrypted = File.Open(savePath, FileMode.Create,
          FileAccess.Write, FileShare.None))
      {
        using (CryptoStream cs = new CryptoStream(encrypted, encryptor,
            CryptoStreamMode.Write))
        {
          plainText.CopyTo(cs);
        }
      }
      return true;
    }
    catch{
      //..
      return false;
    }
  }
}

static void AESDecryptFile(MemoryStream outputStream, string encryptedFilePath,
    byte[] Key, byte[] IV)
{
  // Check arguments.
  if (encryptedFilePath == null || encryptedFilePath.Length <= 0)
    throw new ArgumentNullException("file path");
  if (Key == null || Key.Length <= 0)
   throw new ArgumentNullException("Key");
  if (IV == null || IV.Length <= 0)
    throw new ArgumentNullException("IV");


  // Create an Aes object
  // with the specified key and IV.
  using (Aes aesAlg = Aes.Create())
  {
    aesAlg.Key = Key;
    aesAlg.IV = IV;

    // Create a decryptor to perform the stream transform.
    ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
```

```
    // Create the streams used for decryption.
    using (FileStream encrypted = File.Open(encryptedFilePath, FileMode.Open))
    {
      using (CryptoStream cs = new CryptoStream(encrypted, decryptor,
          CryptoStreamMode.Read))
      {
        int read;
        //16 kB at a time are processed
        byte[] buffer = new byte[16*1024];

        try
        {
          while ((read = cs.Read(buffer, 0, buffer.Length)) > 0)
          {
            outputStream.Write(buffer, 0, read);
          }
        }
        catch (CryptographicException ex_CryptographicException)
        {
          outputStream=null;
          //..
        }
      }
    }
  }
}
```

The Aes class inherits from System.Security.Cryptography.SymmetricAlgorithm class, and allows to encrypt and decrypt data using the Advanced Encryption Standard (AES) algorithm. There are four other implementations of symmetric algorithms: DES, RC2, Rijndael and TripleDES. The best choice in terms of strength is AES, however also TripleDES and Rijndael are acceptable choices, the others should be avoided. The difference between AES and Rijndael implementations is that the first one conforms to FIPS-197 specifications for AES, while the other implements the original Rijndael version, that provides some additional options. If choosing the Rijndael implementation it must be kept in mind however that this choice could result in interoperability issues with other AES implementations compliant to the standard.

The Create() method creates an object containing all the necessary parameters for the encryption, and initialises them to a default value. In the provided example the relevant parameters are the following:

- Block size: the default is set to 128 and should not be changed.
- IV: by default contains a 128 bit initialization vector, generated randomly upon creation of the Aes instance.
- Key: by default contains a 256 bit key, generated randomly upon creation of the Aes instance.
- Mode: describes the block cipher mode, by default CBC.
- Padding: describes the padding mode, by default PKCS7.

In the proposed example, in place of the IV and Key values generated by the constructor it has been decided to obtain the values as parameters, for ease of management in the calling method.

The provided block cipher modes are: CBC, ECB, OFB, CFB, CTS. As previously explained, it is advised to avoid ECB. Unfortunately there are no available implementations for Authenticated Encryption (AE) modes.

The provided padding modes are:

- None: no padding is done. Even when applicable should be avoided as it could cause interpretation issues if upon decryption padding is expected.
- PKCS7: padding consists of a sequence of bytes, each of which is equal to the total number of padding bytes added.
- Zeros: padding consists of bytes set to 0. Should be avoided because makes the padding predictable and enables the attacker to perform some precomputations useful for attacks.
- ANSIX923: consist of a sequence of bytes set to 0 terminated by the length of the padding. Should be avoided because predictable.
- ISO10126: consists of random data terminated by the length of the padding.

An example of use for the encryption of uploaded .sqlite databases:

```
private void UploadWholeFile(HttpContextBase requestContext,
    List<ViewDataUploadFilesResult> statuses)
{

  var request = requestContext.Request;
  for (int i = 0; i < request.Files.Count; i++)
  {
    HttpPostedFileBase file=request.Files[i];
    //validate file
    //...
    //generate a random hexadecimal string for file name and verify uniqueness
    //...
    var fullPath = Path.Combine(StorageRoot, generatedRandomName);
    using (Aes aes = Aes.Create())
    {
      RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
      byte[] k = new byte[32];
      byte[] i = new byte[16];
      rng.GetNonZeroBytes(b);
      rng.GetNonZeroBytes(i);
      aes.Key = k;
      aes.IV = i;
      // Encrypt the string to an array of bytes.
      if(!AESEncryptStream(file.InputStream, aes.Key, aes.IV, fullPath)){
        if(File.Exists(fullPath))
          File.Delete(fullPath);
          //manage failure
      }
    }
    //...
  }
}
```

In the provided usage example it was decided to generate the key and initialisation vector using the RNGCryptoServiceProvider instead of the values provided by the Aes constructor as the documentation is not clear about how are those values generated.

The key used for AES encryption must be in its turn encrypted before being stored, for this purpose asymmetric encryption and decryption methods were implemented using the System.Security.Cryptography.RSACryptoServiceProvider class [40]:

```
public static byte[] RSAEncryptBytes(byte[] DataToEncrypt, RSAParameters RSAKeyInfo,
    bool DoOAEPPadding)
{
  try
  {
```

```
    byte[] encryptedData;
    //Create a new instance of RSACryptoServiceProvider and pass the key info.
    using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
    {
        RSA.ImportParameters(RSAKeyInfo);
        encryptedData = RSA.Encrypt(DataToEncrypt, DoOAEPPadding);
    }
    return encryptedData;
  }
  catch (CryptographicException e)
  {
    //...
    return null;
  }

}

public static byte[] RSADecryptBytes(byte[] DataToDecrypt, RSAParameters RSAKeyInfo,
    bool DoOAEPPadding)
{
  try
  {
      byte[] decryptedData;
      //Create a new instance of RSACryptoServiceProvider and pass the key info.
      using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
      {
          RSA.ImportParameters(RSAKeyInfo);
          decryptedData = RSA.Decrypt(DataToDecrypt, DoOAEPPadding);
      }
      return decryptedData;
  }
  catch (CryptographicException e)
  {
    //...
    return null;
  }
}
```

The RSACryptoServiceProvider class allows to encrypt or sign data using the RSA asymmetric algorithm. The constructor of the class creates an object generating a new key pair, with predefined parameters (modulus, exponent, etc.) however it is possible to provide different parameters using the ImportParameters method. RSACryptoServiceProvider's Encrypt and Decrypt methods, in addition to the data, accept also a boolean that should be true to perform the operation using OAEP padding (only available on a computer running Microsoft Windows XP or later) or false to use PKCS#1 v1.5 padding. If available OAEP is considered to be more secure.

The methods RSAEncryptBytes and RSADecryptBytes can be used in reverse order for signing and validating data.

The following code shows how the defined methods are used in SoD App to manage the keys used for AES:

```
private void UploadWholeFile(HttpContextBase requestContext,
    List<ViewDataUploadFilesResult> statuses)
{

  var request = requestContext.Request;
  for (int i = 0; i < request.Files.Count; i++)
  {
      HttpPostedFileBase file=request.Files[i];
    //validate file
```

```
        //...
        //generate a random hexadecimal string for file name and verify uniqueness
        //...
    var fullPath = Path.Combine(StorageRoot, generatedRandomName);
        using (Aes aes = Aes.Create())
    {
      RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
      byte[] b = new byte[32];
      rng.GetNonZeroBytes(b);
      aes.Key = b;
      // Encrypt the string to an array of bytes.
      if(!EncryptStream(file.InputStream, aes.Key, aes.IV, fullPath)){

           if(File.Exists(fullPath))
                File.Delete(fullPath);

            //manage failure
          }

      byte[] encryptedKey;

      FileStream fs = new FileStream(@"path to certificate", FileMode.Open,
          FileAccess.Read);
      int size = (int)fs.Length;
      byte[] rawCert = new byte[size];
      size = fs.Read(data, 0, size);
      fs.Close();

      X509Certificate2 certificate = new X509Certificate2(rawCert, "password",
          X509KeyStorageFlags.Exportable);

      using (RSACryptoServiceProvider RSA = certificate.PublicKey.Key as
          RSACryptoServiceProvider)
      {
        encryptedKey = RSAEncryptBytes(aes.Key, RSA.ExportParameters(false), true);
      }

      //..
    }
    //...
  }
}
```

To manage the key pair the System.Security.Cryptography.X509Certificates.X509Certificate2 class was used. The selected constructor takes as parameters a byte array containing the raw certificate, the password to the certificate and a key storage flag. The X509KeyStorageFlags must be set to Exportable to allow access to the private key.

The certificate that was used to instantiate the X509Certificate2 object is in .pfx format and was generated with the following command:

```
pkcs12 -export -in cert.crt -inkey prvKey.pem -out cert.pfx
```

Where cert.crt is a signed certificate in the following format:

```
    -----BEGIN CERTIFICATE-----
MIIDLDCCAhQCCQDX1PNRy2U/+zANBgkqhkiG9w0BAQsFADBXMQswCQYDVQQGEwJJ
VDETMBEGA1UECAwKU29tZS1TdGF0ZTEhMB8GA1UECgwYSW50ZXJuZXQgV2lkZ2l0
cyBQdHkgTHRkMRAwDgYDVQQDDAdtb3JnYW5hMCAXDTE4MTEwNjEzMzI1MVoYDzIy
...
3PPR+Bq4M7/T2VzxtPi/QHPGsGDt0Okg+4SIvBDk4ZU5z3jkj2G7Y4SpQglu/w4i
```

```
+ILRB1R1sET6lPYeMBBS4FwCv9U2DU9t2K1G6EX949AmhKVIdlHSLtHYmV/aSviO
-----END CERTIFICATE-----
```

And prvKey.pem contains the private key in the format:

```
   -----BEGIN PRIVATE KEY-----
MIIEvwIBADANBgkqhkiG9w0BAQEFAASCBKkwggSlAgEAAoIBAQDHEWoGLbAkaSOR
rghvWiqPgPJ2nJQYpcHW5QEqf6UOSrJiwyQEqE56OLuRQWgaDCeefnUHlTcZKcqi
zhFTxRVqRZrumj6bq/9Jm2MJD3KlDChlhszu8HWnpTrOm9le8ij7N2hAzNELYODg
...
o25x55hpZdNDtLvOLgz8kvtlX+b7BcbX1K9aj4jhAQKBgQCQuUFXU5t1wpwVzv2M
r6/eZvaC6rxT43jo/269UoQfhWcSu3ESWRxpa1d6bWuBOIt/KNNs+/WjGfLVqXFE
ufX4fhR9F6PljtJYAyzevMXkwQp+IEKJ6BbZVkRWgV9Q4dPZ6zxJRqTck9Ibu2lh
YLW5ZntdkkFI5GmsAWw9JLn1Qg==
-----END PRIVATE KEY-----
```

For this purpose self signed certificates are well suited, as the best approach is to generate special purpose certificates for data encryption that are frequently changed and are used only by the server.

The following snippet is finally used to decode the data:

```
FileStream fs = new FileStream(@"path to certificate", FileMode.Open,
    FileAccess.Read);
int size = (int)fs.Length;
byte[] rawCert = new byte[size];
size = fs.Read(data, 0, size);
fs.Close();

X509Certificate2 certificate = new X509Certificate2(rawCert, "password",
    X509KeyStorageFlags.Exportable);

using (RSACryptoServiceProvider RSA = certificate.PrivateKey as
    RSACryptoServiceProvider)
{
  decryptedData = RSADecryptBytes(encryptedData, RSA.ExportParameters(true), true);
}

//..
```

For simplicity of demonstration the snipped provides the password for the certificate as if it was hardcoded, however this should be avoided. In the following section it will be showed how to correctly manage keys and certificates.

3.6 Keys Management

Using the System.Security.Cryptography.X509Certificates.X509Store class, in .NET, it is possible to manage certificates. It represents a physical store where certificates are persisted and managed.

Using the Windows' certificate store the previous example, implementing RSA encryption, becomes:

```
using (X509Store store = new X509Store(StoreName.My, StoreLocation.CurrentUser))
private void UploadWholeFile(HttpContextBase requestContext,
    List<ViewDataUploadFilesResult> statuses)
```

```
{
  var request = requestContext.Request;
  for (int i = 0; i < request.Files.Count; i++)
  {
    //...
    using (Aes aes = Aes.Create())
    {
      //...
      byte[] encryptedKey;

      X509Certificate2 certificate;

     //get local store
     using (X509Store store = new X509Store(StoreName.My, StoreLocation.CurrentUser))
    {
      store.Open(OpenFlags.Read);
      //find certificate in store
      X509Certificate2Collection certs =
          store.Certificates.Find(X509FindType.FindByKeyUsage,
          X509KeyUsageFlags.KeyEncipherment, true);
      cert2 = certs[0];
      store.Close();
    }

    using (RSACryptoServiceProvider RSA = certificate.PublicKey.Key as
        RSACryptoServiceProvider)
    {
      encryptedKey = RSAEncryptBytes(aes.Key, RSA.ExportParameters(false), true);
      }

      //..
    }
    //...
  }
}
```

The X509Store constructor takes as parameters the store name and location. The store names are those defined by the Windows' Certificate Manager tool (Certmgr.exe), and can be chosen by the developer as he sees more fit. The selected store can reside in one of two store locations: LocalMachine and CurrentUser. The first contains a set of stores assigned to the local machine and is global to all users, however it is necessary to provide administrator privileges to the application in order to access it therefore a more secure solution is to use the CurrentUser location, that contains stores local to a specific user.

After instantiating the store there are many parameters by which it is possible to retrieve the certificate: by Thumbprint, SubjectName, by SubjectDistinguishedName, by IssuerName, by IssuerDistinguishedName, by SerialNumber, by TimeValid, by TimeNotYetValid, by TimeExpired, by TemplateName, by ApplicationPolicy, by CertificatePolicy, by Extension, by KeyUsage or by SubjectKeyIdentifier.

However if extensions are used, for example by selecting KeyUsage as in the proposed snippet, the certificate will have to be created accordingly, containing the respective extension.

The Find function searches the X509Certificate2Collection object contained in the X509Store and accepts as input three parameters, the first specifying the attribute type used for the search, the second specifying the desired attribute value and the last specifying whether to find only valid certificates.

Using an extension such as KeyUsage allows to ignore the actual version of the key when retrieving it from the store. In SoDApp's case the store is maintained to contain only one valid key that has the extension's value set to keyEncipherment at a time and therefore when the key is periodically changed the code is not affected.

For SoD App, certificates are generated and substituted manually, however the following code could be used to add a certificate programmatically:

```
FileStream f = new FileStream(@"path to certificate", FileMode.Open, FileAccess.Read);
int size = (int)f.Length;
byte[] rawCert = new byte[size];
size = f.Read(data, 0, size);
f.Close();

X509Certificate2 certificate = new X509Certificate2( rawCert, "password",
    X509KeyStorageFlags.MachineKeySet | X509KeyStorageFlags.PersistKeySet |
    X509KeyStorageFlags.Exportable);

using (X509Store store = new X509Store(StoreName.My, StoreLocation.CurrentUser))
{
   store.Open(OpenFlags.ReadWrite);
   X509Certificate2Collection certs =
       store.Certificates.Find(X509FindType.FindByKeyUsage,
       X509KeyUsageFlags.KeyEncipherment, true);
   store.Certificates.Remove(certs[0]);
   store.Add(certificate);
   store.Close();
}
```

In this case the password and location of the certificate would be provided by administrator as input.

3.7 Error handling

3.7.1 Exceptions

Whenever there is a piece of code that deals with variables whose value depends on factors external to the application, exceptions could be raised. Some common examples where this verifies are file accesses, network connections, input values, etc. Properly written code should handle such actions by surrounding them with Try-Catch statements, in order to capture the exceptions and properly manage them.

```
public void Analyse(AnalysisJob analysisJob)
{
  try {
     //.. dispatch some analyses on a thread
  }
  //manage most common exceptions
  catch (ThreadAbortException e)
  {
    if (!analysisJob.status.Equals(AnalysisJobStatus.ABORTED))
    {
      if (analysisJob.status.Equals(AnalysisJobStatus.LOADING))
      {
        //properly release resources
      }
      //log failure
      LogHelper.LogHelper.Fatal("Analysis job aborted for user
          ''+analysisJob.ARCHIVE.PROJECT.USER+'' on execute job at " + e.Source +
          e.Message);
      //react to failure
      AbortAnalysis(analysisJob.Analysis.ID);
```

```
    }
  }
  //manage generic exceptions
  catch (Exception e)
  {
    AnalysisJobManager.GetInstance().ChangeJobStatus(analysisJob.Analysis.ID,
        AnalysisJobStatus.FAILED);
    LogHelper.LogHelper.Fatal("Unexpected error for user
        ''+analysisJob.ARCHIVE.PROJECT.USER+'' in analysis job on execute job at " +
        e.Source + e.Message);
  }
}
```

The example shows how failures should be managed, dealing explicitly with the most common exceptions and logging the details of the failure. Catch statements should never be empty or inconclusive but should implement the necessary steps to guarantee that the application can continue to run smoothly and should report the failure to the log. Failure to comply with the first could allow an attacker to exploit the exception. Properly logging unexpected events can help instead in individuating vulnerabilities, how often they occur and if there is suspicious activity, such as a user is repeatedly triggering a specific error.

Among other things, logs should also be employed to provide relevant information that could help detect attacks and suspicious accounts. However it is necessary to be careful to never log sensitive data or too much information about the application, as logs too could be a target for an attacker trying to gather intel on a software.

3.7.2 Resource Release

Another common source of error is the improper release of resources. If the code is not correctly designed an exception raising during the use of the resource could result in a subsequent piece of code, where the resource is released, to never be executed. There are two main approaches to properly handle resources: the Using statement and the Try-Catch-Finally statement.

Using statement: defines a scope of use for a certain resource and manages the correct disposal of objects implementing the IDisposable interface.

```
using (OleDbConnection connection = new OleDbConnection(connectionString))
{
  connection.Open();
  ...
  db.flushTable(tableName);
  reader.Close();
}
```

In this case the Using statement automatically disposes of the connection when the end of the scope is reached. Since the connection cannot be closed without closing the reader also the reader will be correctly disposed. An error triggered inside the using statement will result in the execution exiting the Using scope and therefore the resources will be released.

Try-Catch-Finally statement: the resources are obtained and used in the try block, exceptions are managed in the catch blocks and resources are released in the finally block that is always executed.

```
try{
  connection.Open();
  ...
  db.flushTable(tableName);
}catch(OleDbException ode){
  //manage db exception
}catch(Exception e){
```

```
  //manage generic exception
}finally{
  reader.Close();
  connection.Close();
  connection=null;
}
```

The Using statement has the advantage that the developer does not have to manage the disposal of resources, and therefore is the safest and least error prone option, however it does not provide a mechanism to handle raised exceptions as the Try-Catch-Finally statement does. A possible solution could be to wrap the Using statement in a Try block and manage the Exceptions in the Catch blocks.

3.7.3 Debug Features

Other common scenarios include developers leaving debug code in the release versions, exception messages being used directly as output, etc. This features usually disclose dangerous information about the software.

A good example was found during the testing of the SoD App. The developers had left in the release version of the application a feature that should have been clearly intended only for the development version.

```
public ActionResult Download(string fileName)
{
  try
  {
    //...
  }
  catch (Exception e)
  {
    //log failure
    LogHelper.Fatal("Unexpected error from user " + User.Identity.Name.ToUpper() + "
        on controller " +
        this.ControllerContext.RouteData.Values["controller"].ToString().ToUpper() +
        ", on method " +
        this.ControllerContext.RouteData.Values["action"].ToString().ToUpper());

    //notify user
    return View("~/Views/Shared/Error.cshtml", new HandleErrorInfo(e,
        this.ControllerContext.RouteData.Values["controller"].ToString(),
        this.ControllerContext.RouteData.Values["action"].ToString()));
  }
}
```

In this example the System.Web.Mvc.HandleErrorInfo class is instantiated passing the exception object, the name of the controller class and the name of the action method where the exception originated. As can be seen in figure 3.7.3 what is displayed in the error view for the user is a great deal of information, an attacker could gain a lot of insight on the application and filesystem structure by triggering errors on the website.

In order to mitigate the risk of debug code remaining in the release version it is good practice to wrap all such code using preprocessor directives. When the compiler encounters an #if DEBUG directive it will execute the contained code only if the current mode is Debug.

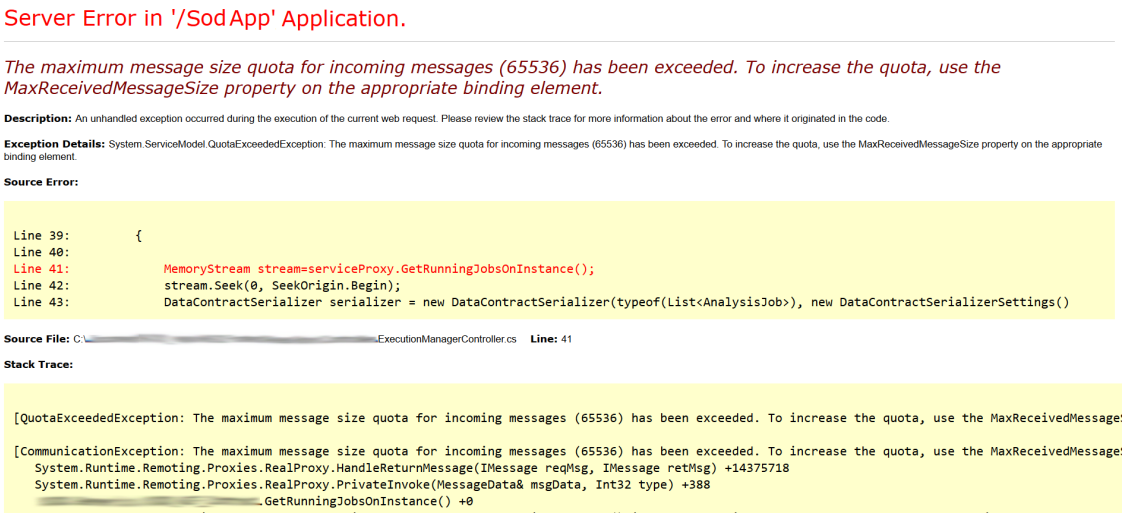

Figure 3.3. Example of HandleErrorInfo output to Error view.

```
public ActionResult Download(string fileName)
{
  try
  {
    //...
  }
  catch (Exception e)
  {
    //log failure
    LogHelper.Fatal("Unexpected error from user " + User.Identity.Name.ToUpper() + "
        on controller " +
        this.ControllerContext.RouteData.Values["controller"].ToString().ToUpper() +
        ", on method " +
        this.ControllerContext.RouteData.Values["action"].ToString().ToUpper());

    //notify user
#if DEBUG then
    return View("~/Views/Shared/DebugError.cshtml", new HandleErrorInfo(e,
        this.ControllerContext.RouteData.Values["controller"].ToString(),
        this.ControllerContext.RouteData.Values["action"].ToString()));
#else
    return View("~/Views/Shared/UserError.cshtml");
#endif
  }
}
```

This solution however requires that the project is configured with the correct build mode. If using Visual Studio, in order to correctly implement the release mode, all projects in the solution and the solution itself must have the build configuration set to Release.

3.8 Analysis of support tools and environment

This section was introduced in the procedure as it is deemed highly important in the process of securing a web application, however it was not in the scope of the activity performed by the candidate and therefore the practical aspects have not been analysed.

Chapter 4

Review and Maintenance

When the securing of the web application has been completed a different person/team should review and test the results of the analysis and implementation performed in the previous phases.

The application could be analysed again using the previously proposed Black Box and White Box tools and hopefully the report should not detect anything relevant. An even better alternative would be to hire a team of expert penetration testers to perform some in depth tests on both the application and the server configuration, by cloning the web server to another machine that can be freely attacked by the pen-testers.

The pen-testers will be able to give a more accurate picture of the remaining risks, and could also perform social engineering tests to verify the actual physical security of the devices and the employee's awareness.

After it has been established that the risks have been mitigated to a satisfactory degree, the tester should define a maintenance policy for the web application. The policy should define at least the following:

- Who is in charge of performing the maintenance. At least two distinct persons that for certain tasks, such as keys management, should be both required.
- How often are the different tasks of the maintenance performed.
- Evaluate the security measures implemented towards the user and establish whether they are still secure with respect to the evolution of technologies and attacks. Such measures could be authentication methods, password renewal process, identity verification process, etc.
- How to manage key renewals. For each type of key or certificate that has to be renewed the policy should specify the correct procedure and how to deal with old keys.
- How to manage database re-keying.
- How to manage the update of used libraries. A list of libraries with their version should be maintained and updated accordingly.
- How to manage the update of used software and services. It is advised to configure automatic updates wherever possible and especially for security updates.
- How to manage the list of algorithms and ciphers used. A list should be maintained and updated.
- How to manage the application routine analysis using Black Box and White Box analysis tools.
- How testing of the application by a penetration testing team.
- How to manage backups.

The application should be designed to make the aforementioned tasks as simple to execute as possible, providing specific interfaces and features to the administrator. If the maintenance is too cumbersome there is the possibility that busy administrators will skip over tasks deemed less important or too time consuming, or perform them badly, which would result in the deterioration of the application's security.

Chapter 5

Conclusions

Securing a web application is a very complex task that requires a lot of attention to many different aspects and details.

The Internet offers a great variety of tools and information to help in the process, however most of this information is partial, outdated, incorrect, situational, making it a complex process to distinguish and filter the correct approach.

The procedure devised by the candidate adopts a structure where first the analytic part is discussed, with the aim of providing a structured and organised procedure to tackle the securing of the application minimising the chances of missing important details. This part also gives all the necessary information to understand the most common vulnerabilities found in web application, in order to arm the tester with the right tools to analyse the software at hand with an educated and critical approach.

The procedure then offers practical examples extracted from the real case study confronted by the candidate. This part is the result of the research and analysis performed on the solutions found online, and offers those that were evaluated and tested by the candidate to be correct approaches, along with an the explanation of how they improve the security of the application.

The idea of structuring the procedure as described derives from the following observations and experiences, made during the practical experience of the candidate:

- Often applications are not developed with security in mind, therefore it has to be implemented afterwards. This unstructured approach could easily result in security being implemented messily and leaving many vulnerabilities. The idea of a structured procedure came to the candidate after realising that an organised approach was necessary for her task in order to minimise the chances of important details being missed.

- Securing of software is often delegated to third parties that have little to no communication with the team of developers. The "Understanding the Application" section is the result of what the candidate found was necessary knowledge to gather in order to be able to correctly analyse the application.

- Security is not seen as a task that brings additional value but as a necessary and costly task. As a consequence insufficient time and resources are invested in the process. The purpose of this dissertation is to concentrate most of the necessary information needed during the securing, in order to allow the testers to optimize their time and focus and be able to produce more secure software.

Some of the topics covered in this dissertation were not actually implemented on the real case study application, as the amount of time available was sufficient to research, define and test the correct implementations for only some of them, therefore some priorities had to be defined, resulting in an incomplete job.

The candidate feels that if she had had a guideline that would have helped her from the beginning to correctly structure the work, and provided examples to address her toward correctly formulated solution, she might have produced a significantly more secure application.

The exact implementation examples unfortunately have a very restricted scope of utility, as they require that the reader is using a similar intersection of framework, version and application structure. However they are not meant to provide copy paste code ready to work, on the contrary they are designed to provide an idea of correct implementation, highlighting the most important factors to consider when implementing custom solutions.

The wish of the candidate is that this work might help other developers to easily and more efficiently integrate all the proper security measures in their applications, investing more time in discovering and fixing vulnerabilities and less in repairing mistakes done during the process, redoing work and discriminating valuable information versus misguided approaches.

Bibliography

[1] Roadmap for defeating DDOS attacks, https://www.sans.org/dosstep/roadmap

[2] Legacy System, https://en.wikipedia.org/wiki/Legacy_system

[3] Open Web Application Security Project, https://www.owasp.org

[4] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[5] Principle of Least Privilege, https://en.wikipedia.org/wiki/Principle_of_least_privilege

[6] Social Engineering, https://en.wikipedia.org/wiki/Social_engineering_(security)

[7] OWASP ZAP, https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[8] Brute-force Attack, https://en.wikipedia.org/wiki/Brute-force_attack

[9] Jeremiah Grossman, "Cross-Site Tracing", 20 January 2003 http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf

[10] Arshan Dabirsiaghi, "Bypassing Web Authentication and Authorization with HTTP Verb Tampering", 28 May 2008 https://dl.packetstormsecurity.net/papers/web/Bypassing_VBAAC_with_HTTP_Verb_Tampering.pdf

[11] Denial of Service attack, https://www.owasp.org/index.php/Denial_of_Service

[12] J. Klensin, "Application Techniques for Checking and Transformation of Names", RFC-3696, February 2004, DOI 10.17487/RFC3696

[13] Input Validation Cheat Sheet, https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet#Email_Address_Validation

[14] Session Management, https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

[15] Token Lengh, https://www.owasp.org/index.php/Insufficient_Session-ID_Length

[16] Payment Card Industry (PCI) Data Security Standard, "Requirements and Security Assessment Procedures", May 2018 https://www.pcisecuritystandards.org/document_library

[17] Meltem Sönmez Turan, Elaine Barker, William Burr, Lily Chen, "Recommendation for Password-Based Key Derivation: Part 1: Storage Applications", SP 800-132, December 2010 DOI 10.6028/NIST.SP.800-132

[18] Elaine Barker, "Recommendation for Key Management", SP 800-57, January 2016, DOI 10.6028/NIST.SP.800-57pt1r4

[19] Checkmarx, Application Security Testing and Static Code Analysis Tool, https://www.checkmarx.com

[20] Key Size, https://en.wikipedia.org/wiki/Key_size#cite_note-15

[21] System.Web.Mvc.ModelStateDictionary class, https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.modelstatedictionary?view=aspnet-mvc-5.2

[22] System.ComponentModel.Data Annotations namespace, https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations?view=netframework-4.7.2

[23] Entity Framework, https://docs.microsoft.com/en-gb/ef/ef6/index

[24] System.Web.Security.AntiXss.AntiXssEncoder class, https://docs.microsoft.com/en-us/dotnet/api/system.web.security.antixss.antixssencoder?view=netframework-4.7

[25] OLE DB.NET Framework Data Provider namespace, https://docs.microsoft.com/it-it/dotnet/api/system.data.oledb?view=netframework-4.7.2

[26] MIME Type Detection in Windows Internet Explorer, https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/ms775147(v=vs.85)
[27] HttpRuntimeSection.MaxRequestLength property, https://docs.microsoft.com/en-us/dotnet/api/system.web.configuration.httpruntimesection.maxrequestlength?view=netframework-4.7.2
[28] Request Limits element, https://docs.microsoft.com/en-us/iis/configuration/system.webserver/security/requestfiltering/requestlimits/
[29] System.IO.Compression.ZipArchive Class, https://docs.microsoft.com/en-us/dotnet/api/system.io.compression.ziparchive?redirectedfrom=MSDN&view=netframework-4.7.2
[30] SecureString Class, https://docs.microsoft.com/en-us/dotnet/api/system.security.securestring?redirectedfrom=MSDN&view=netframework-4.7.2
[31] System.Security.Cryptography.RNGCryptoServiceProvider class, https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rngcryptoserviceprovider?redirectedfrom=MSDN&view=netframework-4.7.2
[32] HttpCookie class, https://docs.microsoft.com/it-it/dotnet/api/system.web.httpcookie?redirectedfrom=MSDN&view=netframework-4.7.2
[33] HttpSessionState class, https://docs.microsoft.com/en-us/dotnet/api/system.web.sessionstate.httpsessionstate?view=netframework-4.7.2
[34] Argon2, https://github.com/p-h-c/phc-winner-argon2
[35] .NET Argon2 library, https://github.com/kmaragon/Konscious.Security.Cryptography
[36] A. Biryukov, D. Dinu, D. Khovratovich, Internet-Draft, "The memory-hard Argon2 password hash and proof-of-work function", 03 August 2017, https://tools.ietf.org/html/draft-irtf-cfrg-argon2-03
[37] Joel Alwen, Jeremiah Blocki, "Towards Practical Attacks on Argon2i and Balloon Hashing", 2017 IEEE European Symposium on Security and Privacy (EuroS&P), Paris (France), 26-28 April, 2017, pp. 9-10, DOI 10.1109/EuroSP.2017.47
[38] OWASP, Password Storage Cheat Sheet, https://www.owasp.org/index.php/Password_Storage_Cheat_Sheetl
[39] GoogleAuthenticator, https://github.com/BrandonPotter/GoogleAuthenticator
[40] System.Security.Cryptography.RSACryptoServiceProvider class, https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rsacryptoserviceprovider?view=netframework-4.7.2