Web development: Management and foundation of a secure application

Camila Fonseca
DETI
University of Aveiro
Aveiro, Portugal
cffonseca@ua.pt

Pedro Pinto
DETI
University of Aveiro
Aveiro, Portugal
pspinto@ua.pt

Rodrigo Lima
DETI
University of Aveiro
Aveiro, Portugal
rodrigoflima@ua.pt

***Abstract*—A successful web application requires security to be a priority during early planning, management and development, in order to ensure ensure correctness and normal behavior, protected from external and internal threats. Recent research has shown that most companies are vulnerable to cyberattacks because they don't implement a solid strategy in the development nor in the maintenance phase and due to current threats, data exposure is a huge risk that must be mitigated, making this process more critical than ever. This project aims to develop a web application for selling products, providing a safe environment to perform online payments through the implementation of secure coding and latest tools, making it a more secure and scalable application.**

***Index Terms*—web, security, reliability, robustness**

I. Introduction

Over the years, the number of attacks targeting web application have been increasing, harming online business, due to the lack of strategies implemented by development companies to counteract phenomena like software supply chain attacks, distributed denial of service (DDoS), or common well-known vulnerabilities [1]. This type of cyberattacks might compromise its availability, change the normal operation and give access to key systems or information.

This document proposes the elaboration of a strategy, containing the key points needed to develop a secure web marketplace, which, by design, offers a vast surface of attack due to the high user interaction it requires. To make it resilient to common and recent attacks, which threaten the normal behavior of the business model, we must take in consideration three of the most important pillars of risk management: security by design, defense in depth and zero trust policy; it is also essential to mention and understand the main sources of attacks, in order to provide a practical solution:

A. Availability

The availability of service can be compromised for a number of reasons. It may be caused by poorly handled exceptions or other harmful practices in code, or by malicious actors.

Taking the necessary precautions can prevent a malicious actor from causing a denial of service (DOS) [2] and compromising the availability of a product.

B. Input validation and sanitization

GET requests, POST requests, and cookies are the main channels through which data input is sent from the user's browser to the web server.

Users can edit, modify, and manipulate cookies to gain access to the web server. As encoded data passes into the web server, input sanitization filters it. By implementing this filtering, the product becomes more secure and robust, preventing the exposure of sensitive data and undefined behavior in our service.

C. Enforcement of good data management practices and data laws

Due to the nature of a web application that is designed to be interacted with, some user data needs to be stored. Having user accounts requires sensitive data such as emails and passwords to exist, and these are a valuable commodity sought after by malicious actors. Besides, compliance with existing regulation must be ensured.

To this end, all kinds of user data should be stored and transported safely, and kept to the bare minimum necessary for the service to function. Confidential and sensitive data should be stored encrypted or not at all, if it can be avoided.

D. Supply chain quality assurance

To ease development, it is common that libraries and external dependencies are used in web development. However, these present an extended attack surface that is often out of our control. To keep the risk to a minimum, dependencies should be kept updated, and analyzed with external tools to detect vulnerabilities in those, so that we may be aware and properly prepared.

In case a dependency is known to be compromised, it should be quickly swapped out, which requires low code coupling in our service itself.

E. Non repudiation and accountability

Due to the transactional nature of a web store, once an user has taken an action it cannot and should not be rolled back by either of the parties, and it's important to ensure that it cannot be plausibly denied a transaction took place. To achieve this, actions taken by users should have immediate feedback and a

record of the action taken, such as emails, for the user needs to be sure it took place. From a non-user-facing perspective, thorough logs should be created and kept for an adequately long amount of time, complete with timestamps.

II. Secure Life Cycle Description

A. Education and Awareness

The development team should coordinate, between the heads of each department, structure and basic security practices in collaboration with security experts, to raise awareness about recent threats and procedures taken to solve/mitigate. Additional training should be given, not only to developers but also to every collaborator in the company, to prevent social engineering attacks that might leak confidential information to attackers.

Programmers directly involved in the software project should be aware of all the dependencies attached to their work, such as, tools or frameworks that might bring additional vulnerabilities to the application.

B. Project Inception

This is a preliminary phase of new project or realignment an existing one, aimed to define the problem and propose a solution, analyze internal and external factors that might impact the project, performing an environmental analysis (e.g. SWOT, PERT), conduct a feasibility study to evaluate the feasibility and cost-effectiveness of our solution, define a timeline for its implementation and assess the risks that may jeopardize the security of our application.

Some of the decisions, related to security, might also take in consideration cost and time needed to solve the vulnerabilities in order to keep the delivery schedule. Those already known in the Common Vulnerabilities and Exposures (CVE) are already classified and their score might give some influence at the time of the decision, the security experts should, also, classify any unknown vulnerabilities that might appear during the development phase attributing a score based on the Common Vulnerability Scoring System (CVSS), to evaluate the theat level of a vulnerability based on three metric groups: Base, Temporal and Environmental.

C. Analysis and Requirements

Analysis and requirements gathering is an essential step to any software development process, but specially so when planning a secure app. In this phase both functional and non-functional requirements should be specified.

Having everything planned out from the beginning allows for easier risk management, as well as vastly reducing the chances of features thought of later on in the cycle introducing unforeseen vulnerabilities and weaknesses.

Requirements should be exhaustive so that they can be referred to later on in the process, providing a common reference for everyone working on the project. These requirements should be composed by a mixed team of developers and security engineers, so that the requirements for the store don't overlook functionality for security or vice-versa.

D. Architectural and Detailed Design

All technical details of the architecture should be discussed and documented, not only what has been selected but also why it has been selected. This way, future mistakes can be avoided.

The documentation should be comprehensive, and include specific details such as installed versions, and it should be kept updated as the software itself is updated. This type of internal documentation should be backed up either to an internal server or an external service such as Notion to ensure there are always backups.

A security and risk assessment should be carried out, carefully detailing possible exposed surfaces, such as user-facing services, and the defense mechanisms in use such as cryptography for data encryption, protocols and who should be able to access what.

E. Implementation and Testing

To ensure reliability and robustness, secure software development practices such as code reviews, coding standards, and Test Driven Development (TDD) should be used. The use of coding standards will facilitate maintainability while catching unwanted bugs and behaviors. Tests and verifications must be performed on the developed code. There should be coverage of all aspects of testing, from unit tests to end-to-end tests (E2E).

The tests shouldn't only reflect basic service interaction, they should also cover edge cases, impossible input scenarios, and error handling. This can be achieved with fuzzing or with the help of a tool like for example sonar cloud.

Audits of the full service should be conducted. These should also cover security. It is possible to identify errors in code and in runtime by performing penetration testing and static code analysis as well as dynamic code analysis.

The flaws found in the service should be taken very seriously and should not be rediscovered during subsequent audits.

F. Release, deployment, and support

Best practices should be followed when releasing and deploying a service. They can range from load balancing to security.

In order to ensure a smooth life cycle, it is highly recommended to create an Incident Response plan and a Deployment and Release plan.

The deployment can be made with the help of infrastructure as code, for example use of Docker, Kubernetes, Ansible or and Terraform.

As a service matures, security flaws and unexpected events are bound to happen. It is the responsibility of the creator or maintainer of the service to handle these events.

When responding to an incident, transparency and legal compliance should be the top priorities.

Following an incident, documentation and reoccurrence prevention should be addressed.

III. Security Requirements

As part of this section, we will be discussing the set of security requirements that are essential to the development of the proposed solution. These requirements can be divided into different types based on the type of specification and the security requirement.

These requirements are clear and concise, with the purpose of not getting misunderstood because they are fundamental for the correct functioning of the application. Also, they should be testable and measurable, in order to achieve their feasibility. The following table enumerates those requirements, each of them with a unique identifier starting with REQ, followed by an integer.

ID	Requirement Description	Comment	Type
REQ-1	The system shall have 99.99% of uptime service.	Keeping the server available in the event of crashes or DoS attacks.	Availability
REQ-2	The system shall apply load balancing.	Redundancy will minimize the impact of system failures.	Availability, Performance
REQ-3	The system shall record HTTP & HTTPS requests sources.	Maintainability, rollback, and forensic analysis are made easier.	Non-Repudiation, Accountability
REQ-4	The system shall have all its communications encrypted (HTTPS).	Sniffing and men-in-the-middle attacks can be forewarned.	Confidentiality
REQ-5	The system shall have its stored data encrypted.	Improved data integrity and compliance with GDPR.	Confidentiality, Integrity
REQ-6	The system database shall be backed up daily.	Defending against ransomware attacks and data loss.	Integrity
REQ-7	The system shall implement strict user input sanitization while taking into account the input context.	Lack of sanitization can lead to XSS, command injection, SQL injection, and other attacks.	Integrity
REQ-8	The system shall apply WAF or DDoS protection appliance.	Preventing layer 7 (application-level) attacks.	Availability
REQ-9	The system shall authorize the access to personal account data.	Only the respective user can access its personal settings.	Confidentiality
REQ-10	The system shall apply malware control.	Prevent and detect malware by means of antivirus or firewalls.	Security Policy
REQ-11	The system shall prevent clickjacking and Cross-Site Request Forgery.	Prevent common attacks which mirrors the website and/or execute actions without the user's consent.	Security Policy
REQ-12	The system shall have vulnerability management, for continuous inspection of code quality and perform automatic reviews (SonarQube).	A process to identify and rectify vulnerabilities in the application, patch process.	Attack detection
REQ-13	The system shall secure the design of the logic of the web application.	Ensure the proper implementation and rejected any deviating behavior.	Integrity
REQ-14	The system shall authenticate via cryptographically secure tokens (cookies).	Communications related to authentication need to be encrypted.	Confidentiality, Authentication
REQ-15	The system shall not give any meaningful feedback on failed log-in attempts.	Avoiding giving possible attackers any information that may aid them in enumerating users or guessing passwords.	Confidentiality, Authentication
REQ-16	The system shall block the user for an hour, when he fails to login 3 times in less than 5 minutes.	To prevent brute force attacks on user credentials.	Integrity
REQ-17	The system shall log every transaction done on the platform.	There must be a record of previous transactions in case there's a dispute or there's a system error and a rollback is needed.	Non-Repudiation, Auditing
REQ-18	The system shall handle all exceptions, even if just to prevent attackers from knowing it occured.	Unhandled exceptions can provide attackers with information about the inner workings of the system.	Confidentiality
REQ-19	The system shall hide specific error messages from the user.	Generic messages should be exchanged, providing minimum information to a potential attacker.	Confidentiality, Integrity
REQ-20	The system shall log all server activity generated by the user.	Detecting possible attacks, and investigating previous ones.	Accountability

IV. Penetration Testing

Penetration testing is an important method to ensure the web application is not vulnerable to the most common attack angles by mimicking an external attacker's behavior, with variable degrees of internal system knowledge, and it should be carried out often, with variable scope ranges, while balancing the cost of carrying out a penetration test with the value gained from it. If the vulnerabilities found from a costly in-depth engagement are trivial, which could just as easily have been found with an automated Vulnerability Scanner, it would not justify such an endeavor. As such, a variety of approaches should be employed.

A. Approaches - Cost vs Benefit

Penetration tests can be costly, as the technical knowledge needed to make hiring professionals expensive, and as such they should be used only when their usefulness can be maximized. Since the most likely event to introduce new vulnerabilities into the system is after changes, new updates being deployed are a good metric to use in order to decide when to carry out an in-depth penetration test. It is important to note that such penetration tests should be carried out *before* the update is deployed to a production environment.

Major changes to infrastructure or application logic should be thoroughly tested before being deployed, and as such, ideally have a white-box (That is, a penetration test where the source code and configuration files are available to the tester.) assessment done, to make it possible to catch as many errors and be able to reach as far into the system as possible. Smaller changes to the code should always be monitored with automated tools, but may not require a new penetration test to be done.

However, white-box penetration tests aren't always desirable, since a malicious actor would not have access to that kind of information, and resources would be spent mitigating angles of attack that are unfeasible in a real-world scenario. As such, black-box assessments have a narrow scope, reducing costs, and as such is feasible to be done more often. Periodically, these should be employed even if there are no alterations to the codebase, to cover possible previously missed vulnerabilities.

B. Tools

A web marketplace doesn't have the most stringent security requirements - if there's a zero-day released to the public, it is feasible to shut down the servers for a fix - and as such doesn't require an in-house, full-time pentesting team available. As such, penetration tests would ideally be outsourced to an external company, and then their output is taken into consideration by the development and testing teams.

So, in-house, custom proprietary tools would most certainly not be needed, leaving the choice of tool to the hired penetration testers. It is expected the use of an automated scanner to cover the most well-known and obvious attack surfaces, as well as existing public tools that make up the arsenal of most penetration testers, like the Metasploit Framework, Burp Suite and SQLMap.

C. Targets

As a web marketplace is a web application, most of the penetration testing should be focused on specifically Web Vulnerabilities. For this, the methodology employed should be to collect as much information on the application as possible - What technologies are being used, what internal services may exist, what data can be extracted from DNS records and other kinds of data that can be obtained via solid Recon skills.

The focus of such a penetration test should be the vulnerabilities present in the OWASP Top Ten, since these are the most common afflicting web applications. Since the backbone of a web marketplace is its Database, and many parts of the application interact and rely on it, SQL Injection should be a focus in a penetration test. Particular attention should be given to the control panels offered to the sellers on the app, which being strongly tied with user input are particularly vulnerable.

The other big aspect that should be thoroughly be tested is Access Control, as it is a very important component on the application, on which hinge many of the other security aspects. It is much less useful to take special care protecting against SQL injections if the seller panels are open to anyone, allowing anyone to manipulate the Database, and admin panels being easily reachable make many attacks trivial.

Access control is a broad subject, covering such attacks as those targeting Cryptographic methods, which encompass proper password storage, to avoid them being cracked in case of a data leak, and proper usage of Cookie Signing and Encrypting to avoid them being tampered with or easily stolen, among others. Since this web application is a web marketplace, it is vital to have protections in place to prevent users from impersonating others and accessing data they shouldn't have access to, and for those protections to be thoroughly and constantly tested.

As most of the Web Application is user-facing, and it is expected for users to interact with it, it is particularly important to test the mechanisms given to the user for this interaction. Particularly, any input forms that accept direct user input. These would be a primary target for any attacker, and as such must be very well secured. Penetration testing is what assures us that this security exists.

Aside from the webapp frontend, there are other targets worthy of being tested as well, such as the organization who works on development itself. There's multiple angles of attack that target people, such as social engineering and phishing, and these can be tested, in an attempt to compromise them. An attacker obtaining access to the development environment or developer tools, such as Kubernetes or slack, could be just as damaging as a SQL Injection vulnerability being found and as such, just as necessary to be put to the test. The infrastructure surrounding the web application is also a possible target, and the processes through which the organization operates may be part of an engagement's scope as well, although these, given proper access control, should not be ever accessible.

V. Fuzz testing

An automated way of finding implementation bugs using malformed/semi-malformed data injection is called fuzz testing or fuzzing.

Fuzz testing is a powerful tool that helps to identify potential security vulnerabilities in a system or application. It is a form of automated testing that works by sending unexpected, random inputs to the system in order to uncover any potential errors or bugs. Fuzz testing is an essential part of a comprehensive security testing process and helps to protect against malicious attacks.

Fuzzing relies on the assumption that every program contains bugs waiting to be discovered. As a result, a systematic approach should eventually find them.

It is an alternative method of testing software (hand code review, debugging) due to its non-human nature. By requiring only a small amount of work to implement, it complements other software testing methodologies but does not replace them entirely.

A. Tools

Fuzzing tools are programs designed to generate random or unexpected inputs to test the system under test. Fuzzing tools are used to send a barrage of random inputs to the program in order to find edge cases and possible bugs or errors. There are various different types of fuzzing tools, each with its own characteristics and applications. Some common types of fuzzing tools include mutation-based fuzzers, generation-based fuzzers, and hybrid fuzzers. Mutation-based fuzzers mutate existing data to generate new inputs, while generation-based fuzzers use rule-based processes to generate new inputs. Hybrid fuzzers use both mutation and generation techniques. Fuzzing tools are available for a range of different platforms and programming languages, including Windows, Linux, Android, and JavaScript.

B. Methodology

The methodology that will be employed is based on selecting a suitable fuzzing tool. When selecting a fuzzing tool, the one chosen must be one that is compatible with the language used in the application. In addition, it should be able to generate valid data as well as invalid data.

The first step is to identify all input points in the application that accept user-controlled input. These may include parameters passed to the application, the source of data, and the format of the data.

The next step is to create inputs that cover the full range of potential inputs, including expected and unexpected values. The tool should also create a variety of different inputs, such as POST data and file inputs.

The final step is to execute the tests in the application and monitor for any errors or exceptions. QA should also perform manual testing to complement the testing being done.

The most suitable fuzzing tools for this project are AFL, developed by Google, and Ffuf, both open-source and actively maintained.

Ffuf can be used for the front end and back end while AFL can be used for the back end.

C. Attack Vectors

Web applications are a critical part of most businesses, and as such, they can be vulnerable to attack. Common web application attack vectors include SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). These attacks can be used to access confidential information or to manipulate data.

Fuzzing can be used to identify weak spots and can even be used to gain access to the web application, if applied correctly, it will reveal critical bugs in software. Some of those bugs can reveal attack vectors within the application that is being tested.

Some of those attack vectors include:

- Use of bad characters in inputs
- Mismanagement of allocated memory
- Format String Errors (FSE)
- Integer Overflows (INT)
- Miss handling of errors
- SQL injection
- XSS
- Insecure direct object references (IDOR)
- Insecure deserialization
- File upload vulnerabilities

Errors can be detected by fuzzing inputs in a variety of formats and forms. As an example, some inputs that can be generated by the fuzzing tool to test the attack vectors mentioned, are the following:

- Use of bad characters in inputs

  `\x00\x0a\x20`
- Mismanagement of allocated memory

  `AAAAAAAAAAAAAAAAAAAAAAAAAAA...A`
- Format String Errors (FSE)

  `%s%p%x%d`
- Integer Overflows (INT)

  `0x3fffffff`
- Miss handling of errors

  `20/20/2020`
- SQL injection

  `' OR 1=1--`
- XSS

  `>"><script>alert("XSS")</script>&`
- Insecure direct object references (IDOR)

  `http://www.example.com/account?id=0`
- Insecure deserialization

  `)....username.....admin..role.....admin..`
- File upload vulnerabilities

  `notAPDFfile.pdf.exe`

If bugs are present, the application will have erratic behavior if met with an attack through the previously mentioned attack vectors. To mitigate this issue, secure and responsible behavior should be adopted. The pretended behavior that the application should demonstrate while facing the attack vectors are the following:

- Use of bad characters in inputs: The application should filter the bad characters.
- Mismanagement of allocated memory: The application should better handle the management of allocated memory or it should be built with a memory-safe programming language.
- Format String Errors (FSE): The application should be linted, and the errors should be fixed. The final result should be the application not having these errors or mitigating them by filtering inputs.
- Integer Overflows (INT): The application should limit the input that the user is able to insert.
- Miss handling of errors: The application shouldn't disclose the underlying source code, instead it should give out an error with the specific context or it should give an error code.
- SQL injection: The application should sanitize its input before reaching the backend. The application should also use prepared statements.
- XSS: The application should sanitize its input before it's shown in the front end.
- Insecure direct object references (IDOR): The application should have restricted access controls to user-specific data. The application shouldn't also disclose the existence of that data. It should give a 404 error.
- Insecure deserialization: The application should verify the data given. If it encounters unverified data, it should discard it and demand verified data.
- File upload vulnerabilities: The application should sanitize the files uploaded and restrict access to the directory where these remain.

D. Testing

TST-01: Send multiple requests, ranging from 200 to 500 requests per second, and check if they are equally distributed between the available pods.

As previously mentioned, it's planned to implement Kubernetes as a deployment plan. This system helps in the management of containers, which includes an easy approach to apply a load balance mechanism. It's an important feature to test, since it increases the application availability (REQ-01, 02). In case of the failure of one pod, the system shall keep its normal operation.

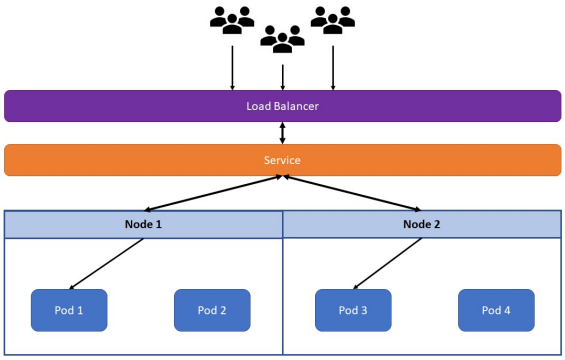

Fig. 1. Load balancer schema in a Kubernetes architecture

TST-02: Test all possible user interactions (HTTP Requests) and verify that all are being logged.

In terms of normal traffic to the main page of the web application, logs play an important role to prevent possible attacks, being DDoS or an address blocked by the system which will be marked and listed in the firewall black list. On the other hand, logs are also fundamental when doing transactions, disallowing an user to repudiate its actions (REQ-03, 17, 20).

TST-03: Do a Man-in-the-Middle(MiTM) attack and verify that the attacker is unable to read the content of each packet.

The communication between the server and the client it's done over the internet, so we have to assume that all packets being exchanged are susceptible to attacks like MiTM. Bearing this in mind, all packets should be encrypted (HTTPS), leaving the attacker with unintelligible data (REQ-05).

TST-04: Directly steal information from the database and verify that its content is unreadable.

Similar to TST-03, the test consists of stealing information from the database through some attack vector, once in the possession of the perpetrator the data should also be encrypted, in other words, the stolen content is incomprehensible, the developers should avoid the use of obsolete cryptographic algorithms (REQ-04).

TST-05: Delete partial/all data stored in database and check that it's restored with the help of backups.

It's good practice to do regular backups of the system's essential data, in this project the primary target is the database, the test consists in the database restoration, given a certain stored copy (REQ-06). At this stage, the developers shall, also, guarantee the integrity of each backup, avoiding the storage of a corrupted database which is unproductive.

TST-06: Insert invalid inputs and check if the application is having its correct behavior.

During the validation process, the application shall not change its normal operation, or be affected in anyway by an invalid input nor an attack (i.e. SQL Injection). For instance, if the user is filling the field related to the age, this application can only accept positive numbers (REQ-07, 18). The following snippet demonstrates this example:

DataAnnotations

```
[Range(1, int.MaxValue
, ErrorMessage = "Only positive number allowed")]
public int Age {get; set;}
```

Code

```
public ActionResult Create(Account acc) {
    if (acc.Age <= 0) {
        return BadRequest();
    }
}
```

TST-07: Conduct a DDoS attack with a source address from a black list, check if the policies in the firewall are effective and the protection against massive requests is successful.

This test serves to demonstrate the correct application of both WAF and DDoS implementations (REQ-01, 08, 19), hereupon, it is expected that the availability of the system will be increased. The botnets won't be able to make a connection to the web application, or their packets will be discarded, also, their source should be black listed.

TST-08: Test the authorization with two different accounts. Try to access the personal data of an user, other then the one currently authenticated.

With this test the developers need to guarantee that an anonymous user or an authenticated one can't have access to another's user personal data, such as, purchases, credit cards or address information. Usually and order has an unique identifier, which could be used to access via URL, in this experiment, beyond the restriction to another user data, the application shall not give any meaningful hint of the information existence (REQ-09, 13, 18).

TST-09: Send malware to the network (controlled environment) and verify if the malware control software detects the its presence.

In a controlled and secure environment the security experts shall test the quality of malware management software and implementation to avoid undesirable attacks that might compromise the availability or confidentiality, through data theft (REQ-10).

TST-10: Test the prevention of clickjacking, check that it's not possible to apply those attacks to this web application.

During the development of web application some of the aspects related to security might pass unnoticed, using clickjacking as an example, it's possible to mirror the aspect and actions of this project leading the user to believe that he is interacting with a legitimate website. The attacker takes advantage of this misconfiguration and hides invisible elements making the user to perform unwanted actions (REQ-11). The following line prevents this type of attack using Nginx:

```
add_header X-Frame-Options sameorigin always;
```

TST-11: Test the detection of vulnerabilities by launching outdated versions of software being used by the system and verify the associated flaw.

The security experts shall implement and test a framework destined to scan and manage vulnerabilities in the system running this application. With this test it's intended that a software (i.e. OpenVAS) is capable of detecting outdated or vulnerable systems, such as, container, virtual machines or software that can be used as an attack vector (REQ-12). Also, SonarSource should be implemented for continuous inspection of code quality and perform automatic reviews with static analysis of code to detect bugs and code smells.

TST-12: Test the web application functionalities and check the correct operation or exceptions associated.

This test aims the exploration of all possible paths in the code developed, mainly the ones related to user interaction. The purpose of this tests is to detect any flaws while handling exceptions and check that the logic of business implemented is in line with what was stipulated by the project manager. (REQ-13, 18, 19). It shall be used software to do unit tests, if provided, or by manual testing too.

TST-13: Test the cryptography of the tokens exchanged between the client and server, verify that is not possible do decipher the cookies.

In correlation with REQ-11 and REQ-14, the web application shall use cryptographically secure tokens (cookies) to avoid information leaks and also, to prevent against CSRF attacks. Even if the attacker captures the cookie it won't be possible to see the contents that it carries. They should be different in every HTTP request, being a GET, POST, PUT or DELETE, by doing this an attacker won't be able to perform CSFR attacks, which consists in the execution of unintended actions, on the part of a legitimate user, through a phishing scam. ASP.NET is an example of the implementation, because on each request a new cookie is generated, and an attacker won't be able to make a useful script with a cookie he doesn't know.

TST-14: Test the application login mechanism. Verify that after three attempts in between a certain period of time the system blocks for an hour.

With this test it's expected to prevent brute force attacks that might give access to an attacker. The application shall reject any new attempt at login, for a certain period of time, if the limit has been reached, this time will be increased if the user continuously keeps failing the login after the blocking time expires.

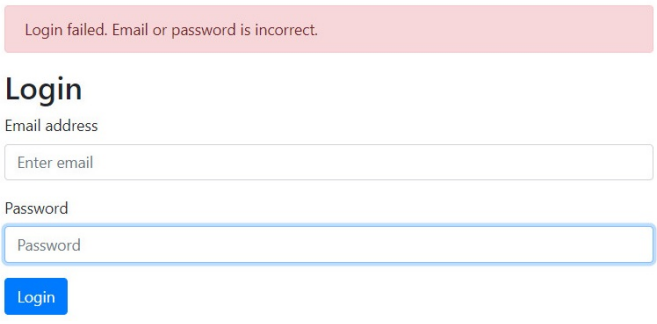

Fig. 2. Example of an error message after a failed login attempt

Also, it shouldn't be displayed any error or troubleshooting message that might give an information about existence of an

account under a certain email or phone number, that could lead the perpetrator to improve its attack (REQ-15, 16, 19).

VI. Hazard/Threat Analysis

A. Hazard Log

A Hazard Log is a record used and maintained to track the progress on identified hazard analysis, risk assessments, reduction or acceptance. At this stage, the document should reference all the hazards, identifying their causes and consequences, possible safety measures taken to mitigate its repercussions and categorizing the probability of happening those events, as well as the associated severity level and risk rating.

- Denial of Service
- Unauthorized Access
- Data Deletion
- Data Disclosure
- Supply-Chain Attacks
- Storage Hardware Failure

This document is created after a preliminary hazard analysis and should be updated during the project's development.

B. Failure Modes and Effects Analysis

An analysis on predicted system behaviour, for specific system functions, if they're failing in ways represented by the following Failure Modes:

- No Function - The specified function doesn't work at all;
- Incorrect Function - The specified function is doing something other than what it is supposed to;
- Delayed Function - The specified function's timing is off - either taking too long or too little.

Starting from a correct functioning system, for each Failure Mode it is presented the effect of the function in failure, both for same system level and a higher system level - Higher system level is more abstract consequences, lower are more technical. It is attributed a category based on how impactful the function in failure is, as well as the method that can be used to detect the system is failing.

References

[1] Positive Technologies, "Business in the crosshairs: analyzing attack scenarios,", December 2021.

[2] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. 2007. Denial of service or denial of security?