




You've Been Hacked

An (Interactive) Course on Web Security

Paul Duplys

 @duplys

 duplys

 [linkedin.com/in/paulduplys/](https://www.linkedin.com/in/paulduplys/)

man slides

(Interactive) course on web security based on Carsten Eiler's book "You've Been Hacked".

Who is the audience? How can I use the book? How can I explore the app?

whoami

short intro/bio.

0x0: Preliminaries

Finding Vulnerabilities in Web Applications

In a nutshell, **to find vulnerabilities in your web application**, ...

1. ... test various values for parameters used by the web application and see what happens (conceptually similar to fuzzing)
2. ... check web application code for bugs that may lead to security vulnerabilities (typically missing checks of input values or missing countermeasures against certain types of attacks)

The [Open Web Application Security Project \(OWASP\)](#) maintains a list of Top 10 vulnerabilities in web applications.

Top 10 Web Application Security Risks

1. **Injection.** Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
2. **Broken Authentication.** Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.
3. **Sensitive Data Exposure.** Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
4. **XML External Entities (XXE).** Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
5. **Broken Access Control.** Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

Conventions

- ▶ Alice, Bob: legitimate users
- ▶ Eve: malicious user, attacker
- ▶ Server: web server running a web application
- ▶ Client: web browser (or computer running the web browser)

GitHub Repo

- ▶ <https://github.com/duplys/youve-been-hacked>
- ▶ Dockerfile & setup instructions
- ▶ Write-ups
- ▶ Code

Building Docker Image

- ▶ Running the demo web application in a Docker container is the easiest way to get started
- ▶ Docker dir contains `Dockerfile` for the vulnerable web application
- ▶ Build the container using `make build` or `docker-compose`

Starting Docker Containers

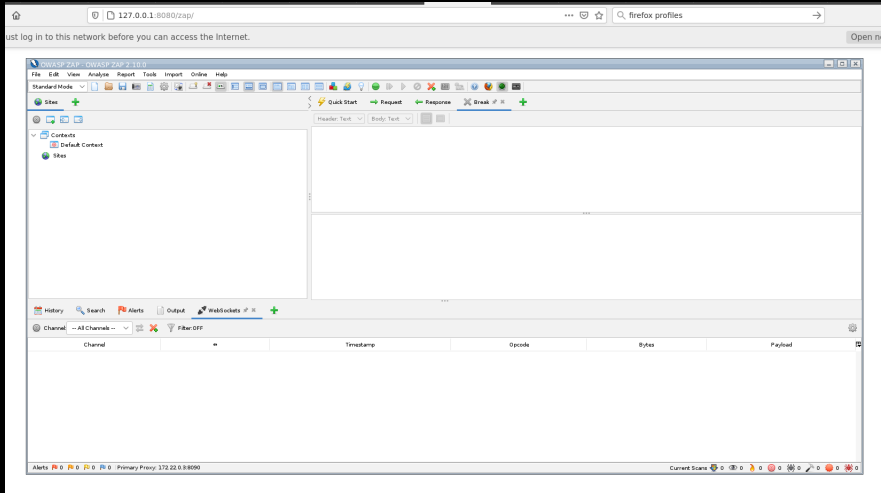
- ▶ You'll need an empty directory `tmp` in the `Docker` dir
- ▶ In `Docker` dir, run `$ docker-compose up`

Setting up ZAP

You need to activate ZAProxy (ZAP), configure your browser to proxy via ZAP and import the public ZAP Root certificate (see <https://www.zaproxy.org/docs/docker/webswing/> for details). Do the following steps:

- ▶ Go to `http://127.0.0.1:8080/zap/`
- ▶ You'll see how the ZAP Web UI starts
- ▶ Start a ZAP session, choose "don't want to persist"
- ▶ Select "Update All" in the "Manage Add-ons" window

Setting up ZAP



Setting up ZAP

- ▶ Go to "Tools" → "Options" → "Dynamic SSL Certificates" → "Save"
- ▶ Save ZAP certificate on your host and import it into your browser.
- ▶ Read off the ZAP's IP address and port number at the bottom of ZAP's window
- ▶ Configure your web browser to use that IP/port as proxy

Accessing the Vulnerable Web App through ZAP

- ▶ Look up the vulnerable app container IP address (for docker-compose)
- ▶ Run `docker container inspect docker_vulnapp_1` and look for `IPAddress:` under `Networks:`
- ▶ If the container's IP address on your machine is `x.y.z.w`, you can access the vulnerable web app under `http://x.y.z.w/daten/kapitel1.html`

Cleaning Up

- ▶ Run `$ docker-compose down`

Fahrplan

- ▶ Get to know your target
- ▶ Test for attacks on web application's state
- ▶ Test for attacks on authentication
- ▶ Test for cross-site-scripting (XSS)
- ▶ Test for SQL injection
- ▶ Test for other injection-based vulnerabilities
- ▶ Test for attacks on file operations
- ▶ Test for buffer overflows, format strings and integer bugs
- ▶ Test for architectural attacks
- ▶ Test for attacks on the web server

0x3: Stateful Attacks

URL Jumping

- ▶ Eve abuses the intended sequence of web pages. This gives her access to functions that by design must be unavailable at that point in time.

URL Jumping: An Example

In a typical web shop, the intended sequence of transactions – i.e., pages that a shop's customer visits – could look something like this:

1. Search for a suitable product at `www.shop.xyz/catalogue.php`
2. add an article to the shopping cart at `www.shop.xyz/add.php`
3. verify the customer order at `www.shop.xyz/order.php`
4. provide shipping information (customer's postal address) at `www.shop.xyz/shipping.php`
5. enter credit card information at `www.shop.xyz/payment.php`
6. checkout & finish the order at `www.shop.xyz/checkout.php`

URL Jumping: An Example

- ▶ If the attacker can jump from step 4 (`www.shop.xyz/shipping.php`) directly to step 6 (`www.shop.xyz/checkout.php`), she might be able to order an item without paying for it
- ▶ Another example: the attacker could try to post a message to an online forum with creating an account
- ▶ In general, the attacker tries to identify URLs that must be called in a specific sequence and tries to manipulate this sequence

URL Jumping: Locating the Vulnerabilities

- ▶ Make a complete list of URLs in your web application and whether certain URLs must be called in a specific order
- ▶ Use information gathered during recon phase
- ▶ For every URL sequence:
 - ▶ Call the URLs in the intended order and document the parameters passed to the web application
 - ▶ Diverge from the intended sequence and see what happens. Are there any error messages (e.g., "You first must create a user") or are specific URL/pages not accessible?
 - ▶ If there is no error message and the URLs can be accessed in any order, check whether this represents an actual, exploitable vulnerability

Defense against URL Jumping

- ▶ URL jumping is a security issue only when the sequence in which URLs are accessed is relevant for the web application
- ▶ In that case, you need to check that the correct sequence is used, e.g., `checkout.php` would check if the user previously visited `payment.php` which, in turn, would check if the user previously visited `shipping.php`

Defending against URL Jumping

There are 3 options to identify a previously visited URL:

1. The URL (or the corresponding ID) is stored in a hidden input field or in an URL parameter
2. The URL (or the corresponding ID) is stored in a cookie
3. The URL, from which the user is allegedly coming, is compared to the Referer field value in the HTTP header

Defending against URL Jumping

- ▶ As shown previously, option 1 (hidden input field) and option 2 (cookie) are insecure as they can be easily manipulate by the attacker
- ▶ That leaves us with option 3 (Referer value)

An HTTP request could look like this:

```
Get /checkout.php HTTP/1.1
Host: www.shop.xyz
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X; de-de) [.]
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: de-de
Connection: keep-alive
Referer: https://www.shop.xyz/payment.php
```

Defending against URL Jumping

- ▶ The `Referer` value is automatically set by the web browser
- ▶ In case of a bookmark or a direct entry in the web browser's address field the `Referer` value does not exist in the HTTP header
- ▶ However, the `Referer` value can be omitted by the web browser (for privacy reasons), removed from the HTTP request by a proxy or manipulated by the attacker
- ▶ `Referer` value can be manipulated using a tool like ZAPProxy or using a dedicated client where the individual values of the HTTP header can be defined manually
- ▶ So `Referer` value is also insecure

Defending against URL Jumping

- ▶ In the end, the only effective defense against URL jumping – similar to other attacks – is to store state information on the server
- ▶ Specifically, the information about the visited URLs for each user can be stored temporarily in the web application (or persistently in a database)

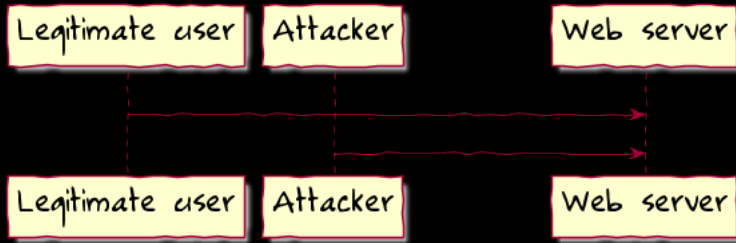
Session Hijacking

- ▶ Numerous attacks can be alleviated by storing state information on the server (instead of client)
- ▶ In that case, the web application needs a way to identify the user
- ▶ The notion of a *session* is used to associate state information (stored on the server) to a specific user
- ▶ The web application assigns each session a unique *session identifier* (session-ID)

Session Hijacking

- ▶ Session-ID is a number or a combination of numbers and letters that uniquely identifies a session
- ▶ Session-ID is assigned to the client (user) upon the first HTTP request to the web application
- ▶ Every subsequent request carries the session-ID so that the web application can identify the client (user) and retrieve the corresponding state information stored on the server

Session Hijacking



- ▶ Because the session-ID must be stored on the client, it can be manipulated or stolen
- ▶ An attack exploiting this vulnerability is referred to as *session hijacking*
- ▶ The attacker gets hold of the session-ID information to gain unauthorized access to the web application

Session Hijacking

```
Name=Alice  
userID=1234  
last=2021-08-24
```

- ▶ A special case of session hijacking is the so-called *authorization bypass*
- ▶ Example: web application identifies the client (user) based on a cookie containing the user name, the user ID and the date of the last session
- ▶ Is there are user with ID 1235?
- ▶ The attacker can change userID and use the manipulated cookie to take on the identity of another user

Finding Session Hijacking Vulnerabilities

- ▶ First, identify the session-ID in the web application, e.g., search for parameters whose names contain ID, TOKEN, SESSIONID, ID or similar strings. These parameters can be located in hidden input fields, URL parameters or in cookies.
- ▶ Once you identified a potential session-ID, change it and check whether you can impersonate another user

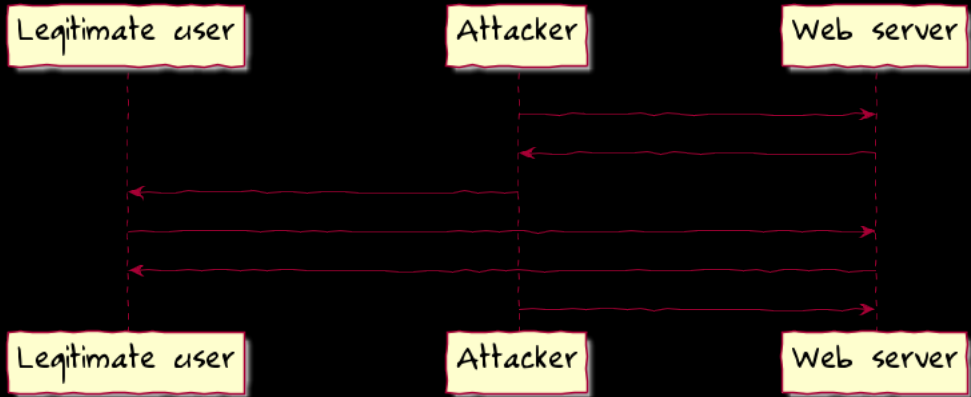
Defending against Session Hijacking

- ▶ Session hijacking requires a valid session-ID. The attacker has essentially 4 options to get hold of it:
 - ▶ Cross-site-scripting (XSS): XSS attacks are commonly used to steal cookies (more on this later). Make sure your web application has no XSS vulnerabilities. In addition, you can set `HTTPOnly` flag to prevent access to cookies from within JavaScript
 - ▶ Eavesdropping network traffic: if the attacker can read network traffic between the client and the web application, she can extract the session-ID. Ensure that your web application uses TLS (HTTPS).
 - ▶ Searching logfiles: if the attacker has access to the web server (or proxy) logfiles, she can search them for session-IDs that were transmitted via GET requests. Session-ID should therefore be transmitted in cookies or via POST requests.
 - ▶ Brute-force attack: the attacker can try to guess the format of your session-IDs and try out various (random) combinations until she finds a valid session-ID. You should use large random values as session-IDs.

Defending against Session Hijacking

- ▶ Performing session identification based on the combination of the session-ID and the IP address of the client makes session hijacking more difficult
- ▶ Instead of the IP address, you can also use e.g., the User-Agent header (i.e., user the client browser "model" as an additional identification factor)
- ▶ To reduce session hijacking-related risks, you should end the sessions after a specific time and make the corresponding session-IDs invalid
- ▶ This can be done using a "hard" time-out (e.g., every session ends after 30 minutes) or a "soft" time-out (e.g., session ends after 10 minutes of inactivity)

Session Fixation



- ▶ Instead of stealing or guessing the session-ID, the attacker attempts to fixate other user's session-ID
- ▶ Session fixation attacks are typically web based and rely on session identifiers being accepted from URL parameters or POST data

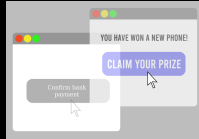
Detecting Session Fixation Vulnerabilities

- ▶ Use the information from the recon phase to identify session-IDs
- ▶ Call the web application with a session-ID that is already used, i.e., has been set by the web application
- ▶ Example: if the entry to the web application is `www.example.xyz` and the web application, in turn, returns `www.example.xyz/index.php?session=[valid-session-ID]`, call this URL directly
- ▶ If you can login and your session-ID is `[valid-session-ID]`, you've found a session fixation vulnerability
- ▶ If session-ID is transmitted in a cookie, use ZAPProxy to manipulate it

Defending against Session Fixation

- ▶ Make sure your web application generates a new, random session-ID each time a user logs in
- ▶ The same behaviour must be implemented when a new users signs up
- ▶ Some web applications accept arbitrary session-IDs, i.e., session-IDs that they have not generated
- ▶ This makes Eve's life easier because she doesn't need to obtain a valid session-ID
- ▶ Thus, your web application should only accept session-IDs that it actually generated
- ▶ Make sure it's impossible to reactivate expired sessions with known "old" session-IDs. If a session expired, a new session must be created with a new session-ID and a new state. Otherwise Eve can mount replay attacks.
- ▶ You should also leverage the information in HTTP header's `Referer` field. If the observed sequence of URLs doesn't match the expected sequence, either URL jumping is taking place or two clients (users) are using the same session-ID

Clickjacking



- ▶ Eve tricks Alice into clicking on something different from what Alice perceives and, as a result,
- ▶ This way, Eve can trick Alice into performing undesired actions by clicking on concealed links
- ▶ Clickjacking works because JavaScript allows to load a transparent layer over a web page and have the user's input affect that transparent layer without the user noticing
- ▶ For clickjacking to work, Alice must visit a page under Eve's control (because JavaScript is needed to create the transparent layer)

Clickjacking

Example:

- ▶ Alice receives an email with a link to a video about a news item on a page under Eve's control
- ▶ Eve overlays a product page on Amazon on top of the "play" button of the news video
- ▶ Alice clicks on "play" to start the video, but actually buys the product from Amazon
- ▶ Eve can only send a single click, so she relies on the fact that Alice is both logged into Amazon.com and has the 1-click ordering enabled

Clickjacking

- ▶ Technically, Eve embeds the target page in an iframe into the web page that will be displayed to Alice
- ▶ Eve reduces the size of the iframe such that only the target link is included; the rest of the target page is beyond the iframe
- ▶ Eve can configure the iframe to follow the mouse pointer. So regardless where Alice clicks, she will always click on the target link
- ▶ Finally, Eve adjusts the opacity of the `div` element that spans the iframe such that the iframe becomes invisible to Alice
- ▶ After tricking Alice into clicking the target link, Eve can bring the visible page into the foreground so that Alice will likely not notice the attack

Detecting Clickjacking Vulnerabilities

- ▶ Check whether your application contains pages where only authenticated users can perform certain actions
- ▶ Then, check whether these pages can be embedded in iframes

Defending against Clickjacking

- ▶ To defeat clickjacking, you must prevent relevant web pages of your web application from being embedded in an iframe
- ▶ A *framebuster* is a small JS script that prevents the web page from being displayed in a frame (however, HTML5 capabilities – loading a page in a frame with the `sandbox` attribute – allow Eve to bypass any framebusters)
- ▶ `X-Frame-Option` header can be used (by the web server) to specify preferred framing policy:
 - ▶ `deny` prevents any framing
 - ▶ `sameorigin` prevents framing by external sites
 - ▶ `allow-from origin` allows framing only by the specified site
- ▶ The `frame-ancestors` directive of the Content Security Policy can be used to allow or disallow embedding a web page in an iframe

Cross-Site Request Forgery

- ▶ Cross-Site Request Forgery (CSRF) attacks exploit web application's trust in Alice to perform unauthorized actions on her behalf
- ▶ Eve's goal is to trick Alice into unknowingly submitting an HTTP/web request to a web application that Alice has privileged access to (but Eve does not)
- ▶ Unlike Cross-Site Scripting (XSS) which exploits the trust a user has for a particular site, CSRF exploits the trust that a site has in a user's browser
- ▶

Cross-Site Request Forgery

- ▶ At risk are web applications that perform actions based on input from trusted and authenticated users without requiring the user to authorize the specific action
- ▶ For instance, a user who is authenticated by a cookie saved in the user's web browser could unknowingly send an HTTP request to a site that trusts the user and thereby cause an unwanted action

Cross-Site Request Forgery

Example 1:

- ▶ `www.server.xyz/app/index.php?action=logout` is used to log out users
- ▶ The web application identifies users through a session-ID stored in a cookie
- ▶ The cookie is automatically sent by the user's web browser
- ▶ The web application is an online forum where links can be stored in posts
- ▶ Eve publishes a post with the above link disguised by an
`more info here`
- ▶ Alice clicks on the link and is logged out

Cross-Site Request Forgery

Example 2:

- ▶ New user is added when `www.server.xyz/app/admin/adduser.php?name=NAME&pass=PWD` is called by an admin
- ▶ The web application identifies users through a session-ID stored in a cookie
- ▶ Eve prepares a malicious web page and tricks the admin to open this page
- ▶ The web page contains
``
- ▶ The (logged in via cookie) admin opens the malicious page
- ▶ Admin's web browser sends a GET request to load the image
- ▶ The URL to create a new user is called instead
- ▶ The cookie to authenticate the admin is automatically sent by her web browser

Cross-Site Request Forgery

Example 3:

- Instead of a GET request, the web application uses a form – and, thus, a POST request – to add new users:

```
<form method="post" action="http://www.server.xyz/app/admin">  
  Name: <input name="username"> <br>  
  Password: <input name="password"> <br>  
  <input type="submit" value="Add User">  
</form>
```

- The data sent to the application is in the POST request, i.e., it doesn't show in the URL
- Eve prepares a malicious web page and tricks the admin to open this page
- The web page contains a form that is auto-submitted upon opening the pages:

```
<form name="csrf" method="post" action="http://www.server.xyz/app/admin">  
  <input type="hidden" name="username" value="eve">  
  <input type="hidden" name="password" value="eve">
```

Cross-Site Request Forgery

Example 3 (cont'd):

- ▶ Eve prepares a malicious web page and tricks the admin to open this page
- ▶ The web page contains a form that is auto-submitted upon opening the pages:

```
<form name="csrf" method="post" action="http://www.server.x"
  <input type="hidden" name="username" value="eve">
  <input type="hidden" name="password" value="eve">
</form>
<script>document.CSRF.submit()</script>
```

- ▶ The (logged in via cookie) admin opens the malicious page
- ▶ The form is submitted automatically, as if the admin clicked on "Add User"
- ▶ The cookie to authenticate the admin is automatically sent by her web browser

Cross-Site Request Forgery

- ▶ Essentially, it is sufficient to trick an authorized user to send a maliciously crafted HTTP request
- ▶ As a result, CSRF is not limited to web pages
- ▶ Any document format that supports scripting can be exploited, e.g., HTML e-mails and multimedia files

Detecting CSRF Vulnerabilities

- ▶ A web application is susceptible to CSRF when it implements actions that can be triggered via a static URL (GET request) or a static POST request (like a static form)
- ▶ To prevent CSRF attacks, the corresponding URLs or forms (POST requests) contain the so-called *Anti-CSRF-Token*
- ▶ The *anti-CSRF-token* is a random value that Eve cannot guess
- ▶ The action associated with the GET or POST request is only performed if the token contains the correct value

Detecting CSRF Vulnerabilities

- ▶ Thus, you need to check whether all GET and POST requests perform an action that requires authorization contain a CSRF-token
- ▶ Check whether these requests contain parameters that change every time the request is Sensitive
- ▶ If any of these requests is static, i.e., doesn't change over time, the web application is vulnerable to CSRF

Defending against CSRF

- ▶ You must add a unique, random token for every relevant GET and POST request
- ▶ It is important that the token has sufficient entropy so that Eve cannot simply send multiple request with highly likely token values
- ▶ The web application must only perform actions when the correct token is supplied
- ▶ Then, Eve cannot prepare a valid request because she can't guess the random token value (she could learn the token value using XSS, though)
- ▶ For really important actions, you should add an additional password check

0x4: Attacks on Authentication

0x5: Cross-Site-Scripting (XSS)

0x6: SQL Injection

0x7: Other Injection-Based Vulnerabilities

0x8: Attacks on File Operations

0x9: Buffer Overflows, Format Strings and Integer Bugs

0xA: Architectural Attacks

0xB: Attacks on the Web Server

0xB: Misc

Bibliography