Never believe what comes from the user. Always validate client inputs.

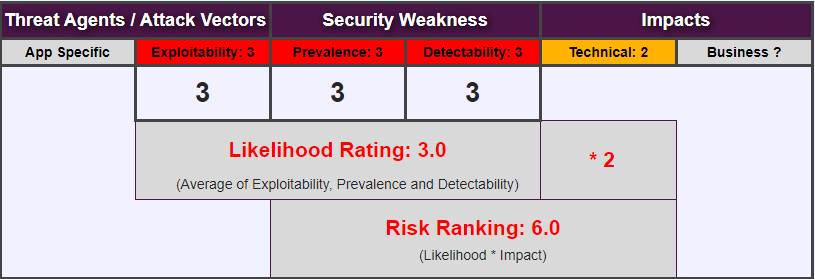
Using existing frameworks, a servlet technology is a good Idea, but is not all the way.

XSS goes by the same idea as SQL injection, where a hacker checks our vulnerabilities and ultimately injects malicious scripts.

Is a type of Injection attack.

Following OWASP, browsers have become better at handling XSS over the years but is still a huge thread following the OWASP description.

XSS is the second most prevalent issue in the OWASP Top 10 and is found in around two thirds of all applications.

* Exploitability: Easy to exploit
* Prevalence: It is seen very often
* Detectability: It is easy to detect
* 
* XSS has fallen in the OWASP ranking over the years, since the technical impact is considered moderate.

The browsers are getting better and better to protect against XSS attacks.

* Reflected XSS
* Stored XSS
* DOM XSS

We can train and experiment with XSS through the Juice shop demo.

* Show at exam.
  + SESSIONID stealing
    - HTTP is stateless.
    - Session tokens/cookies and therefore the SESSIONID are stateless
    - If the hackers acquire a session token and therefore the SESSIONID, whey will be able to impersonate and authenticate as us. It is the keys to the house.
  + Phishing
    - Malicious attacks through mail.
  + XSS in different forms

Handling Untrusted Data

* Handle all data is untrusted, especially user/client input.

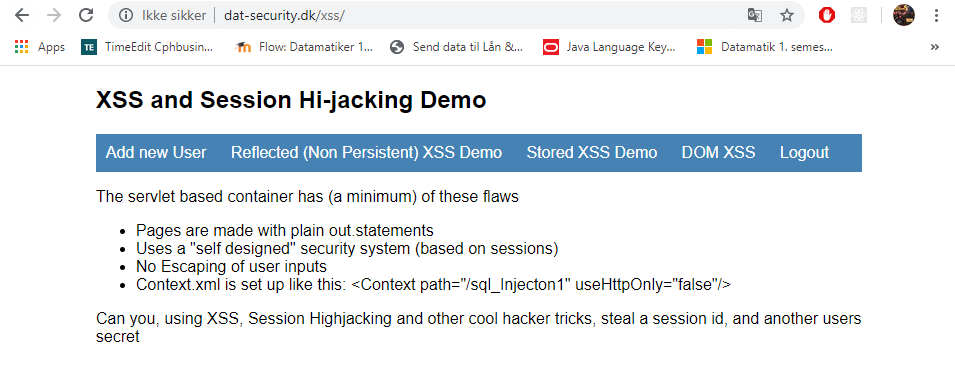
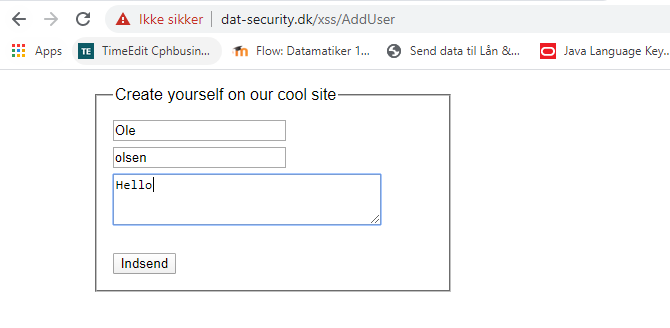
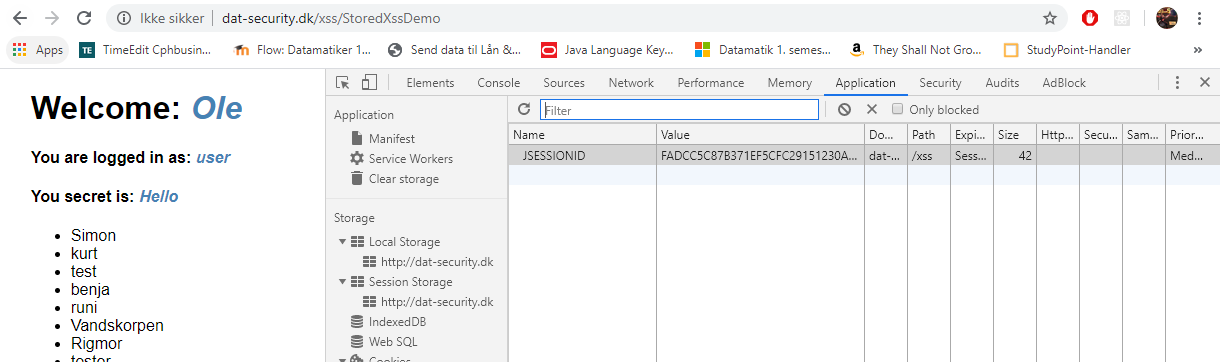
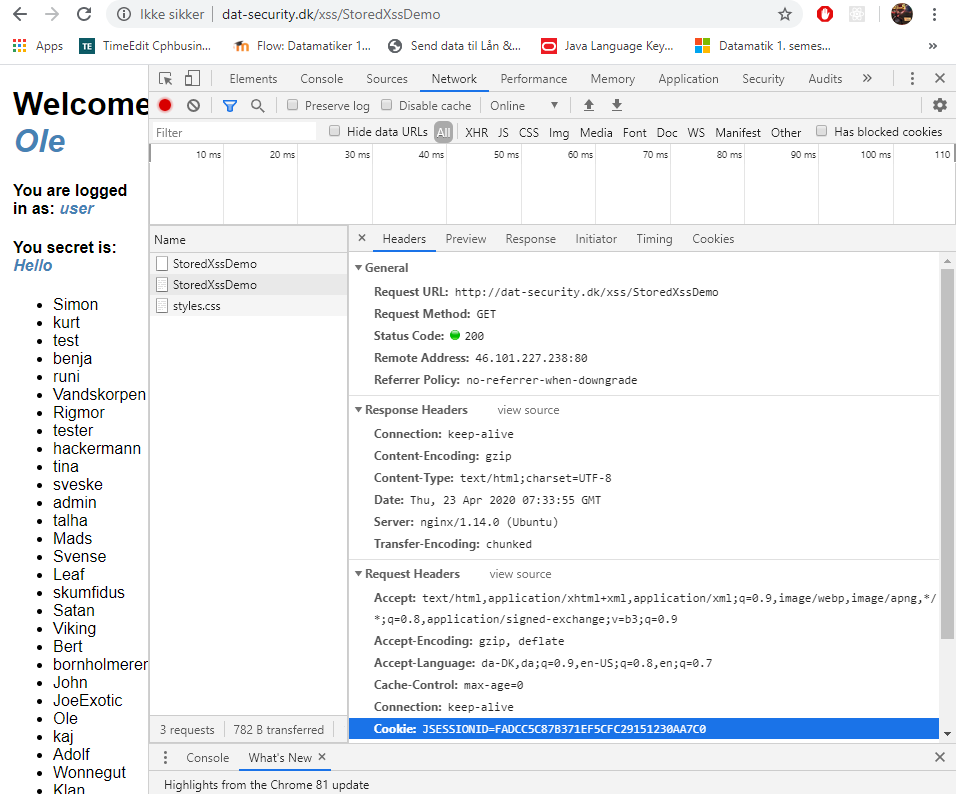
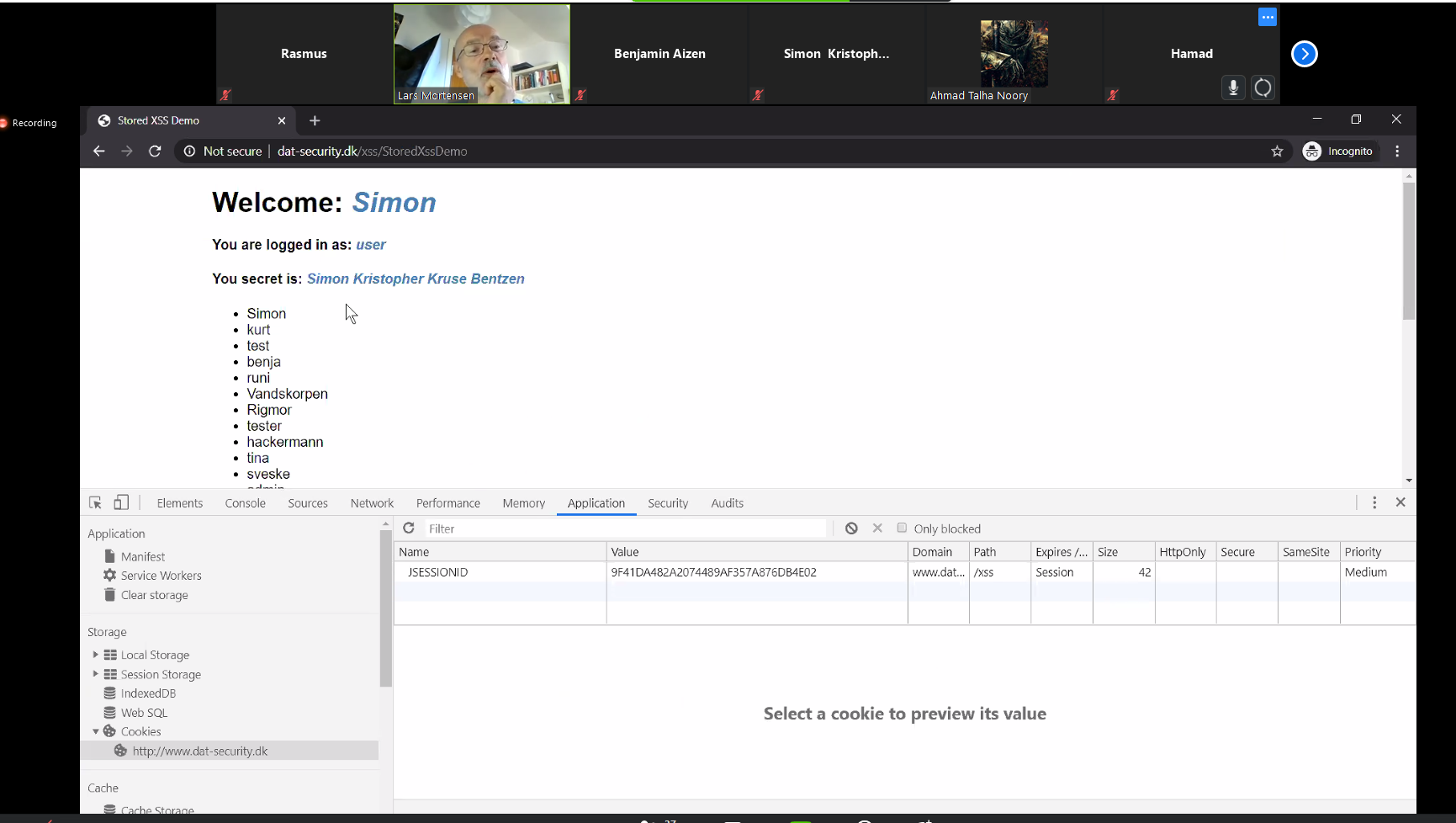
HTML Escaping

* Escape dangerous characters and tags form input fields.

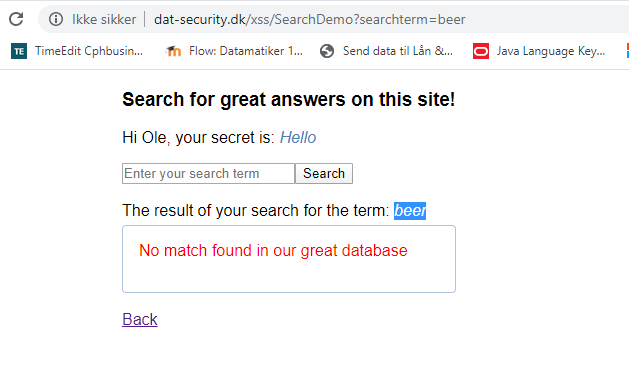
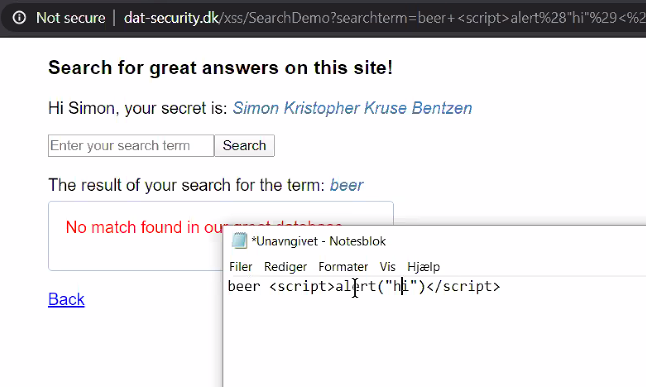
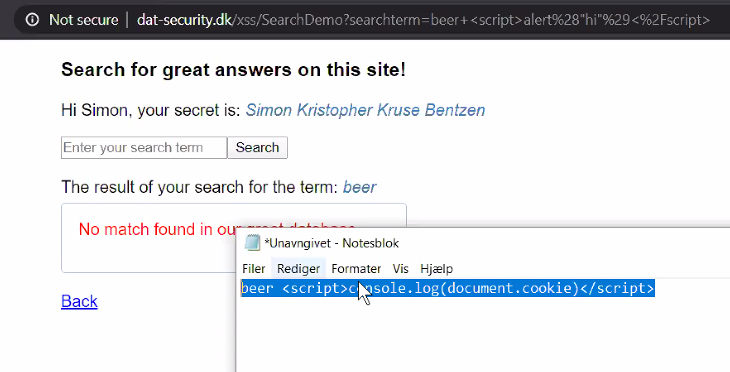
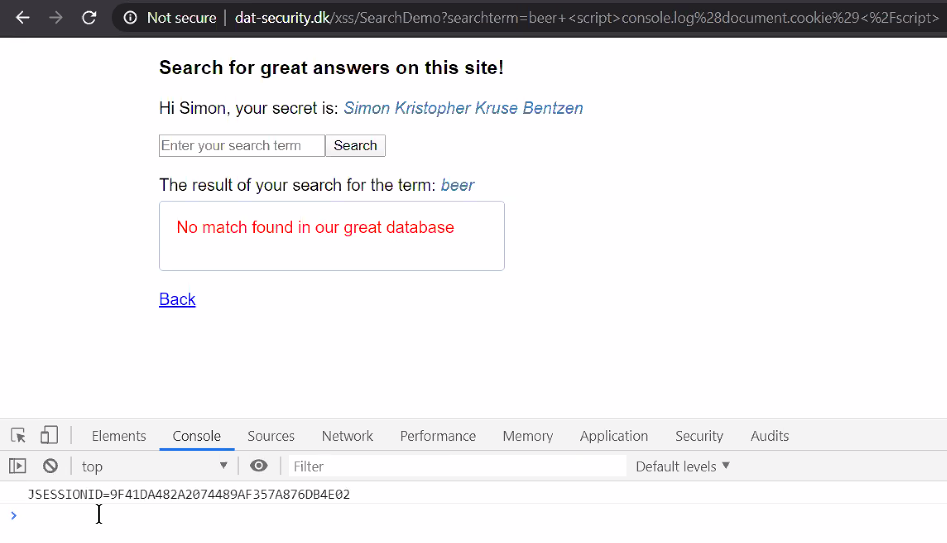
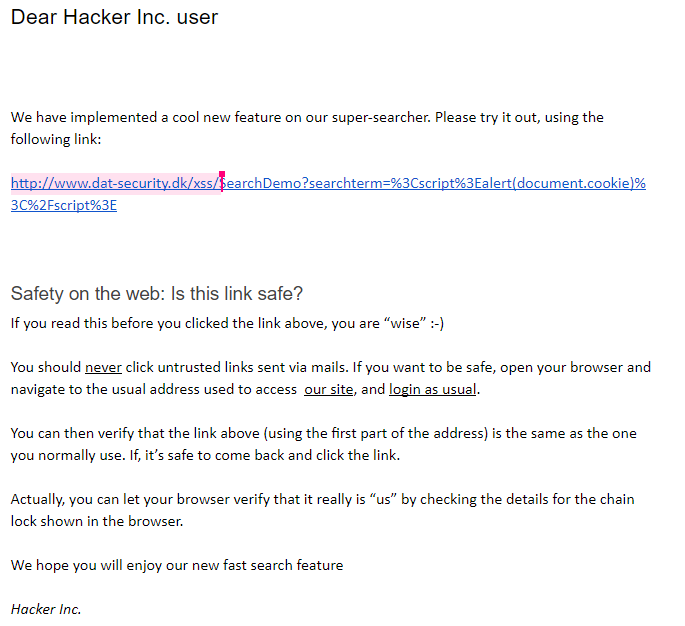
HTML Sanitizing

* Includes a white or blacklist of input that should or should not be allowed.

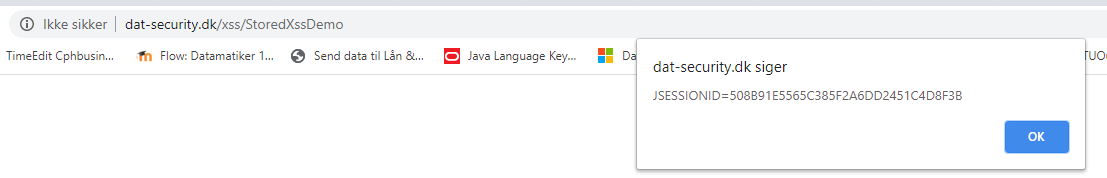
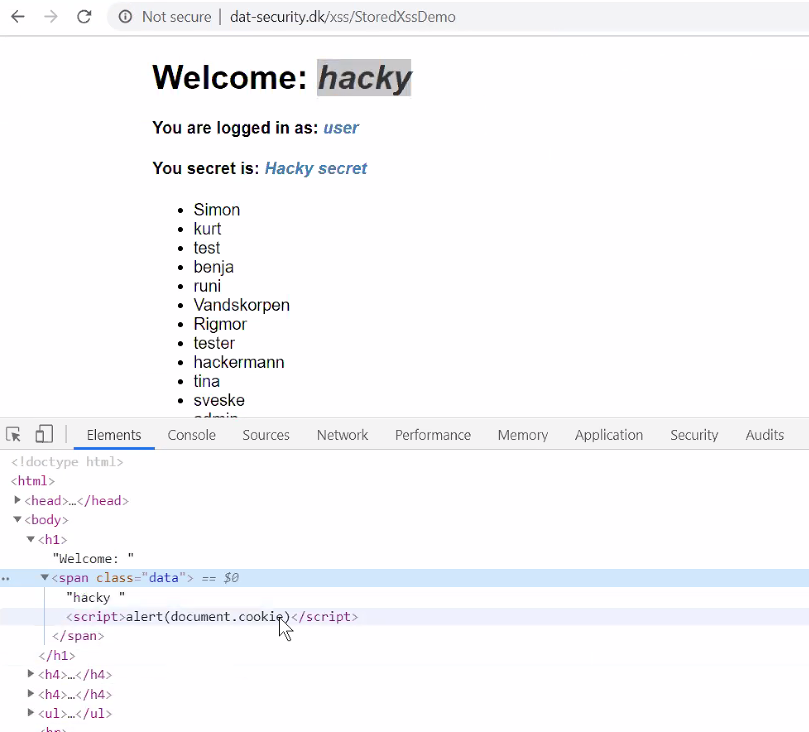
Live Demo

* Unsecure page  
  
* Creating a user  
  
* JSESSIONID is a cookie generated by Servlet containers like Tomcat and used for session management in J2EE web application for HTTP protocol.  
  
* Denne cookie bliver sent med for hver request til serveren. Denne token fortæller serveren hvem jeg er og at jeg har en gyldig session i gang med serveren.  
  
* Hvis en eller anden skulle få fat i dette JSESSIONID kan de identificere som mig og over tage min session. Session cookies er derfor super vigtige!  
  
* Hvis en hacker for fat i dette JSESSIONID, kan han overtage identiteten af brugeren og alt det brugeren kan gøre. Hackeren skal blot sætte session idet ind i sin egen browser.

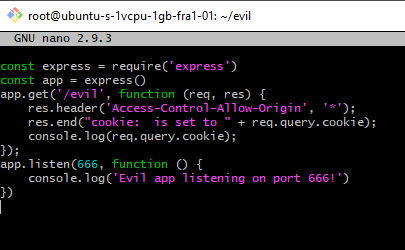
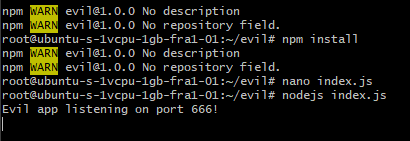
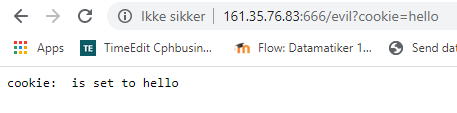
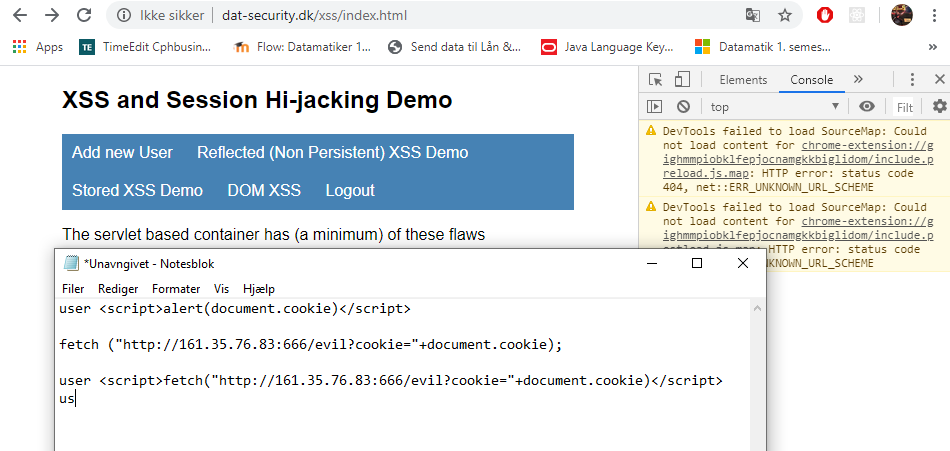
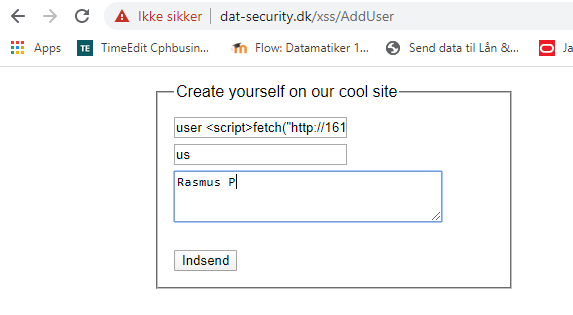
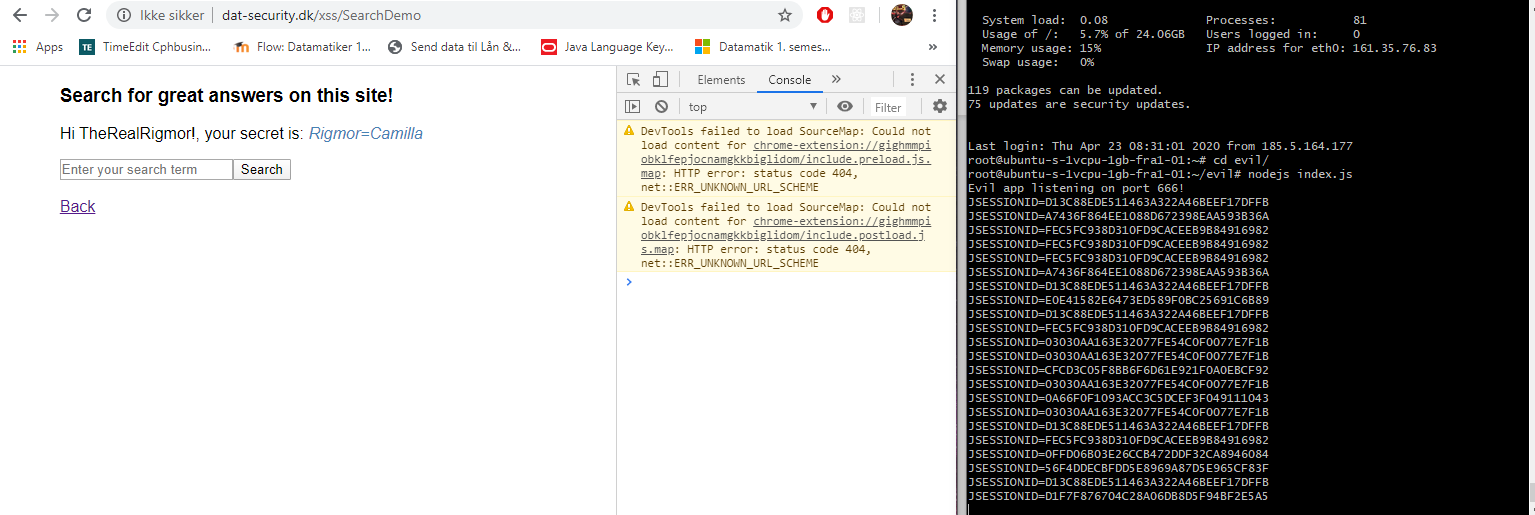
Reflected XSS and Phishing

* If we look at this example, we can see that whatever we write, appears in the in the page itself. Can we exploit this?  
  
* It turns out that we can write html <script> tags in the input field, and it remains unfiltered and also appears in the URL itself.  
  
* What if we try to log the cookie and SESSIONID?  
  
* Now we can log the browsers current SESSIONID.  
  
* But this is our own SESSIONID. Can we somehow access the SESSIONID of someone else?
* What if we send them a phishing mail!  
  

Persistent (stored) XSS

* What if we try to include some XSS in other users’ browsers through persistence?
* Can we create a user that includes a script tag?
* What if we could make the database store that a then load that script tag back into the client application, for an exploit to happen.
* Afterwards Other users will experience this problem, when trying to access the list with Hacky’s name. How strange!  
  
* However, if we look in the page elements for the name list, we will see that Hacky includes a script tag in his name, one that can be executed when other users access the page. For here he could for example log user tokens of other users a bit like we did before.  
  

Hacking attempt Contest

* 
* 
* 
* 
* 
* document.cookie="JSESSIONID=D13C88EDE511463A322A46BEEF17DFFB"  
  
* Fejlen her var at serveren hverken sanitized ordentligt for vores brug af farlige script tags, eller brugte HTTPOnly hvilket kunne have stoppet browserens JavaScript fra at tilgå cookien.

XSS Prevention Cheat Sheet

* RULE #0 - Never Insert Untrusted Data Except in Allowed Locations
* RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content
* RULE #2 - Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes
* RULE #3 - JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values
* RULE #3.1 - HTML escape JSON values in an HTML context and read the data with JSON.parse
* RULE #4 - CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values
* RULE #5 - URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values
* RULE #6 - Sanitize HTML Markup with a Library Designed for the Job
* RULE #7 - Avoid JavaScript URL's
* RULE #8 - Prevent DOM-based XSS
* Potentielle problemer er der hvor vi har behov for data fra brugeren.
* Dette gælder både for JavaScript, JSON, css, url
* Vi skal altid huske at sanitize vores data.
  + Sanitize betyder fjern alle farlige tegn
  + Dette forgår typisk ved en blacklist eller en whitelist.
    - Whitelist er normalt den bedste strategi, da vi bare skal tilkendegive hvad vi godt må indholdet.
  + Der kigges i whitelisten og fjerner alt der ikke må være der.
* You can Escape the dangerous characters with HTML entity encoding to prevent switching into any execution context.
  + Der er nogle tegn i html der er specielle.
  + Vi kan bruge encoding til at escape disse tegn fra en HTML context ved hjælp af encoding.
* Vi bør altid sanitize med et library der er designet og beregnet til jobbet.
  + Det er en rigtig dårlig ide at opbygge sit eget.
  + Hvis vi selv laver saitizer pakker, kan vi løve ind i mange sikkerhedsproblemer og uhensigtsmæssigheder.
  + Lad vær med at tro at du kan gøre det bedre end de professionelle kan. OWASP f.eks. Vi skal ikke opfinde løsninger selv.
  + Sanitizeren skal også altid være opdateret.
* Brug altid HTTPOnly cookie flag.
  + Mere eller mindre default i alle servere i dag.
  + Dette flag gør at det kun er http der har adgang til vores cookie, og ikke noget JavaScript der ville kunne eksploitere den. The browser will not reveal the cookie.
* Hvis der bruges sådan noget som dangerouslySetInnerHTML skal man være sikker på at det er begrænset til allowed locations.
* React som framework, beskytter langt hen ad vejen mod XSS og rydder også selv op i den eksekverede kode. Dog kan untrusted data stadig være et problem på serveren, hvor denne data stadig kul kunne befinde sig på databasen. Hvis man så en anden gang skifter framework til et der ikke beskytter mod dette, vil disse persisterede angrebsdata være en potentiel trussel.

Burg OWASP Juice Shop cheats heat til eksamensdemonstration

* Det er vigtigt bade at kunne vise XSS, men også at kunne fortælle om hvordan det forgår og hvordan man kan forhindre det.