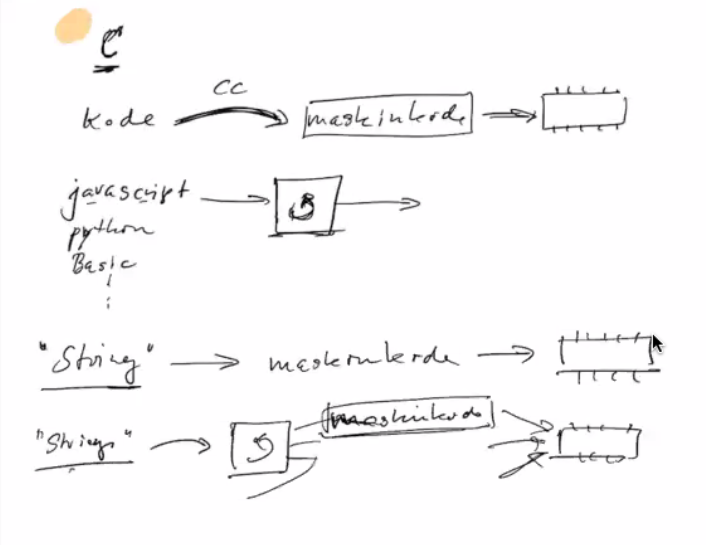
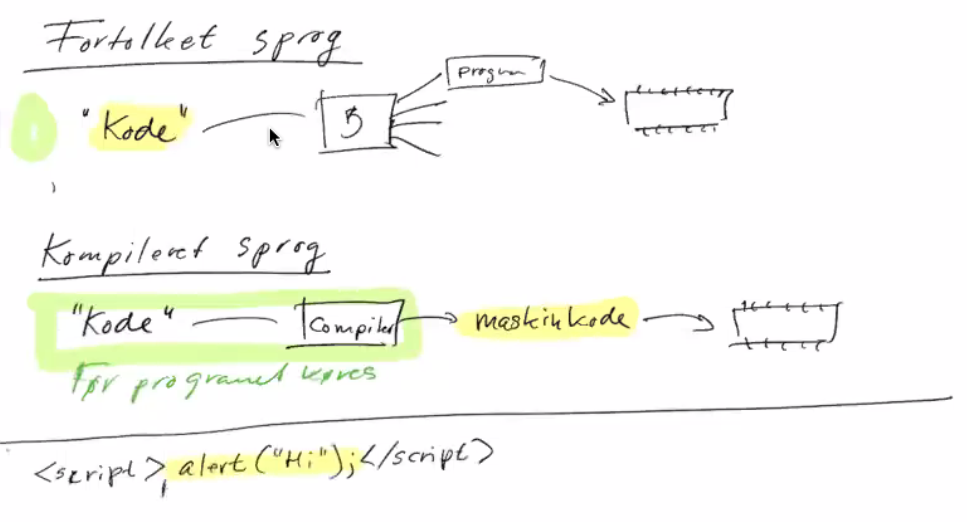
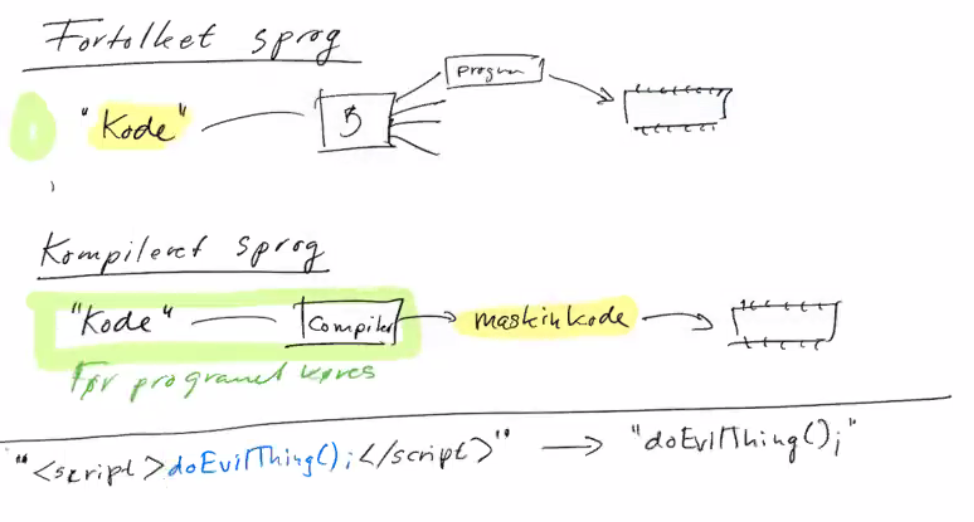
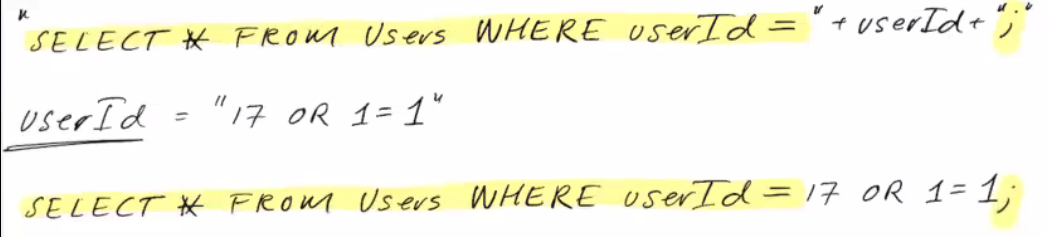
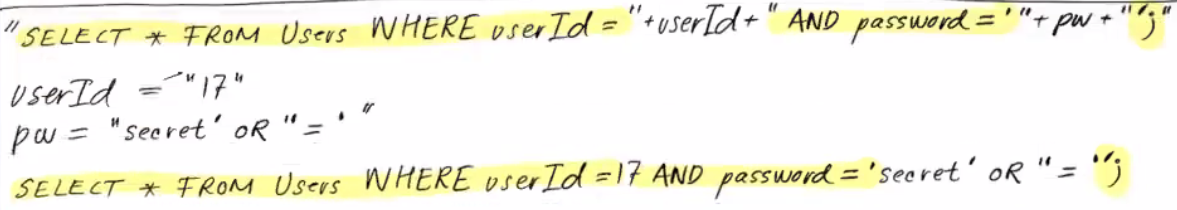
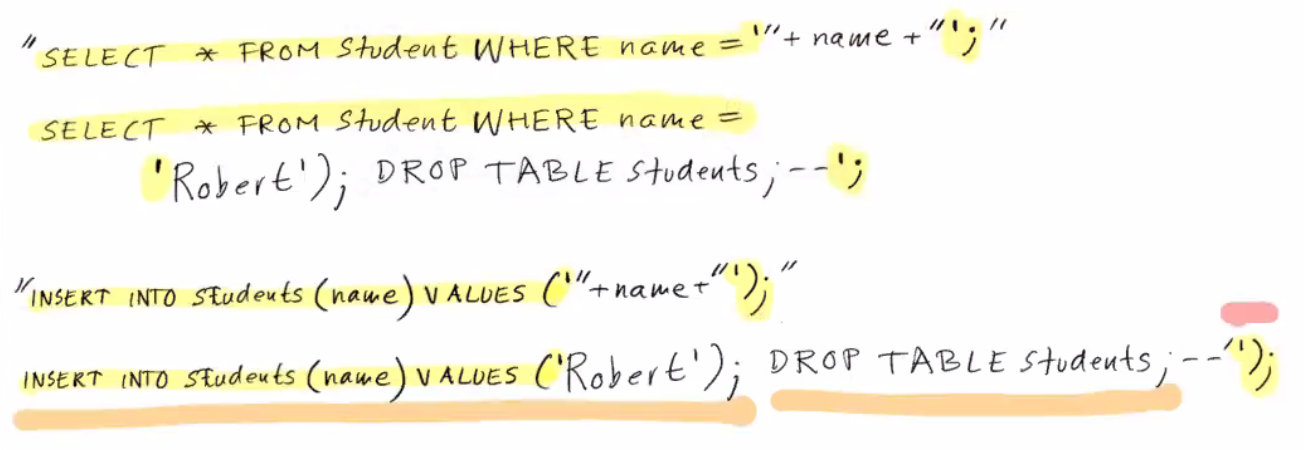
Injections

* A lot more than just SQL injections.
* Once we are talking about interpreted languages, there is the risk of injection.
* Ever time the interpreted language interpreters code, can we exploit the interpretation, all the way down on a machine code processor level.
  + Forskel på compile og fortolkede sprog:
    - I compile sprog bliver koden kært igennem en compiler til noget maskine kode der kan køre direkte på processoren. Køre før man køre programmet. Når koden køres gennem en compiler, laves det om til maskine kode.
    - Hvis man tager noget JavaScript, har man en JavaScript engine, der læster koden step for step, og udføre de steps hver gang. Her vil der være en fortolker der tager linje for linje og siger, hvad skal der ske. Fortolkeren laver programmet om til maskine kode i små bider, genne nogle fore-kompilerede programmer. Køre mens man køre programmet. Fortolkeren vælger for hver at kodelinjerne, hvilket program der skal køre på processoren. Dette gøres runtime. Therefore, we need to make sure that no one can change our code when working with interpreted languages.
    - Injektion sårbarheden ligger i at koden i et fortolket sprog, kan ændres inden vi køre de enkelte fore-kompilerede programmer. Dette er ikke muligt i et compilesprog, da compileren tjekker at koden er i orden inden den køre, hvorved der ikke længere kan fortages ændringer af koden. Fordi koden kan byttes om på inden der kompiler gennem de fore-kompilerede programmer, er fortolkede sprog sårbare over for at hackere kan lave ændringer i de brugte tags inden koden køres.
    - 
    - 
    - 
    - SQL queries are interpreted and are therefore vulnerable to injection attacks.

Injection is number one in the OWASP top 10 list.

* JavaScript (XXS cross site scripting) and SQL injections are among the most venerable interpreted languages due to their widespread use.

SQL injection

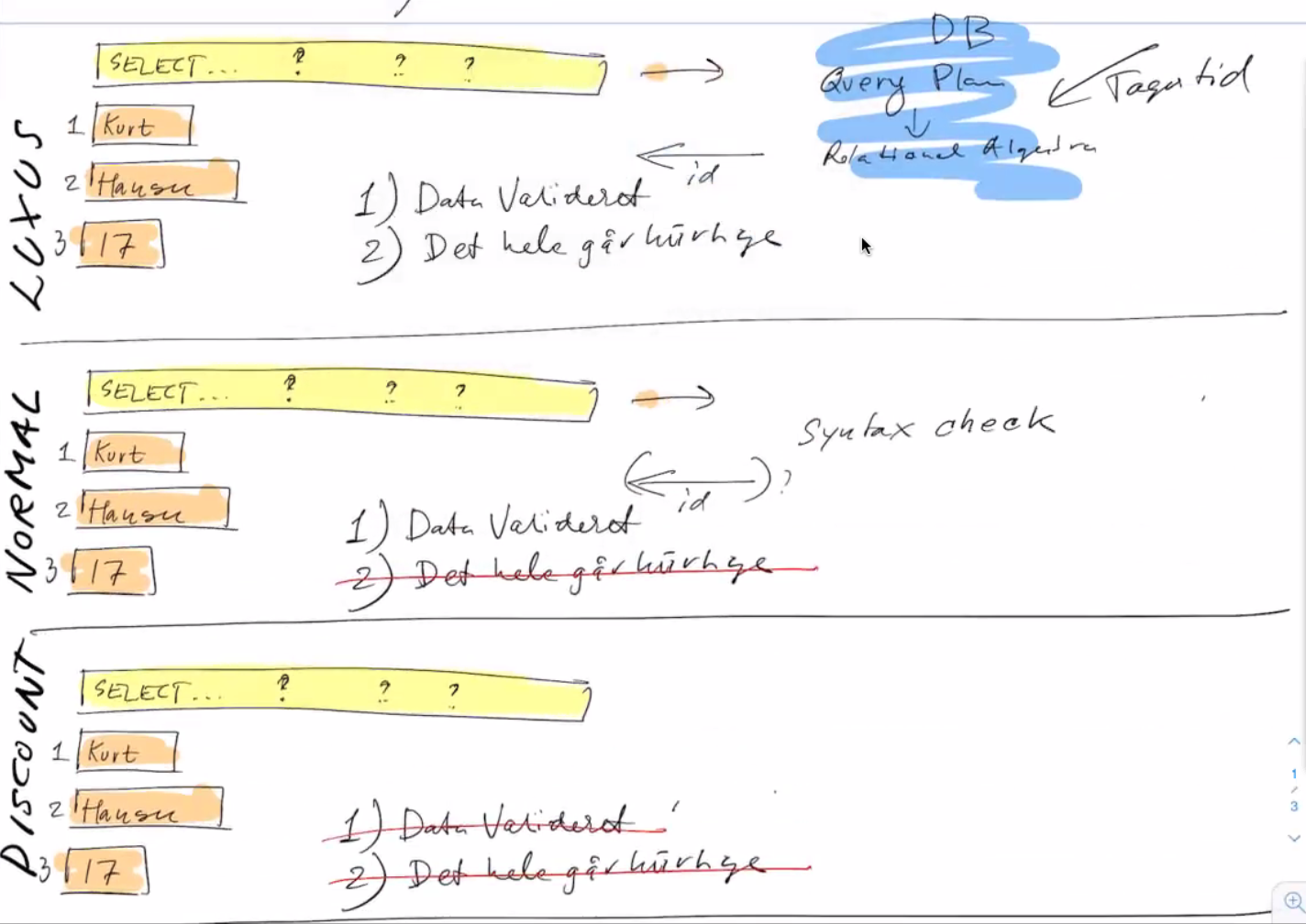
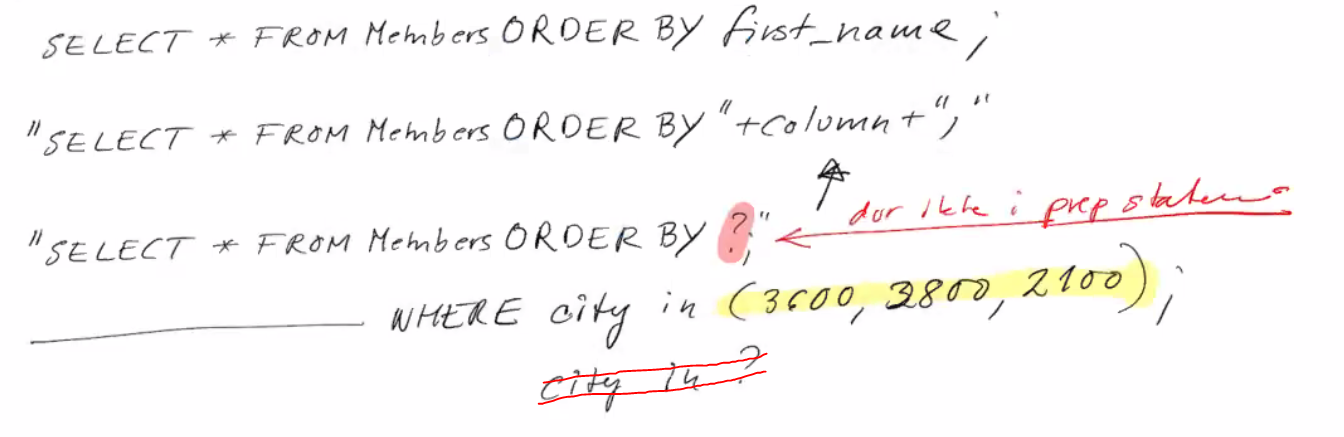
* SQL injection based on 1 = 1 is always True.
  + We can use this to our advantage by, knowing that whenever 1 = 1 we will always get a true condition.
  + Example of this is where we are asked to give a user-id, but instead of just giving I at number and guess, we can give it a condition saying, ‘number or 1 = 1’. The SQL above is valid and will return ALL rows from the "Users" table, since OR 1=1 is always TRUE. An illustration of this can be seen down below.
  + 
* SQL injection based on ‘’ = ‘’ is always True.
  + Like the example above, we can also use other conditional statements like ‘’ = ‘’ to get the SQL query to deliver information to us. The SQL above is valid and will return all rows from the "Users" table, since OR ""="" is always TRUE.
  + 
* Down below is an example of an SQL injection attack illustrated, where batched SQL statements and an -- outcomment of the last ending tag.
  + It is in most cases important to make sure that the SQL query is correct in its syntax since it possible might not return anything if it fails to execute. That is why we outcomment the last part of the query, to make sure that the inject query ends as it should.
* 

In SQL queries AND binds stronger than OR, which makes it so that our AND condition that we provide the query for injection will overrule the OR and ultimately determine what is returned to us.

How to find DB object names.

* SQL Exceptions
  + As hackers we can use the exceptions that are thrown and forwarded to us, to determine the very syntax or technologies used. Worst would almost be to get a snippet of the SQL used as well as all the table and object names.
  + Here we want to never forward SQL exceptions to our client. We don’t want to give the hacker an idea of what is going on behind down in the SQL.
* Guessing
  + By trying and testing different SQL code in a user input, we can see if the server’s backend won’t supply us with some SQL related information, exceptions or an idea of whether we are getting closer to something or not. We can also try some of the more popular SQL injection attempts to see if there should be any typical flaws that we can exploit.
* Insider knowledge
  + Former coworkers are also a thread to injection attacks since they have inside knowledge of the technologies, workarounds or queries used and hoe they may be exploited. This is obviously also a vulnerability and a possible attack vector.

How to prevent SQL injection.

* Manual formatting is not good
  + Doing our formatting manually is both labor intensive, and leaves room for incompletion. Manual formatting can accidently ruin our data, by formatting it before we execute the SQL query. An example is when someone has an ‘ in their name, and we have manually formatted the SQL to not accept ‘ since we don’t what SQL injections changing our query condition. Because we can ruin our data like this or leave even bigger holes vulnerable.
* Escaping
  + Escape sequences are used within an SQL statement to tell the driver that the escaped part of the SQL string should be handled differently. When the JDBC driver processes the escaped part of an SQL string, it translates that part of the string into SQL code that SQL Server understands. We can use escaping to get rid of/escape different characters within our SQL string. An escape sequence is a sequence of characters that does not represent itself when used inside a character or string literal.
* Parameterized queries
  + Prepared statements
    - Prepared statements are about preparing a query with unspecified variables called parameters which are typically marked with ‘?’. These unspecified variables will stand in for the placeholders in the query. The idea of a native prepared statement is smart and simple: the query and the data are sent to the server separated from each other, and thus there is no chance for them to interfere. Which makes SQL injection outright impossible. The queries can also.
    - 
    - It almost always pays off to use a prepared statement if you can. The main and most essential benefit of prepared statements is the elimination of all the dangers of manual formatting.
    - 
  + Works for static queries
    - Static SQL is SQL statements in an application that do not change at runtime and, therefore, can be hard-coded into the application. If your query is completely static, and doesn't contain any variable in it, there is no need to use a prepared statement.
* Placeholders
  + For dynamic queries
    - When dealing with dynamic queries we can never be sure that the variables contain valid values or not. Here the placeholders make sure that our statement contains the values needed to execute the query, and that they are valid values. We have to implement placeholders for every dynamic value or type. 
  + For Arrays
    - There are also placeholders for arrays by using the type SET. The special SET type consists for a comma separated identifier=string value pairs.
  + Whitelisting
    - Man kan lave en blacklist over statements der ikke er tilladte.
    - Man kunne derimod også lave en whitelist over de statements der er tilladte, og på den måde sikre sig mod uforudsete statements. Nogle man har godkendt på forhånd i modsætning til nogen man har afvist.
    - Hvis noget bliver afvist, kan man logge det samt det statement der blev forsøgt kørt.

Placeholders

* Identifier placeholder (single identifier)
* Identifier list (comma separated identifiers)
* Integer list (comma separated integers)
* Strings list (commas separated strings)
* The special SET type consists for a comma separated identifier=string value pairs.
* And many more types ....

Why manual formatting is bad

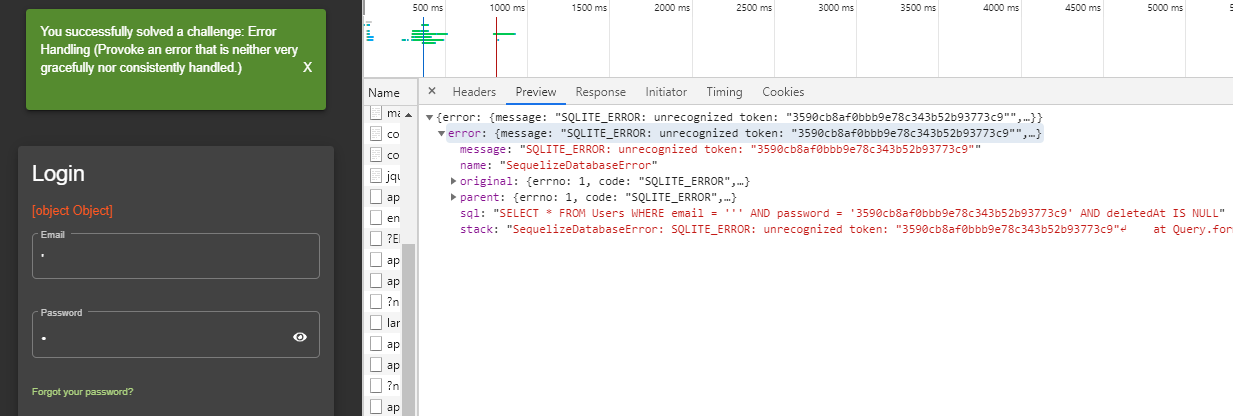
* Manual formatting can be incomplete
* Manual formatting can be applied to the wrong literal
* Manual formatting is essentially a non-obligatory measure.
* Manual formatting can be separated from the actual query execution by a considerable distance.

Prepared statements

* Prepare: An SQL statement template is created and sent to the database. Certain values are left unspecified, called parameters ("?"). Example: INSERT INTO MyGuests VALUES(?, ?, ?)
* The database parses, compiles, and performs query optimization on the SQL statement template. Stores the result without executing it
* Execute: At a later time, the application binds the values to the parameters, and the database executes the statement. The application may execute the statement as many times as it wants with
* different values § Can be emulated for certain dbs.

Example in OWASP Juice Shop:



If we type username: ‘ and pass: ‘, we can provoke an SQL error that we can later find in the preview section under networks.  


Using this we can see the syntax of the SQL used, and can thereby figure out how to cheat it.  
Lets use the 1=1 condition to login as the first user in the active table. (We could also use ‘ or ‘’=’’;)

sql: "SELECT \* FROM Users WHERE email = ' ' or 1=1; ~~' AND password = '3590cb8af0bbb9e78c343b52b93773c9'~~”  
