Security I
Secure sockets and SQL Injection

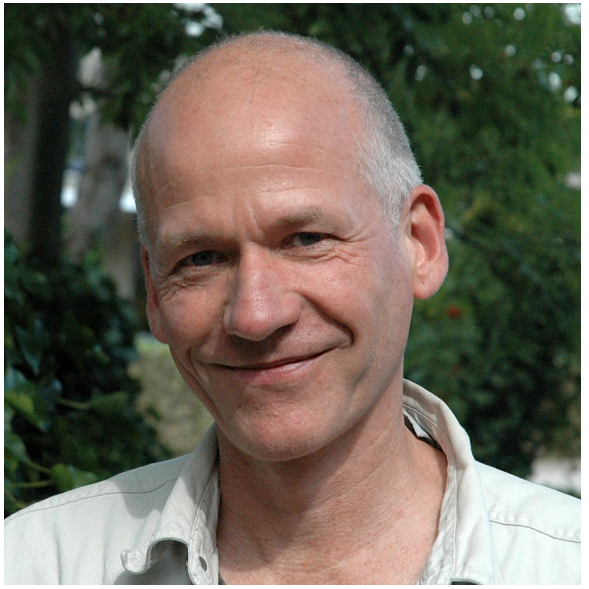
Friday Nov 7th, 8.15-12.00

Niels Jørgensen

Introduction to Security

Three course days on security

- Friday Nov 7th 8-12
- Monday Nov 10th 8-16
- Assignment 5 hand-in Nov 10th 23.55
- Wednesday Nov 12th 8-12



Niels Jørgensen
associate professer (C.Sc)

- teach and research it security
- recent history of cryptography

1) Secure sockets

Secure sockets provides
- confidentiality
- integrity
- authentication

Purpose of topic is also to present
symmetric and asymmetric encryption
- for some students, recapitulation
- for some students, introduction

Secure sockets protocol name
- TLS (actually since 1999)
- SSL (the original protocol form the 1990s)
- today's discussion is about SSL ('essential secure sockets')

2) SQL injection

SQL injection is -

- injection of malicious SQL code into a database
- malicious code to be executed by the database

In our example user interface, the user inputs -

BIO-101’ --

- this causes SQL code to be executed
  which displays the course BIO-101
- even though the user interface was designed to censor this course

The coming course days

2. Password-based authentication

Cryptographic hashing of users' passwords

How to slow down brute-force attacks

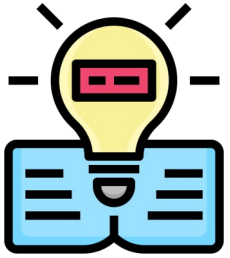
(but retain fast login of legitimate users)?

3. Secure sockets (TSL) (revisited)

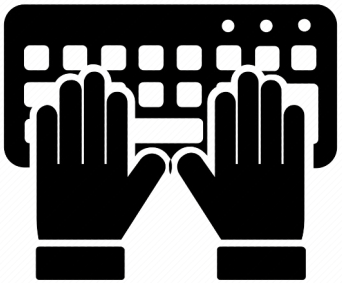
"Forward secrecy"

"Post-compromise security"

Approach:
theory + hands-on



A theoretical topic
- TSL
- symmetric and asymmetric encryption



Two practical, hands-on topics
- SQL injection
- Password hashing
- you will be asked to make modifications to programs downloaded from moodle
- password program may be used in your project

Assignment hand-in Monday 23.55
- work on assignment Monday afternoon
- assignment solutions to be discussed Wednesday

Exam questions for today (suggested, not final)

How does SSL do key agreement?

An example malicious SQL injection

Protection against SQL injection using stored functions

Which are the most important threats?

Threats to break encryption?
(such as SSL encryption)
• ?

Threats to inject macilious code?
(such as in SQL injection)
• ?

Threats to guess user’s passwords?
• ?

Which are the most important threats?

Threats to break encryption?
(such as TSL encryption)
- not a threat if encryption implemented correctly

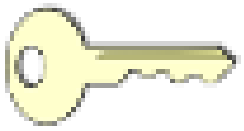
Threats to inject macilious code?
(such as in SQL injection)
- were important threats
- of type “user supposed to enter data only, but user actually enters program parts”

Threats to guess user’s passwords?
- very important threats

Today’s plan

1. Secure sockets
2. SQL Injection background - user friendliness and security
3. The SQL-Injection-Frontend
4. SQL injection attacks
5. How to protect against SQL injection attacks?

Symmetric encryption (essentials)



Symmetric encryption uses a single encryption key
Aka “Secret key” or “shared key” encryption

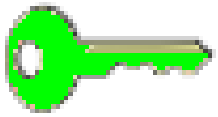
Challenge:

How do the two parties 'manage' the symmetric key
• ie. exchange or agree on a symmetric key?

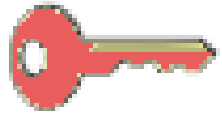
Before the invention of asymmetric encryption,
they would need “another channel” to exchange a key
ie., meet physically

Asymmetric encryption (essentials)

Asymmetric encryption uses two keys



a public key



a private key (private = secret)

Asymmetric encryption is crucial for
- key management
  - sender encrypts s. key using public key of receiver
  - receiver decryptes using his/her private key
- also crucial for digital signatures

Public and private keys come in pairs
- a text encrypted with a public key
  can only be decrypted with the corresponding private key
- and vice versa
- clever but encryption/decryption is slow

(In the meantime, you may download + install)

Instructions -

- are for console-based build/run
- and assume dotnet is already installed
  for which uou may use Henrik's instructional video "dotnet":
  URL: https://www.youtube.com/watch?v=QosnAcJbF8g

1 Open a console / terminal.

2 Go to your over-all C# project directory. Your individual C# projects should be organized as subdirectories of the project directory.

3 Create project: Type "dotnet new console -n SQL-Injection-Frontend". This will determine the directory name of the project; you are free to choose a different name

4 Go to the new project directory that you just created (ie., SQL-Injection-Frontend). Stay in this directory for the remaining steps.

5 Download the two source files to the new project directory. The downloaded source Program.cs should overwrite the existing file of that name in the directory.

6 Add library: type "dotnet add package Npgsql".

7 Finally, to build and run, type "dotnet run SQL-Injection-Frontend".

Secure sockets: a protocol

A protocol =
- rules for communication between two parties

A protocol is defined by a text
- often written a committee

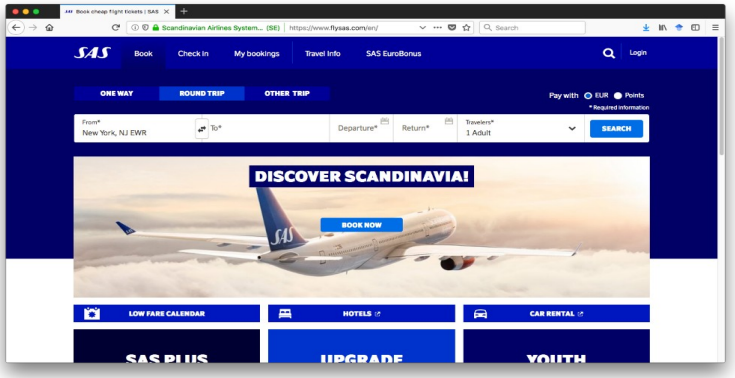
Text defines ..
- client says “Hello”
- serves responds ..

Most Internet protocols are quite long
- RFC 8446 (TSL v. 1.3) is 160 pages

Secure sockets are implemented by different software libraries
- OpenSSL, Microsoft’s ‘Schannel SSP’, ..
- client and server may use different implementations

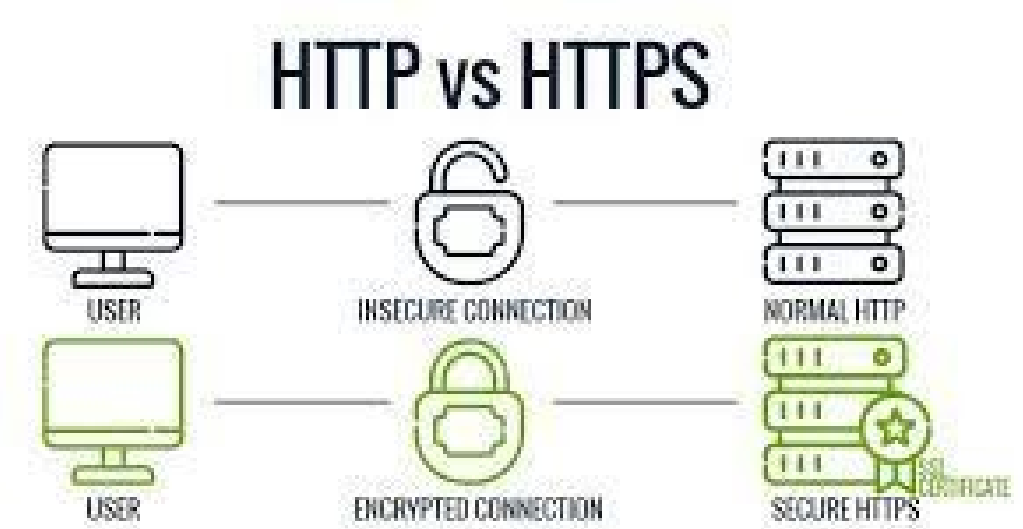
Secure sockets: purpose

Secure sockets provides
- confidentiality
- integrity
- authentication



Airline ticket purchased online, using a payment card
- confidentiality of data: card#, cpr#, your name, destination
- integrity of data: amount to be paid
- authentication: we are talking to server sas.dk, not hacker.com

HTTPS = HTTP + SSL



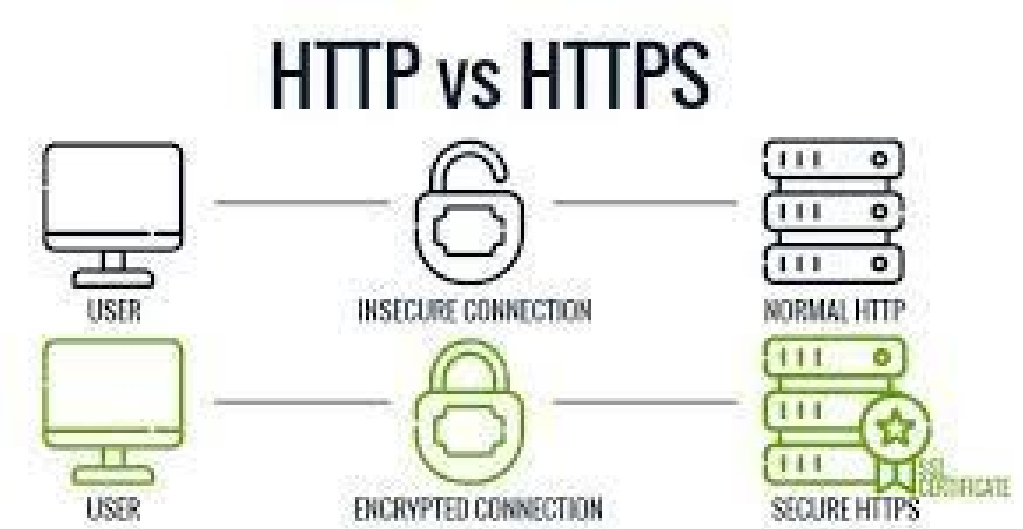
SSL is an easy way for a webserver to attain security

- ‘SSL’ takes care of the security
- the web admin simply (?) configures the web server to use SSL
- then the web site respons to HTTPS queries, not HTTP queries
- all web browser ‘speak’ SSL

SSL is independent from HTTP

- SSL protocol independent from HTTP
- browser and server may use different SSL implementations
- SSL also used in FTPS (secure FTP) and many more..

HTTPS = HTTP + SSL



Most web servers use HTTPS

The Internet privacy organization,
EFF (Electronic Frontier Foundation):

- ".. every web request should be HTTPS"

ELECTRONIC FRONTIER FOUNDATION

HTTPS/SSL handshake (highly simplified)

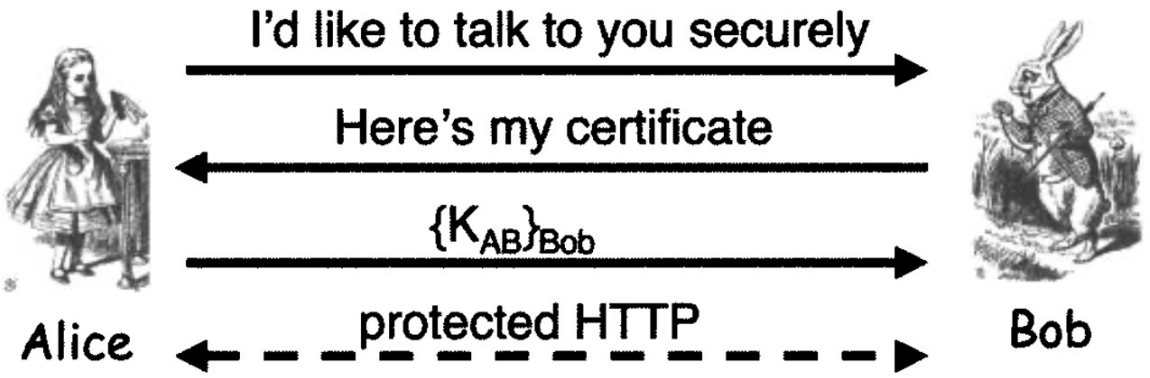
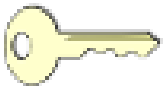


Figure 10.3: Too-Simple Protocol

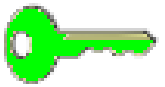
Main purposes of handshake

- Alice (browser) and Bob (server) agree on a symmetric key K_{AB}
- Alice verifies Bob's identity



Notation $\{K_{AB}\}_{Bob}$

- $\{M\}_{Bob}$ is (output of) asymmetric encryption of message M using Bob's **public key**
- notation is from Mark Stamp: Information Security 2/E, Section 4.6



Agreeing on a symmetric encryption key (cont.)

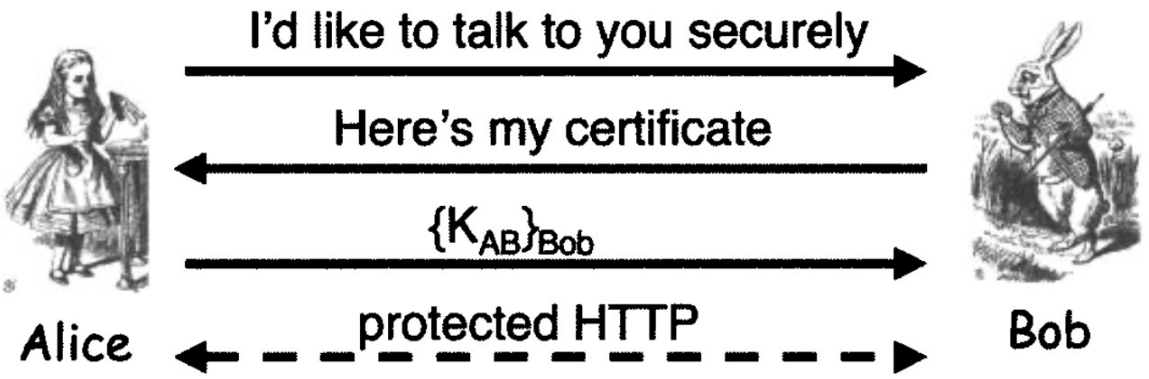


Figure 10.3: Too-Simple Protocol

Bob receives the asymmetrically encrypted symmetric key K_{AB}
- then Bob decrypts K_{AB} using Bob's private key
- nobody else knows Bob's private key

Notation (cont.)
- let's write $[M]_{Bob}$ for decrypting message M using Bob's private key
- then we can write:

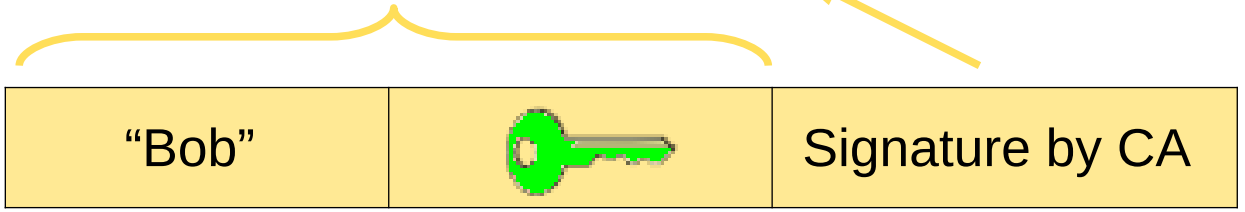
$$[\{K_{AB}\}_{Bob}]_{Bob}] = K_{AB}$$

A PKI certificate

As a web admin, to use SSL you need a (PKI) certificate

You apply (possibly pay) to get a certificate
- from a CA = certificate authority
- ca1.gov.dk, letsencrypt.com, ..
- convince the CA about your identity
- generate a public and private key pair
- submit the public key to the CA

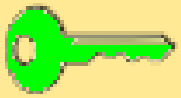
Certificate = text M + signature S



Identity “Bob”
- in practice lots of details such as URL

Notation for signatures: $S = [M]_{CA}$
- $[M]_{CA}$ is digital signature of text M signed by ‘CA’
- signature is output of asymmetric encryption of message M using CA’s private key

A PKI certificate (cont.): signature

“Bob”		Signature by CA

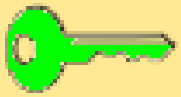
M is the text of the certificate
- “Bob”
- the public key
- (probably theres even more in the certificate)

Notation:
- signature S = $[M]_{CA}$

Alice verifies the signature using the public key of the CA
- Alice’s browser already knows the CA’s public key

Alice compares
- the text M
- with the decrypted signature {S}CA = $\{[M]_{CA}\}_{CA}$ = M
- Alice’s verification succeeds if they are identical

PKI-signing also uses cryptographic hashing

“Bob”		Signature by CA

The (simplified) notation:
- signature S = $[M]_{CA}$

But in actually, the text M is cryptographically hashed before signing, so
- S = $[hash(M)]_{CA}$
- conversely, hashing is used in the verification also

Properties of cryptographic hash functions include (not a full list) -
- for instance sha256
- takes any input M (including very long inputs)
- produces an output of a fixed size, say 256 bits
- computing a hash value is fast
- “fingerprint property”:
  in practice impossible to find two messages M and M’
  so that hash(M) = hash(M’)

} Fingerprint property: “hash value is just as good as original text”

Exercise:
Why agree on a symmetric key?

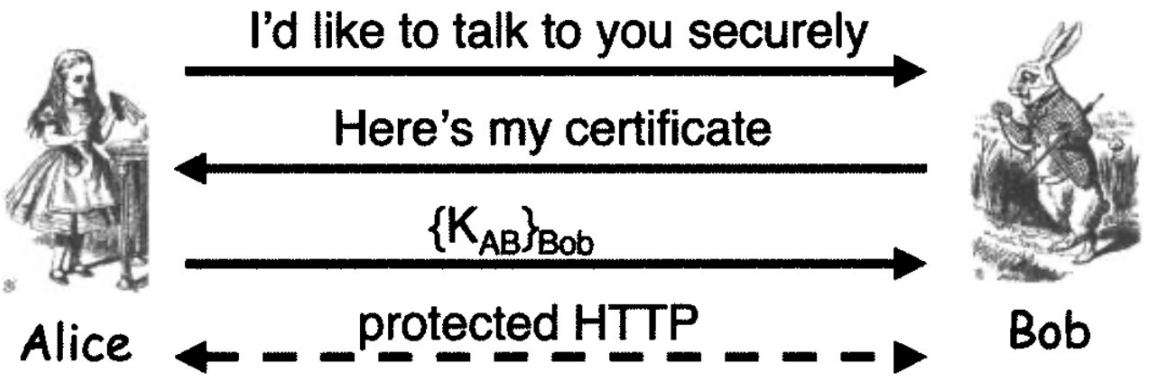


Figure 10.3: Too-Simple Protocol

Why agree on a symmetric key,
instead of Alice just using Bob's public key
to encrypt all of Alice's messages to Bob? (Alice's https requests)

Conversely, Alice could send her certificate to Bob,
and Bob's could use Alice's public key for Bob's https responses
- let's assume it is fairly easy for Alice to obtain a certificate

Symmetric encryption in SSL: selection of ‘cipher’

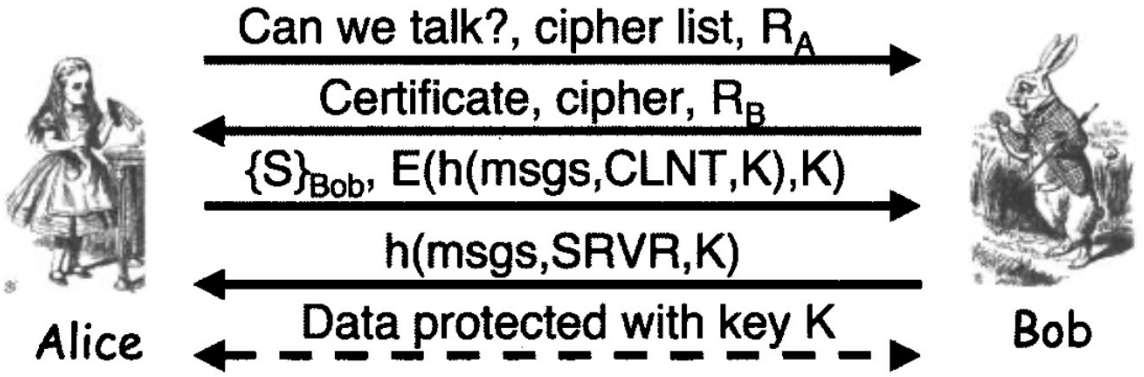


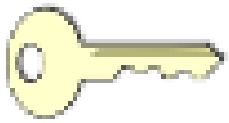
Figure 10.4: Simplified SSL

Actual SSL handshakes (still simplified) also involve selection of ‘cipher’
- algorithm for symmetric encryption
- also algorithm for integrity (cryptographic hashing)

Alice sends ‘cipher list’ (options)
Bob selects and sends ‘cipher’

Most frequently used symmetric algorithm
- AES with 128 bit key

Attempting to break symmetric encryption



Alice

$\{card\#, cpr\#, .. \}_{KAB}$

Bob

Threat assumption:

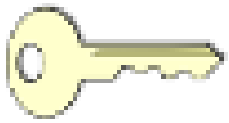
- the attacker knows the encryption algorithm (AES 128 bit)
- but of course not the key
- yet, the attacker has partial information: the text is card# and cpr# (numbers)

Let's use C for the ciphertext $\{card\#, cpr\#, .. \}_{KAB}$

The most promising approach for an attacker is to guess the key brute-force

1. guess a key, say K = 000..000
2. do a decryption, $M = [C]_K$
3. if M actually contains a card# and a cpr#: success
4. else goto step 1 and try the next key

Exercise:
is the attack feasible?



Alice

$\{card\#, cpr\#, .. \}_{KAB}$

Bob

The most promising approach for an attacker is to guess the key brute-force
1. guess a key, say K = 000..000
2. do a decryption, $M = [C]_K$
3. if M actually contains a card# and a cpr#: success
4. else goto step 1 and try the next key

If the attacker has large amounts of computing power,
is this attack feasible? possible in the worst, realistic case?

Today’s plan

1. Secure sockets

2. SQL Injection background - user friendliness and security

3. The SQL-Injection-Frontend

4. SQL injection attacks

5. How to protect against SQL injection attacks?

SQL injection in 2024

ian carroll / Bypassing airport security via SQL injection

Bypassing airport security via SQL injection

08/29/2024

In 2024, Ian Carroll and a co-worker discovered
an SQL injection attack into a website
that controlled access to some US airports' 'bypass security lines'.
The bypass lanes let crew members bypass the security check

URL: https://ian.sh/tsa

Dynamic vs. composed query

Dynamic query = user-provided

Composed query = program-provided part + user-provided part

The notions of dynamic and composed queries

- are not universal computer science terms
  rather I have defined their meaning
  for the purpose of discussing SQL injection

(Convention, red font indicates user-provided input)

Dynamic vs. composed query (cont.)

A composed query may be:

```
select * from course where course_id = 'seach_term'
```

Why would we define composed query?

- for usability - the user need only type a course id
- for security - say, the user should courses only
  or *some* courses only

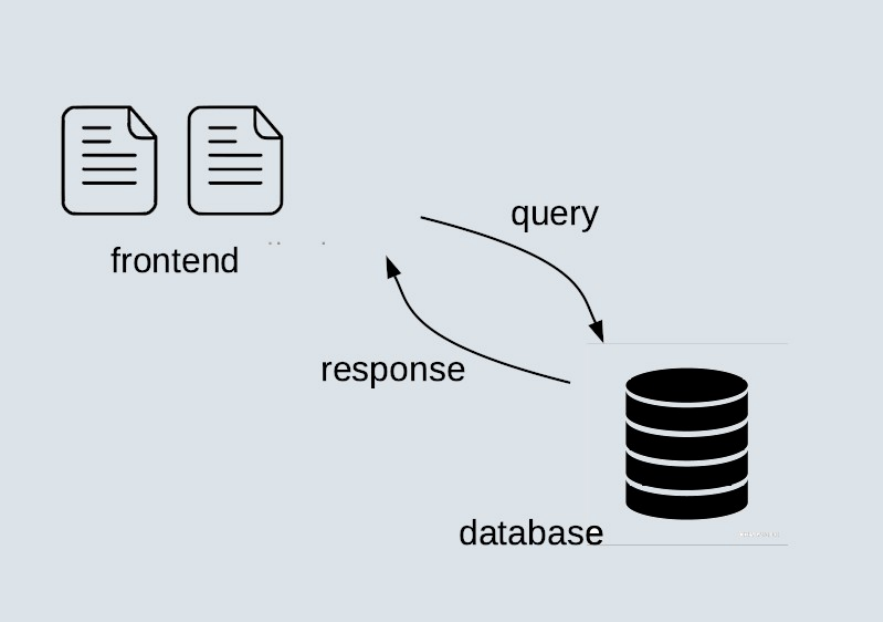
How might we define a composed query?

- by string contatenation (‘addition’)

Composed query: SQL injection

BIO-101’ --

this is user-provded input
that bypasses censoring of Biology courses




```
Please type id of a course: BIO-101' --
Query to be executed: select * from course where course_id = 'BIO-101' --' and dept_name != 'Biology'

 course_id | title              | dept_name | credits
-----------+--------------------+-----------+---------
 BIO-101   | Intro. to Biology | Biology   | 4
(1 row)
```

Dynamic query

An example dynamic query
(entirely user-defined)

```
select * from course;
```

Let's run the query
- against the university db
- from an ordinary SQL client

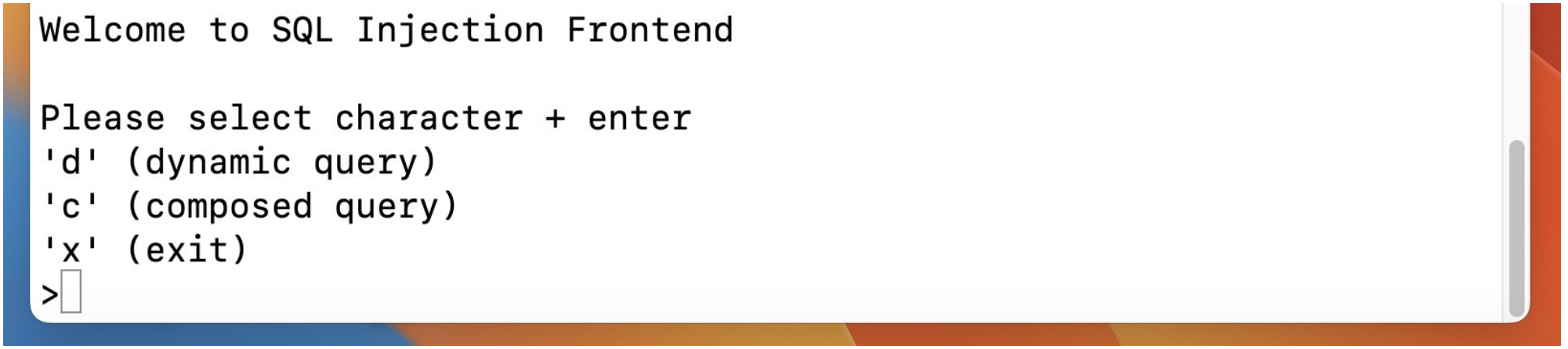
```
university=# select * from course;
 course_id |           title            | dept_name  | credits
-----------+----------------------------+------------+---------
 BIO-101   | Intro. to Biology          | Biology    |       4
 BIO-301   | Genetics                   | Biology    |       4
 BIO-399   | Computational Biology      | Biology    |       3
 CS-101    | Intro. to Computer Science | Comp. Sci. |       4
 CS-190    | Game Design                | Comp. Sci. |       4
 CS-315    | Robotics                   | Comp. Sci. |       3
 CS-319    | Image Processing           | Comp. Sci. |       3
 CS-347    | Database System Concepts   | Comp. Sci. |       3
 EE-181    | Intro. to Digital Systems  | Elec. Eng. |       3
 FIN-201   | Investment Banking         | Finance    |       3
 HIS-351   | World History              | History    |       3
 MU-199    | Music Video Production     | Music      |       3
 PHY-101   | Physical Principles        | Physics    |       4
(13 rows)
```

Exercise (A)

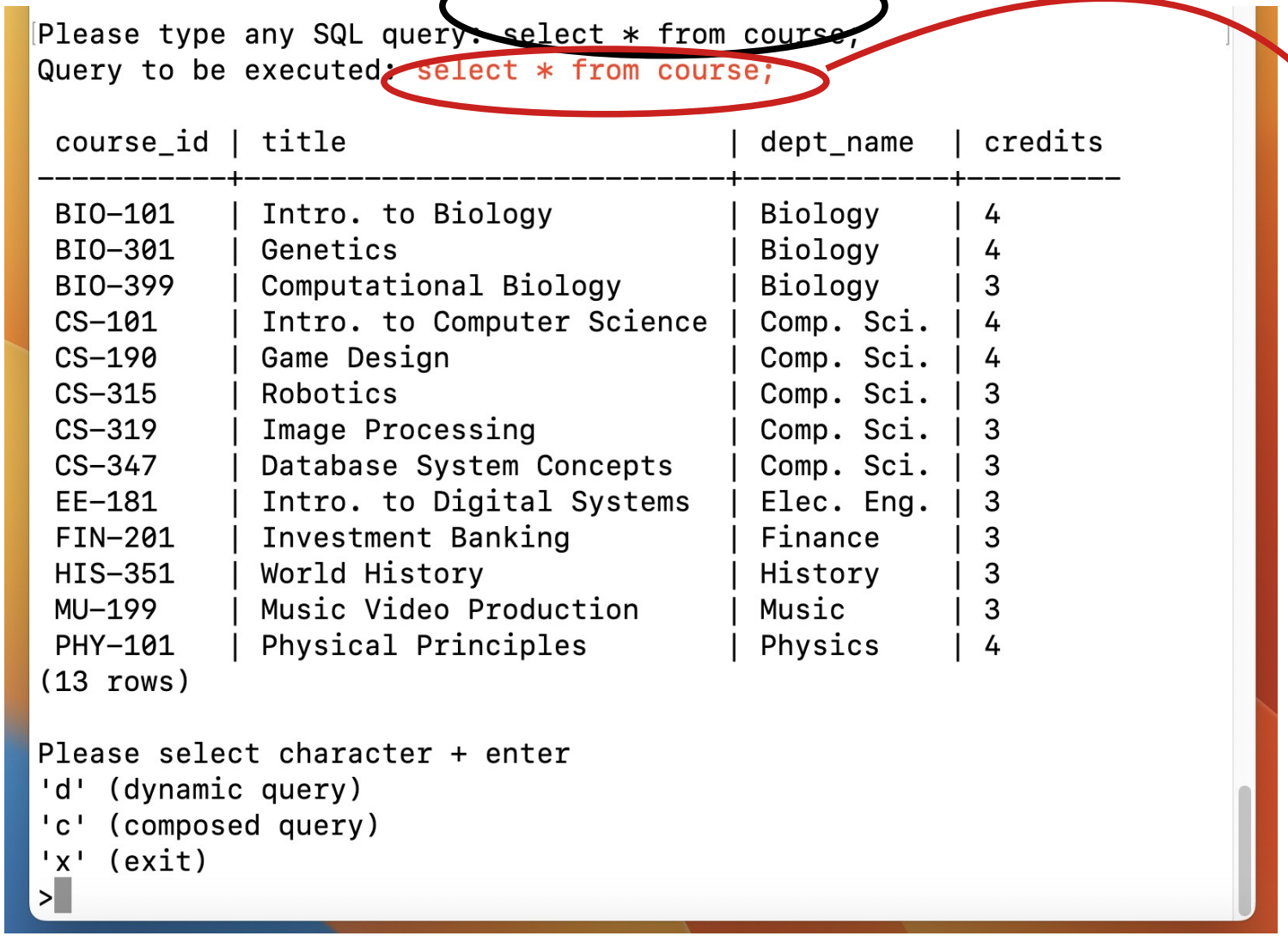
```
select * from course
```

Modify the dynamic query
so that the result set is
all courses from the Biology department

Running
‘SQL Injection Frontend’



d: *dynamic* query



User input

Displays the SQL query sent to database

A composed query

A university has defined an interface
to the university database based on
a composed query

A) For user-friendliness,
the interface lets the user type
only the course_id, say, CS-101

B) Also, since this is a conservative university in the US the interface *blocks courses from the Biology department*

c: composed query
CS-101

```
Please select character + enter
'd' (dynamic query)
'c' (composed query)
'x' (exit)
>c

Please type id of a course: CS-101
Query to be executed: select * from course where course_id = 'CS-101' and dept_name != 'Biology'

 course_id | title                      | dept_name  | credits
-----------+----------------------------+------------+---------
 CS-101    | Intro. to Computer Science | Comp. Sci. | 4
(1 row)
```

As we can see, the user interface

- inserts the course id ‘CS-101’ into a full SQL sentence (for user friendliness)
- and rejects courses from the Biology Department (to protect students against Darwinism)

Combined query (cont.)

```
select * from course
        where course_id = 'seach_term'
        and dept_name != ‘Biology’
```

user-provided

later we will construct this query in C#

Of course !!

- combined queries are dangerous
- may be vulnerable to SQL injections

Today’s plan

1. Secure sockets
2. SQL Injection background - user friendliness and security

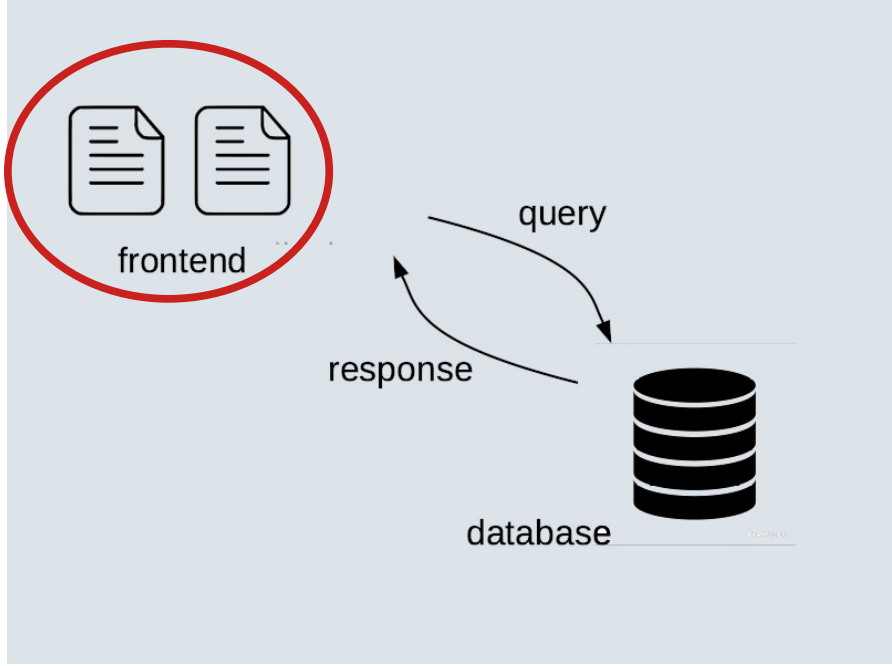
3. The SQL-Injection-Frontend
4. SQL injection attacks
5. How to protect against SQL injection attacks?

The frontend

Program.cs
- user interface
- ‘looping’

QueryConstructor.cs
- composes queries
- sends queries to PostgreSQL_Client
- Assignment questions require modifications

PostgreSQL_Client.cs
- source file for SQL client
- sends queries to database
- prints results
- (don’t modify)



SQL-Injection-Frontend

PostgreSQL_Client.cs

The SQL client
- an instance of the class is a client to a PostgreSQL database
- you are not supposed to change the source code

Application programming interface (API)

- *constructor PostgreSQL_Client(string uname, string pword, string db)*
  The constructor returns a PostgreSQL_Client object connected to the database “db”, with access rights for the user “uname” and password “pword”.
  - you need to change username+password in the call to the constructor (in QueryConstructor.cs)

- *method void query(string? sql)*
  The method takes as input an sql query, for instance “select * from course” and prints the query result in the console

- *method query(string? sql, string? name, string? val).*
  Same as query/1 The second and third parameters form a name/value pair. They are needed for separate parameter passing.

Program.cs

```
// Welcome message
Console.WriteLine("Welcome to SQL Injection Frontend\n");

QueryConstructor qConstructor = new QueryConstructor();

// the user interface
string? s = "x";

do {
  Console.Write("Please select character + enter\n"
        + "'d' (dynamic query)\n"
        + "'c' (composed query)\n"
        + "'x' (exit)\n"
        + ">");
  s = Console.ReadLine();
  Console.WriteLine();
  switch (s) {
     case "d":
       qConstructor.dynamicQuery();
       break;
   // .. the rest is omitted
```

Entry point of frontend

dynamic query

composed query

Exercises

(B)
Run SQL-Injection-Frontend,
select ‘d’ and type a query.

(C)
Select ‘c’ and provide the search term ‘BIO-101’. Is the “security” condition in place?

(D)
Modify the C# code in QueryConstructor.cs so that the composed query is more flexible. If the user inputs the search term ‘CS’, all computer science courses should be shown. *Hint:* The query should be constructed using elements from Figure 4 (in my notes).

How is a combined query constructed?

```
Please type id of a course: CS-101
Query to be executed: select * from course where course_id = 'CS-101' and dept_name != 'Biology'

 course_id | title                       | dept_name  | credits
-----------+-----------------------------+------------+---------
 CS-101    | Intro. to Computer Science | Comp. Sci. | 4
(1 row)
```

CS-101

```
select * from course
    where course_id = 'CS-101'
    and dept_name != 'Biology'
```

Program.cs (below) does *"query construction by string concatenation"*

- you may use similar approach in JDBC (Java)
- I say again: this is the vulnerable approach we want to avoid

```
string staticSQLbefore = "select * from course where course_id = '";
Console.Write("Please type id of a course, or part of id: ");
string? user_defined = Console.ReadLine();
string staticSQLafter = "' and dept_name != 'Biology'";
string sql = staticSQLbefore + user_defined + staticSQLafter;
```

Today’s plan

1. Secure sockets
2. SQL Injection background - user friendliness and security
3. The SQL-Injection-Frontend

4. SQL injection attacks
5. How to protect against SQL injection attacks?

Exercise (E)

Select ‘c’ and type an injection attack input,
so that the Biology course BIO-301 Genetics
is returned.

Use the approach indicated in my notes
“SQL-Injection and defenses”

See the section “Hacking a specific Biology course” (composed query)

How does SQL injection work?

```
Please type id of a course: BIO-101' --
Query to be executed: select * from course where course_id = 'BIO-101' --' and dept_name != 'Biology'

 course_id | title              | dept_name | credits
-----------+--------------------+-----------+---------
 BIO-101   | Intro. to Biology | Biology   | 4
(1 row)
```

```
select * from course
    where course_id = 'BIO-101'--'
    and dept_name != 'Biology'
```

- black is static (program-provided)
- red is dynamic (user-provided)
- underlined is executed
- not underlined is interpreted as comment

Main trick

- hyphen pair (--) is start of comment

Also, quotation mark (‘) in input BIO-101’-- ensures that BIO-101 is search literal

Exercise (F)

Select ‘c’, and type an injection attack input,
so that all biology courses are returned in the result set.

Use the approaches indicated in my paper “SQL-Injection-Frontend”

(This time use sections “Hacking all biology courses”
and/or “Hacking all biology courses more elegantly”)

Today’s plan

1. Secure sockets
2. SQL Injection background - user friendliness and security
3. The SQL-Injection-Frontend
4. SQL injection attacks
5. How to protect against SQL injection attacks?

What is going wrong?

CS-101

```
select * from course
      where course_id = 'CS-101'
      and dept_name != 'Biology'
```

In the intended use (where the user is benign)

- the user is providing only *data* (CS-101)
  = a course literal to be used as a search literal

But ---- a malign user may provide input that are *program* parts

- --
- ;
- or true

And we are not *validating* the user input

- ie., we are not validating that the user input has no program parts

Program-for-data replacement is seen also, eg. in C buffer overflows

Protection

The main idea behind protection against SQL injection attacks is to *validate the user input*

- ensure it contains only data, not program

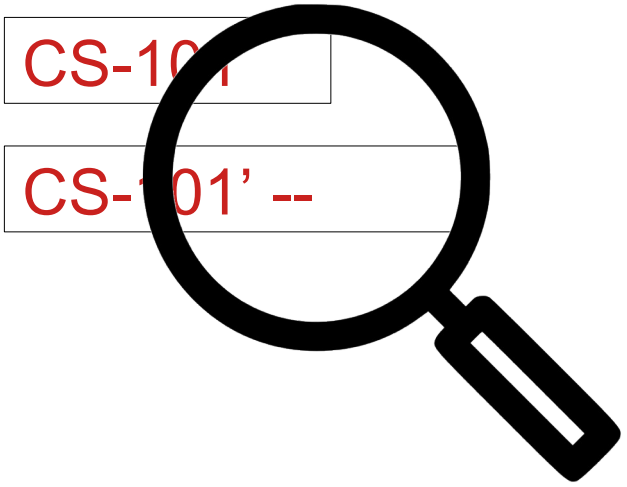
```
select * from course
      where course_id = 'CS-101'
      and dept_name != 'Biology'
```

We will validate the user input ‘CS-101’ to chech for program elements such as ‘--’

In our context of C# and PostgreSQL, our approach is to use

- “separate parameter passing”
- as defined by the C# library Npgsql
- and implemented in query/3 of PostgreSQL_Client.cs
- (ideally I would present a universal solution but I am not aware of one)

But first, let's take a simpler approach: *Allow-list and Disallow-list* Input Validation



An allow list (or positive list)
is a set of allowed values for the input data
(or a description of a set)

Conversely, a disallow list (or negative list)
is a set / description of values not allowed.

Protecting by positive / negative lists is
- implementing 'by hand' (programmer's hand)
- "Do-It-Yourself" protection
- rather than using built-in features
  (such as 'separate parameter passing")

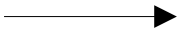
Exercises

(G)
(Negative lists.) Suggest characters and/or strings to include in a negative list. The negative list should protect dynamic queries against SQL injection attacks. Also point to some characcters or string that you would like, on the one hand, to exclude, but where the exclusion would also have disadvantages.

(H)
(Positive lists.) Would you consider the positive list approach suggested in my notes SQL-Injection and defenses (Section 4) to be secure?

Stored functions with parameters passed separately

```
select * from course
      where course_id = 'CS-101'
      and dept_name != 'Biology'
```



we must turn SQL query into a stored function with course_id as a separate parameter

Intuitively, the idea is that the database system will automatically ..

- validate the parameter part (CS-101)
- non-parameter part (the 'censoring') is "hard-coded" into the stored function so that it is always executed (not skipped due to '--' trick)

Assignment 5, Question 1

Question 1. Define a stored database function that returns a course from the course table in the university database. You may name the stored function safe_course().

The function should have one input parameter of type VARCHAR(8). The function should return courses whose course id match the input. (There will be at most one such course, since the course id is a primary key of the table.)

Naturally, the function should leave out any course offered by the Biology department. Show the SQL code that defines the function.

Question 2 - is about calling safe_course() with separate parameter passing

Question 3 - is about testing your solution against an SQL injection attack

Recap of (stored) SQL functions

Let’s define a stored function that returns all students of a given department,

ie. that returns students from the Physics department similarly as returned by an ordinary SQL select query

```
university=# select * from student where dept_name = 'Physics';
  id   |  name   | dept_name | tot_cred
-------+---------+-----------+----------
 44553 | Peltier | Physics   |       56
 45678 | Levy    | Physics   |       46
 70557 | Snow    | Physics   |        0
(3 rows)

university=#
```

- or from any other department, including Biology
  (if that department is provided as input)

Stored SQL function ‘students()’

So we want to define a function ‘students()’

```
create function students(..)
returns ..
language sql as
$$
    select ..
$$
```

which we may then call as in -

```
university=# select * from students('Physics');
  id   |  name   | dept_name | tot_cred
-------+---------+-----------+----------
 44553 | Peltier | Physics   |       56
 45678 | Levy    | Physics   |       46
 70557 | Snow    | Physics   |        0
(3 rows)

university=#
```

Exercise

Define function students()

Assignment 5

Question 1 - definition of safe_course() is similar to function students()
except that safe_course() must return courses
and ‘censor’ Biology courses

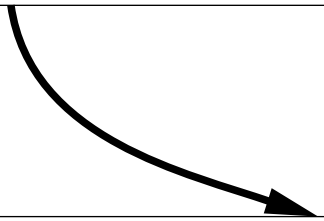
Assignment 5 (cont.)

Question 2. In SQL-Injection-Frontend, in the switch statement of Program.cs, define an option ‘sc’ that calls a method safeComposedQuery().

Also, in QueryConstructor.cs, define method safeComposedQuery(). The new function should be an improvement over composedQuery() in two ways: firstly, it must call the stored database function safe_course(); and secondly, it must use separate parameter passing, including by calling query/3 as defined in PostgreSQL_client.cs.

safeComposedQuery()
Firstly: call safe_course

```
string staticSQLbefore = "select * from course where course_id = '";
Console.Write("Please type id of a course, or part of id: ");
string? user_defined = Console.ReadLine();
string staticSQLafter = "' and dept_name != 'Biology'";
string sql = staticSQLbefore + user_defined + staticSQLafter;
```



```
string staticSQLbefore = "select * from safe_course('";
Console.Write("Please type id of a course, or part of id: ");
string? user_defined = Console.ReadLine();
string staticSQLafter = "')";
string sql = staticSQLbefore + user_defined + staticSQLafter;
```

safeComposedQuery()
Firstly: .. still not safe

```
Please type id of a course: BIO-101') ; select * from course; --
Query to be executed: select * from safe_course('BIO-101') ; select * from course; --') ]

 course_id | title | dept_name | credits
-----------+-------+-----------+---------
(0 rows)

 course_id | title                        | dept_name  | credits
-----------+------------------------------+------------+---------
 BIO-101   | Intro. to Biology            | Biology    | 4
 BIO-301   | Genetics                     | Biology    | 4
 BIO-399   | Computational Biology        | Biology    | 3
 CS-101    | Intro. to Computer Science   | Comp. Sci. | 4
 CS-190    | Game Design                  | Comp. Sci. | 4
 CS-315    | Robotics                     | Comp. Sci. | 3
 CS-319    | Image Processing             | Comp. Sci. | 3
 CS-347    | Database System Concepts     | Comp. Sci. | 3
 EE-181    | Intro. to Digital Systems    | Elec. Eng. | 3
 FIN-201   | Investment Banking           | Finance    | 3
 HIS-351   | World History                | History    | 3
 MU-199    | Music Video Production       | Music      | 3
 PHY-101   | Physical Principles          | Physics    | 4
(13 rows)
```

safeComposedQuery()
.. still not safe (cont.)

```
Please type id of a course: BIO-101') ; select * from course; --
Query to be executed: select * from safe_course('BIO-101') ; select * from course; --')
```

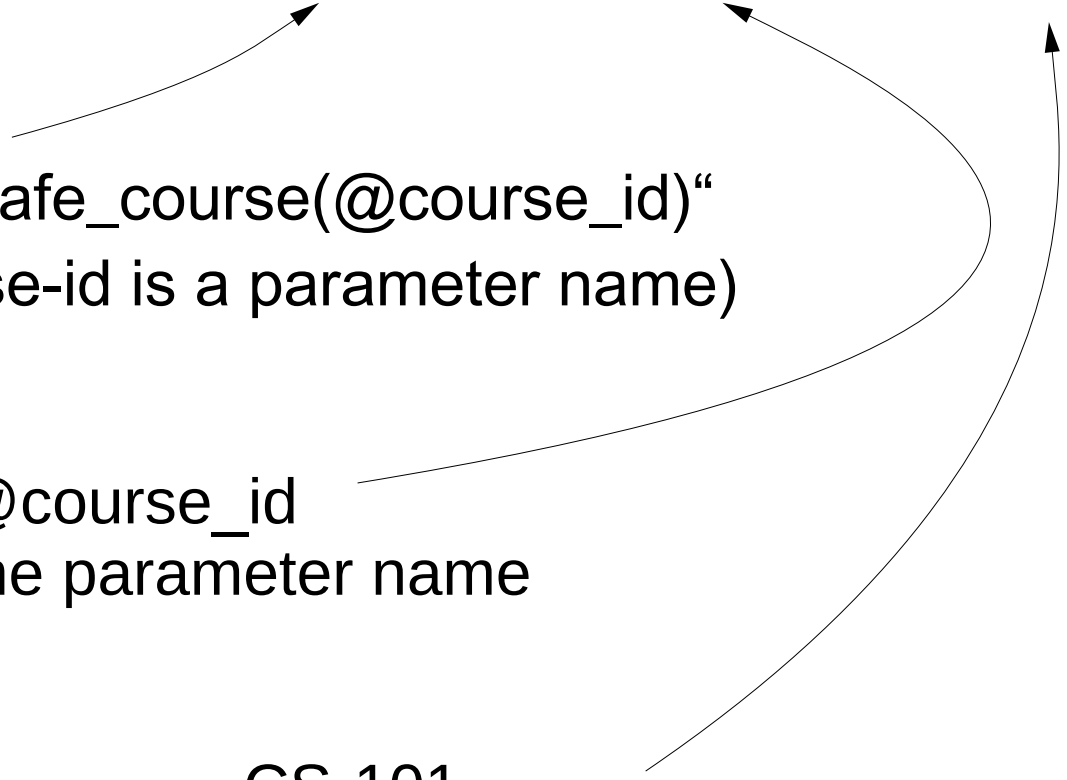
user-defined input: BIO-101') ; select * from course; --
passed to safe_course(): 'BIO-101'

So even though safe_course() ..
- does some parameter checking
- does the censoring (not allowing Biology courses)

.. we are still hacked

Secondly, use query/3 for separate parameter passing

public void query(string? sql, string? name, string? val)



“select * from safe_course(@course_id)“
(where @course-id is a parameter name)

@course_id
the parameter name

CS-101
the parameter value
could be really malicious CS-101’) ; --
but now definitely will be understod as parameter

Using the library Npgsql to connect to the database

The C# program SQL-Injection-Frontend.cs
firstly sets up a database connection,
which is represented by an instance of class NpgsqlConnection:

```
String s = "Host=localhost;Username= ..";
NpgsqlConnection con = new NpgsqlConnection(s);
con.Open();
```

Using Npgsql to execute SQL commands (without separate parameter passing)

The C# program uses an instance of class NpgsqlCommand to represent the SQL statement

```
public void query(string? sql) {
 ..
 NpgsqlCommand cmd = new NpgsqlCommand(sql, con);
 .. cmd.ExecuteReader();
```

con is the database connection

query/3 uses a Npgsql feature for separate parameter passing

```
public void query(string? sql, string? name, string? val) {
   ..
   NpgsqlCommand cmd = new NpgsqlCommand(sql, con);
   cmd.Parameters.Add(new NpgsqlParameter(name, val));
   .. cmd.ExecuteReader();
```

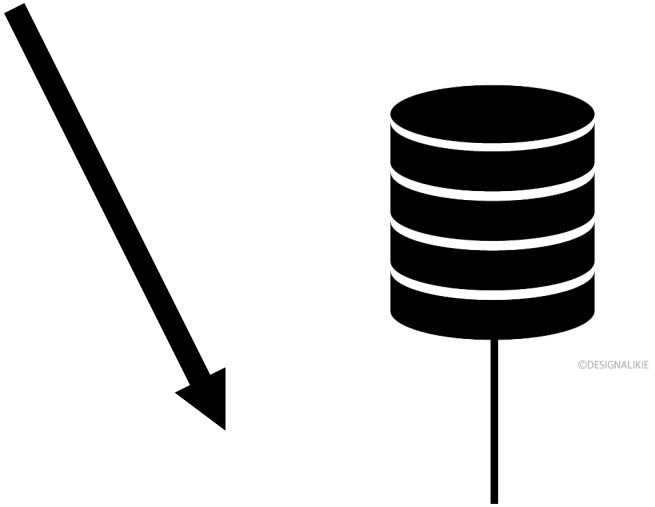
the command (or query) cmd is extended with the parameter name/value pair

Summary of separate parameter passing

select * from safe_course(@course_id)

name: @course_id
value: “CS-101’); select * from course; --”

this malign value
will be identified as invalid
because it is passed
separately as a parameter



create or replace function safe_course(c_id varchar(8))
...

Summary (cont.)

These recommendations are from OWASP Foundation

- Open Web Application Security Project
- URL
  - cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
  - (accessed Nov 4, 2025)

Primary Defenses:

- **Option 1: Use of Prepared Statements (with Parameterized Queries)**
- **Option 2: Use of Properly Constructed Stored Procedures**
- **Option 3: Allow-list Input Validation**

- we can do prepared statements in PostgreSQL as well
  - of course with separate parameter passing
- ‘properly constructed’ (??) - everything must be properly constructed
- separate parameter passing is not a 100% guarantee

Exam questions - SQL injection

When is a C# program vulnerable to SQL injection?

An example malicious SQL injection

Protection against SQL injection using stored functions

Explain how to define an appropriate *stored function*

Emphasize separate parameter passing in *calls* to stored function

Implementation of separate parameter passing by query/3
- query/3 distinguishes between parameter names and values
- parameters names should begin with @, ie., @course_id
- query/3 calls function Parameters.Add of class NpgsqlCommand

Next course day about security: Monday Nov. 11th

2. Password-based authentication

- Hashing of users' passwords
- How to slow down brute-force attacks but retain fast login of legitimate users?

Literature (moodle):

Niels Jørgensen. Password-based user authentication (22 pages).