

SQL-Injection and defenses

Niels Jørgensen, November 5th, 2025

This note explains SQL-injection, a dangerous and common type of attack on relational databases, and discusses defenses against the attacks. The note is intended for teaching at the RUC course “Complex IT Systems” at Computer Science at RUC.

If you make a database accessible on the web, you expect the web user to provide input, say, to search for films or courses. But for at least two reasons you don’t provide the user with a full SQL client:

Usability. The user should be allowed to search for films by providing only simple keywords, such as a term that describes a film, say “Romance”. Of course, the user is (normally) not a programmer: he or she does not know SQL with its **SELECT** operations, **WHERE** clauses, etc.

Security. The user should not be allowed the privilege of executing all kinds of SQL queries. The ability to execute arbitrary queries might compromise privacy (say queries about another user’s query history). Or a malign user may attempt to delete our data (say, by using a **DROP TABLE** command).

Instead, the user normally interacts with the database via some interface. The C# Linq interface to SQL provides some protection against SQL Injection, because it uses the technique of ‘separate parameter passing’. The technique is described in this note. The note also discusses the program “SQL-Injection-Frontend”, a simplified interface developed for the purpose of illustrating SQL Injection and defenses.

Table of Contents

1. Background: ‘combined SQL queries’ in web user interfaces.....	2
A dynamic query.....	2
A combined query.....	3
Why would you define a combined SQL query if it is dangerous?.....	3
2. The SQL-Injection-Frontend.....	5
Source files of “SQL-Injection-Frontend”.....	5
Download, build, and run “SQL-Injection-Frontend”.....	7
Configuration.....	7
Running the dynamic query in SQL-Injection-Frontend.....	8
Running the combined query in SQL-Injection-Frontend.....	8
3. SQL injection attacks.....	10
Hacking a specific biology course (composed query).....	10
Hacking all biology courses.....	11
4. How to protect against SQL injection attacks?.....	13
Do-It-Yourself protection: negative approach (“disallow-lists”).....	13
Do-It-Yourself protection: Positive lists (“allow-lists”).....	14
System defined parameter validation.....	14
Stored functions - recapitulation.....	14
References.....	16

1. Background: ‘combined SQL queries’ in web user interfaces

The threat posed by a SQL injection attack is that the user-provided input, say a term that we expect to be part of a film title, in reality contains malign code.

SQL injection attacks are among the most common exploits of web applications. In its 2017 top-ten list, OWASP listed SQL injection as the number one risk (OWASP Foundation, 2021). OWASP is a non-profit security project.

Among the numerous SQL injection attacks that have been publicly reported is Albert Gonzales’ attack in 2008. Gonzales (with other members of a group) used SQL injection to steal millions of customer credit card numbers from 7-Eleven and other companies. Albert Gonzales was eventually sentenced to no less than 20 years in prison for these and other computer-related crimes (Zetter, 2010).

SQL injections are discussed briefly in (Silberschatz *et al.*, 2020), Chapter 5.1.1.5. They are part of the broader topic of “application security” as discussed in (Silberschatz *et al.*, 2020, p. 437ff).

A database may be vulnerable to an SQL injection attack if it executes a ‘combined SQL query’. A combined SQL query is constructed as a combination of a static and a dynamic part. The static part is program-provided, and the dynamic part is user-provided. The database becomes vulnerable if there is not a sufficient check of the dynamic part.

The notions of a static and dynamic part of a combined query as used in this note are not commonplace; I am adapting them from the terminology used in (Su and Wassermann, 2006).

A dynamic query

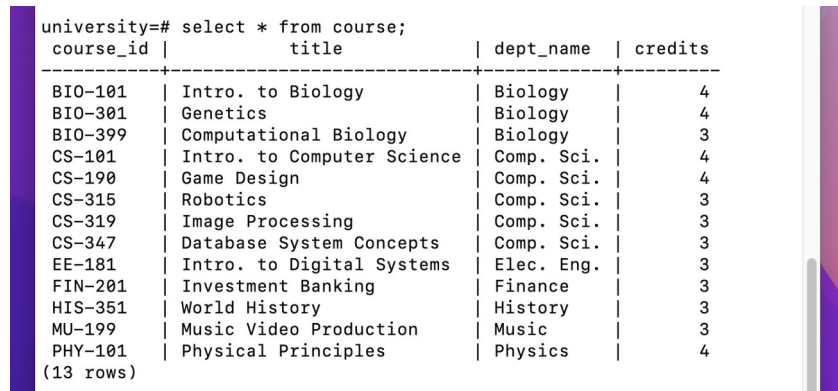
Consider the SQL query below in Figure 1. Let’s say the entire query is provided by the user. Then we call it a dynamic query. The figure uses red font to indicate the dynamic part of a query (in this case the entire query).



```
select * from course
```

Figure 1: An dynamic, ie., entirely user-provided query.

The following result will be shown if we execute this query against the university database.



```

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)

Figure 2: Result of execution of the dynamic query. The result set contains all 13 courses in the course tabel. Screenshot is of the PostgreSQL ‘elephant’ client on a Mac.

Obviously, it is dangerous to allow any user to execute an arbitrary query of the user’s choice, such as query #1. For instance, a user may type a malign query such as “**DROP TABLE** course”.

A combined query

Next in Figure 3 is a combined SQL query. The only part of the query that originates from the user is the course id search term (as before, red font indicates dynamic input). The notion of the static part (shown in black) of the query refers to the part that is not user-provided. Rather, the static part is provided by the program (we will soon se how). The static part does the actual selection from the table.

```

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

```

Figure 3: A combined query.

Even though this combined query is more safe than the dynamic query (Figure 1 above), it may be vulnerable to SQL injection attacks, as we shall see further on in this note. Soon the note will also explain the purpose of the clause involving the Biology department name.

Why would you define a combined SQL query if it is dangerous?

Even though combined SQL queries may pose a vulnerability - if the user-defined part is not sufficiently validated - lets reiterate why developers very often define a combined SQL query:

Usability is a major motivation for defining a combined query. The dynamic query is user friendly: the user has to enter only the course id, and is relieved of typing an entire SQL statement. In fact, with a combined query, we ought to enhance the user interface even further. For instance, we might use the keyword “like” and the wildcard character “%” as shown below in Figure 4. The query is similar to an example in Section 9.8.1 of (Silberschatz *et al.*, 2020, p. 438).

```
select * from course
  where course_id like '%user_defined%'
 and dept_name != 'Biology'
```

Figure 4: A hypothetical, more user-friendly SQL search command (not used in SQL-Injection-Frontend).

This enhancement is more user-friendly because the user may provide only the search term ‘CS’ and get information about all courses that have ‘CS’ in their course id. The user does not have to remember the course numbers such as “CS-101” etc. This note, however, don’t use such user-friendly enhancements. This is for simplicity; the combined query (Figure 3) is already vulnerable to an SQL injection attack even without the enhancements.

Security may be another motivation for combined queries. The database may contain sensitive data. For instance, in the university database, we may consider data about students to be confidential - if a given student can be identified. If a web user were allowed to enter a query such as **SELECT * from student**, that user would see the total credits achieved by any student, and students would be identified by name. So for security reasons, we may define a combined query that let’s the user access only certain tables in the database. In the combined query this is achieved by having the static part fix which table is searched (the course table).

Moreover, we may also consider the constraint about courses from the Biology department as security-related. Perhaps we are at a conservative, religious US institution that considers Darwin’s evolutionary biology to be dangerous. We may summarize the security-related goals pertaining to SQL-Injection-Frontend as follows:

- The user may retrieve only data from the course table.
- From within the course table, the user may not retrieve any data about the biology courses.

In terms of security, instead of the **WHERE** clause -

```
dept_name != 'Biology'
```

perhaps a more realistic constraint might involve the user’s password. But for simplicity, we stick to the Biology constraint.

2. The SQL-Injection-Frontend

The SQL-Injection Frontend has a simple architecture, shown below in Figure 5.

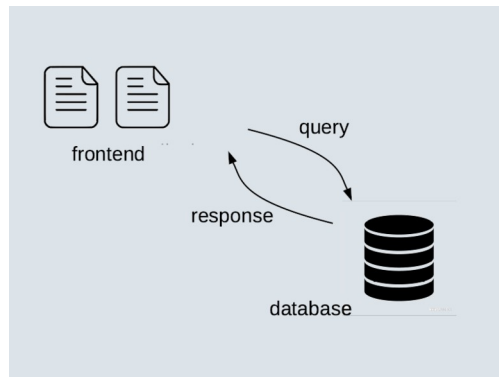


Figure 5: The frontend sends queries to the database (the university database) and receives responses. Internally, the frontend consists of two files only, the query constructor and the SQL client.

The query constructor asks the user for input, either a complete, dynamic SQL query or the user-defined part of a combined query, and sends the query via the client to the database. Finally, the query results are written to the console.

Source files of “SQL-Injection-Frontend”

The frontend consists of three C# source files. The note briefly explains their role, since you need to improve the program in order to solve the assignment questions related to SQL injection.

Program.cs. This is the user interface. By a C# convention, the file is the entry point of the entire frontend application. Specifically, the entry point of the application is the file’s first top-level statement. The source of *Program.cs* is listed below in Figure 6. The program asks the user to select a query type, and depending on the selected type, the program calls *dynamicQuery()* or *composedQuery()*.

QueryConstructor.cs. This program defines the two methods *dynamicQuery()* and *composedQuery()*. Each method constructs a query by asking the user for input, then possibly adding a static part (in *composedQuery()*), and finally sends the full query to the SQL client.

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 file *PostgreSQL_client.cs*. The following is the application programming interface (API) of the client (the functions that other programs are supposed to call).

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

- *method query(string? sql, string? name, string? val)*. Method *query/3* is similar to *query/1*. The second and third parameters form a name/value pair. That is, the name of a variable (presumably a variable name that appears in the first parameter) and the value of the variable. Method *query/3* is needed to construct a safe combined query (without SQL injection).

PostgreSQL_Client.cs contains about 200 program lines and uses the library Npgsql. This open source library lets C# code access a PostgreSQL database server.

```
// 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;
        case "c":
            qConstructor.composedQuery();
            break;
        // omitted: the remaining optional values of string s
    } // end switch
} while (s != "x");
```

Figure 6: Program.cs source listing.

```
// QueryConstructor.cs

public class QueryConstructor {

    public QueryConstructor() {
        client = new PostgreSQL_Client("university", "nielsj", "pizza");
        // retain university database
        // but change username and password
    }
    // omitted: the remaining optional values of string s
}
```

Figure 7: Query constructor.cs partial source listing. You need to change the username and password parameters in the call to *PostgreSQL_Client()*.

Download, build, and run “SQL-Injection-Frontend”

The three source files of SQL-Injection Frontend are available on RUC’s moodle site for the course “Complex IT Systems”. You should define a C# project containing these three files.

If you are participating in the course, you should already know how to define a C# project, and if so, please skip the remainder of this subsection.

You may build and run the application in several ways. Here’s how to build and run the application ‘by hand’, that is, from the command line, without an integrated development environment (IDE). The instructions assume that dotnet is already installed. To install dotnet, you 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”.

Configuration

To do a useful run of the program, you must also modify the parameters to the call to the PostgreSQL_Client constructor. You should modify the user and password parameters. Retain the value of the database parameter (the university database).

All examples assume that the university database is generated by running the script “university_database.sql”. The script is available on RUC’s moodle page of the course Complex IT Systems, under topic “SECTION 1 DATABASE THEORY -- Readings and resources”.

Running the dynamic query in SQL-Injection-Frontend

The first part (after the welcome statement) of the top-level statements in Program.cs asks the user to select a type of query. We assume that the user selects 'd' for 'dynamic query' and enters the dynamic query of Figure 1.

```
Please type any SQL query: select * from course
Query to be executed: 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)

Figure 8: The SQL-Injection-Frontend executing the dynamic query.

As you can see above in Figure 8, the result printed in the console is similar to what you get from any other database client. But also note a feature of the SQL-Injection-Frontend: it always prints the full SQL query that it sends to the database. Also it uses red font when printing the user provided part of a query.

Running the combined query in SQL-Injection-Frontend

Alternatively, our web user may select 'c' for 'combined query'. Now we are making use of the frontend's ability to help form queries from minimal user input. Let's assume the user seeks information about the computer science course CS-101.

```
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)

Figure 9: The SQL-Injection-Frontend executing the combined query where the user-defined part is 'CS-101'.

As indicated in Figure 9 above, the user-defined part gets inserted into a full, combined SQL query.

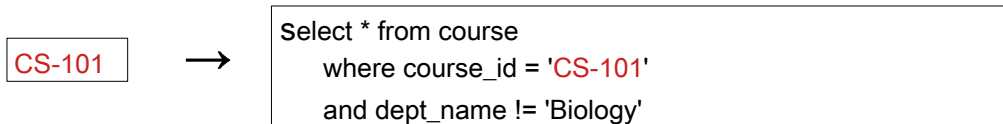


Figure 10: Inserting a user-defined input into a combined query.

Exactly how does the SQL-Injection-Frontend program accomplish the insertion shown in Figure 10?

This is in method `composedQuery()` as defined in `QueryConstructor.cs`. A part of the C# code defining the method is shown below in Figure 11. As you can see, the code uses string concatenation (aka. “string addition” or “string plussing”) to combine the user-provided input with the program-provided input.

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

Figure 11: Code snippet from source file `QueryConstructor.cs` showing the construction of the composed query.

In Figure 11, the composed query is the value of the string `sql`. The string value is data in the C# program. So all the SQL elements, such as “select *”, are strings in the C# program. Note that there are a total of four single quotation marks (') *inside* the string value assigned to string `sql`. You can see this in both Figures 9 and 11. There are quotation marks before and after the course name defined by the user (CS-101), as well as before and after the department name (Biology) defined by the program. We expect the user to type only the course id; therefore the C# program must provide the quotation marks, so that the string becomes a syntactically correct SQL statement.

3. SQL injection attacks

In this note, the purpose of the SQL injection attacks is to get as much course information possible - about all the courses, including the dangerous (?) biology courses.

The attacks target the way queries are composed in the C# method `composedQuery()`. There is no point in attacking the dynamic query, because it allows any SQL statement; indeed, dynamic queries should not be made freely available.

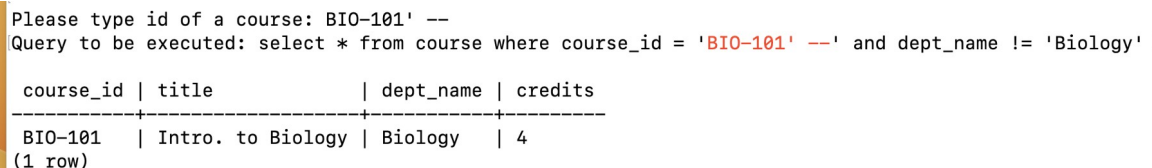
Hacking a specific biology course (composed query)

The first attack provides course information about the course “BIO-101” from the Biology department. I am using the term “hacking” in the sense of “displaying information that someone tries to hide”.

To do an injection attack that bypasses the condition that hides biology courses, the user may type the following input, as the user-defined part of a combined query:

BIO-101' --

The query result includes the biology course! See Figure 12 below.



```
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)
```

Figure 12: A combined query with injection attack as executed by SQL-Injection-Frontend.

The attack uses a pair of hyphens “--”. The SQL interpreter will interpret a hyphen pair as the beginning of a comment.

The note uses underlining to distinguish comments from the part of an SQL query that gets executed, as in Figure 13.

select * from course -- this is a comment

Figure 13: An SQL query with a comment. The part of the query that gets executed is underlined.

To illustrate how the attack works, Figure 14 below uses the distinction between executed part / comment part, as well as the distinction between program-provided (static) / user-provided (dynamic) part:

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

Figure 14: The full combined query with injection attack.

The main trick is the pair of hyphens “--”. The pair terminates the executable part of the query before the condition.

The other trick is the insertion of the quotation mark (') after the course id. This ensures that “BIO-101” is the literal searched for in the `course_id` column. Without the quotation mark, the interpreter would

search for a course id that included the two hyphens. But there is no such course id. Also, if the hyphens were included in the course id, they would not serve their hacking purpose - as a comment indicator.

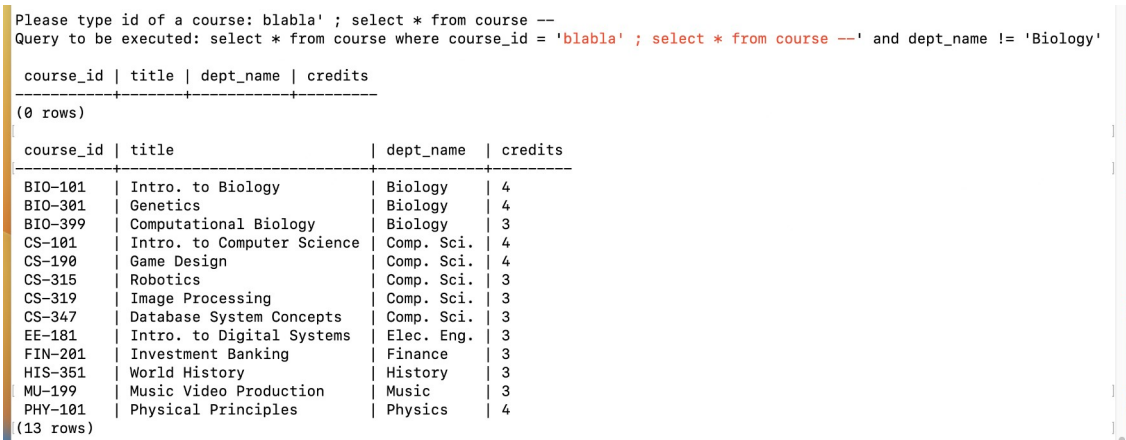
In the fictive example of ‘hacking’ the biology courses, the injection attack is perhaps not so serious. But the attack might have serious implications if the **WHERE** clause had been instead a condition about a user’s password or access rights. An attack on a password condition can use the same tricks.

Hacking all biology courses

Now our web user/hacker will be even more aggressive: he or she wants to display all courses, including all three biology courses. To ‘hack’ these courses, again the user selects ‘c’ for ‘combined query’, and when prompted, the user types:

```
blabla'; select * from course --
```

This is the output from the SQL-Injection-Frontend:



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

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)
```

Figure 15: A combined query with another injection attack, again as executed by SQL-Injection-Frontend.

In other words, the attack succeeds in displaying all three biology courses (and all the rest as well).

To see how the attack works, let’s look again at the full, combined query:

```
select * from course
where course id = 'blabla'; select * from course --' and dept_name != 'Biology'
```

Figure 16: The combined query marked to indicate executable / comment part, as well as static / dynamic part.

Again the attack uses a hyphen pair to escape the condition about biology courses. Moreover, the user-input now defines two SQL **SELECT** statements, before the escape:

The first statement searches for course IDs with the value ‘blabla’. Obviously, there is no course with this id, so the first statement returns zero courses (as you can see above in Figure 15).

The second statement is a full, new SQL select statement that returns all courses (unconditionally).

Hacking all biology courses more elegantly

As an attacker, one might be concerned that the hacking approach of inserting an extra SQL statement is too crude or obvious. For instance, an attacker might worry that the application checks if there are semicolons in the user-provided input (and if there is, blocks the query). All in all, for whatever reason, let's say the attacker wants there to be only a single SQL statement.

For this, the attacker can do the following injection attack. We assume as before that the user/attacker selects options 'c' for 'combined query'.

`blabla' or true --`

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

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)

Figure 17: Combined query with a more elegant injection attack, and executed by SQL-Injection-Frontend.

Here's the query:

```
select * from course
where course_id = 'blabla' or true --' and dept_name != 'Biology'
```

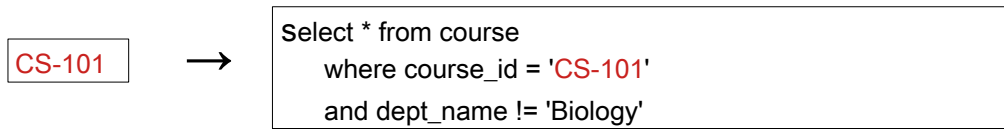
Table 18: Changing a logical condition by adding 'or true'.

Again, the attack does not care about the proper search term ('blabla') and comments away the biology condition (the hyphen pair). And this time, the user-input 'true' turns the over-all condition of the where clause into true. The form of the (remaining) where clause is 'conditionA or conditionB'. The first condition is false, but the second condition is true: the value 'true' is a truth constant - it is always true. Thus the overall where-clause is always true. Therefore all courses are shown.

4. How to protect against SQL injection attacks?

From a helicopter perspective, an SQL injection attack occurs when data and program get mixed up.

Let's apply the program versus data distinction to our intended use of the combined query (same illustration as above in Figure 10).



The weakness of the definition of method `combinedQuery()` was that while it was expecting a piece of data (a course search term, such as `'CS-101'`), it could not detect whether it actually got both data and program parts (such as `'blabla' or true --'`).

Mixing up data and program is a technique that is also used in other attacks, on many types of applications. For example, applications where the programming language C is the source code may be vulnerable to a so-called *buffer overflow* attack. In a buffer overflow attack, the application is expecting data input of a certain size (say, eight bytes). But the malicious user supplies a longer input, which “overflows” the memory location set aside for the input; and the extra input then flows into, or overwrites, another part of the memory. The overwrite may cause a malign program to be executed. The first publicly known buffer overflow attack was the Morris Worm in 1988, which infected thousands of computers in the US and caused a partial Internet break-down (English Wikipedia, 2022).

All protection methods discussed in the note may be implemented by modifying method `combinedQuery()` as defined in the source file `QueryConstructor.cs`.

First we discuss approaches for a programmer to validate the user input by hand. I call this a Do-It-Yourself (DIY) approach. Secondly we discuss approaches using a database system's built-in features for defining stored functions. Using stored functions or prepared statements is the main recommendation for protecting against SQL injection in the database community (Silberschatz *et al.*, 2020).

Do-It-Yourself protection: negative approach (“disallow-lists”)

But first, let's suppose we want to build our own protection mechanism from ground-up. We want to improve method `composedQuery()` by a hand-crafted way of validating the user input.

Since the user input is meant to be a search term to match against a course ID, and since the course ID field is declared to be of type `varchar(8)`, we could check if the input string is at most eight characters. This works well to protect against all the malign inputs discussed so far in this note.

This approach of requiring a certain string length is a form of ‘negative approach’: we exclude all strings larger than eight characters; while any shorter string is accepted, without further checking.

We can take the ‘negative’ approach further by also defining characters or strings that may *not* be in the user-provided input (even if the input is at most eight characters). Such a negative list might include the double hyphen string ‘--’. Note that constructing a good negative list may not be as easy as one might think, since we certainly want to allow strings with a single hyphen, as they are in all course IDs.

Do-It-Yourself protection: Positive lists (“allow-lists”)

Conversely to the ‘negative’ approach, we may construct a ‘positive list’. Specifically, we might define the set of permitted data to be the set of all values in the `course id` column of table `course`. It is indeed possible to construct such a list. One disadvantage is that the approach is obviously inefficient. When our program is run, and the user provides some input, we would essentially be searching the course IDs twice: first, to validate the user input (in the C# program), second, the actual search (by the database system). Also, if new courses are added to the database table, it seems we need to update our positive list.

System defined parameter validation

The recommended approach is to use stored functions that conduct a validation of the user-provided input. Validation in this sense is system-defined, that is, validation depends on how the database system implements stored functions, and how the system supports validation of data. At the course lecture, I will provide some detail about using a built-in feature of Npgsql for separate parameter passing, which does an appropriate validation of the parameter(s). Separate parameter passing is also implemented in the C# Linq (Language Integrated Query) database interface.

Finally, I recapitulate how to define a stored function. Please note that stored functions by themselves do not provide sufficient protection against SQL Injection; you still need to define a proper input validation.

Stored functions - recapitulation

Let us consider how to define a stored functions that we can use in a improved version of method `composedQuery()`. We consider a stored function, rather than a stored procedure, because we would like to obtain a query result. The query result is the course information that the user is asking for, and is indicating by typing a course id. Of course we want to keep censoring Biology courses! We consider the most simple type of stored functions, those that only use SQL in their function bodies.

The problem of actually defining a proper stored function is an assignment question. As a clue to solving the assignment question, let’s consider the definition of a stored function that returns all students of given department. In SQL we may write the following, to query the student table of the university database.

```

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=#

```

The query result is a table with three rows, one for each student in the Physics department.

Recapitulating from your previous course lecture on Database programming, you may define a stored function as follows:

```

create or replace function students(d_id varchar(20))
returns table (ID varchar(5),
              name varchar(20),
              dept_name varchar(20),
              tot_cred numeric(3,0) )
language sql as
$$
select * from student where student.dept_name = d_id;
$$

```

In the above function definition, the columns in the table to be returned are defined similarly to the definitions of table students in the university database. The input variable `d_id` is highlighted in red.

If you load the above function definition into the database, here is the query result you will get if you execute a call to the function:

```

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=#

```

Again, the query result is a table with three Physics students.

The assignment problem is to define a stored function to be called from an improved version of method `composedQuery()`. An improved version that keeps the Biology courses secret. Details about the requirements that the stored function must meet can be seen in Assignment 5.

References

- English Wikipedia, 2022. “The Morris Worm,” at https://en.wikipedia.org/wiki/Morris_worm, accessed 22 September 2022.
- OWASP Foundation, 2021. “OWASP Top Ten,” at <https://owasp.org/www-project-top-ten/#>, accessed 21 September 2022.
- Abraham Silberschatz, Henry F. Korth and S. Sudarshan, 2020. *Database Systems*, 7/E., McGraw Hill.
- Zhendong Su and Gary Wassermann, 2006. “The Essence of Command Injection Attacks in Web Applications,” *Acm Sigplan Notices*, volume 41, number 1, at <https://cs.uwaterloo.ca/~brecht/courses/702/Possible-Readings/security/injection-attacks-web-apps-popl-2006.pdf>.
- Kim Zetter, 2010. “Hacker Sentenced to 20 Years for Breach of Credit Card Processor,” *Wired Magazine*, at url: <https://www.wired.com/2010/03/heartland-sentencing/>.