SQL Insertion

(Unauthorized modifications caused by user-supplied data)

CST8276 Advanced Database Topics

Group #22

Alan Zangerl - zang0005@algonquinlive.com
Rhys Taylor - tayl0658@algonquinlive.com
Nathan Parker - park0545@algonquinlive.com
Aaron Goulding - goul0179@algonquinlive.com
Dongkwan Kim - kim00449@algonquinlive.com
Taeyeon Kim - kim00451@algonquinlive.com

Table of Contents

Introduction

Topic Description

Security on the Internet has always been a hotly contested issue. Whenever a piece of technology is new or updated, vulnerabilities are inevitably discovered. Some vulnerabilities are patched, but some are either never found or are too cumbersome to fix completely. Others are known, but without constant vigilance and upkeep will slip through the cracks. SQL injection is one of those vulnerabilities, where malicious actors can get access or cause destruction merely by inserting the right combination of SQL language into text input fields. You would think that anyone worth their salt as a developer or IT security person would catch this hole in the public-facing portion of the application? It's the equivalent of gaining access to the bank vault by putting on a janitor's uniform and walking right past the security guard. In the past ten years thousands of universities, some corporations, and even governments, have publicly been victims of SQL injection attacks. For our demonstration we hope to explore how it can be achieved and how it can be prevented, resulting in a better understanding as to why something with such potentially disastrous consequences can continue to happen.

Why We Chose This Topic

We chose this topic as we recognized its importance for both the computer programming field in general, and also to us individually as student developers. If we can understand the problem, and better yet, the solution to the problem, now, we can hopefully avoid being on the wrong end of a SQL injection attack in the future. Such preventive knowledge and measures will benefit us professionally, and our future employers, and the clients of our employers who trust us with their data. There are many solutions to the problem depending on the technology being used, and using more than one solution can make a product more robust in its security.

As we will demonstrate, the solution is not some complex abstract concept that is challenging to understand and implement, it is pretty simple and easier to put in place if done early in the development process. A secure solution is not the sole responsibility of the security team. Every role in the development process should assume the worst to reduce the chance of something being missed during a product's vulnerability testing.

Problem Description

Who:

In the last twenty years there have been numerous high-profile occurrences of SQL injection to gain unauthorized access to data, and likely many more occurrences that were never disclosed publicly. SQL injection helped resurrect famous rap legend Tupac Shakur in 2011 [1, 2]. In 2015, a massive number of universities and government institutions were bled for user accounts and other related information [3]. Although Tesla might be cutting edge for their batteries and autonomous driving technology, they need to work on their database designs as they were the subject of a user data breach by SQL injection in 2014 [4] [5]. As recently as 2019, the tech giant Epic Games was identified by researchers to be vulnerable to SQL injection. “Full account takeover could be a nightmare, especially for players of such a hugely popular online game that has been played by 80 million users worldwide, and when a good Fortnite account has been sold on eBay for over $50,000” [6].

SQL injection is a relatively straightforward problem to solve if it is kept at the forefront of organizations' concerns. However, as organizations grow and try to optimize their front-facing applications it becomes difficult to maintain complete security. It only takes a join of two user supplied text fields to potentially open a path to a SQL injection. For medium to large organizations the number of data fields that they manage makes it a daunting challenge to stay vigilant over time. Hackers and malicious actors know that this is a numbers game, but a game that can pay off big if they are persistent and wide in their efforts to find an opening.

The substantial potential costs of system breaches through SQL injection include: the legal costs of any lawsuits, the impact of reduced stock value and the impact to the company's image.

What:

When a bad actor identifies a vulnerability for SQL injection, they compose text in the text input fields on the web form that include SQL commands. If the database is poorly designed, these can run directly on the database. Either returning matching results – usually usernames and passwords [1] – or maliciously modifying tables. Sometimes, the attack is not noticed immediately. If a database has triggers or stored procedures that concatenate strings together after they have been saved in the database, this could lead to malicious code being executed long after it was added to the database. This highlights the importance of understanding how user data is processed and stored within public-facing databases.

Where:

While this problem originates with public users attempting to gain access to protected data on a web server, in reality, this is a database design problem. The design of the database determines the process for accepting and adding user-defined text into the database. If the design of the database is poor, hackers will be able to run SQL commands on the database. If the database is designed well, there will be a clear and hard separation between what is data, and what is the code to run the data, so there is no opportunity for user input to run as code on the database.

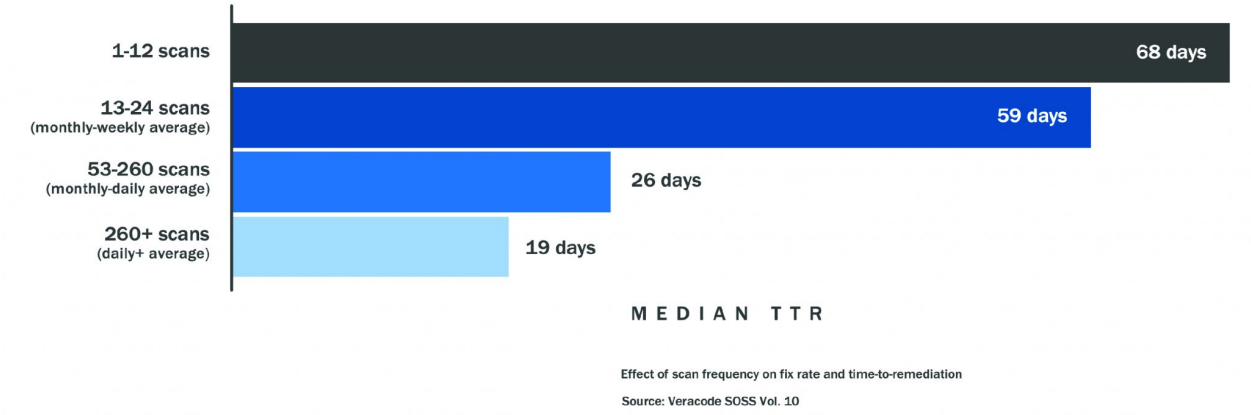
When:

Updates of dependencies and packages from internal sources or even external sources could be factors. Business client feature requests that cause the database design to be modified and results in weaknesses or design flaws. Some cases of SQL injection are immediate. If the input is not validated, and text is directly parsed into SQL statements, user-provided information in the form of SQL statements can be executed, resulting in loss of data or unintended access to sensitive information. However, not all SQL injections happen right away. Depending on the architecture of the system, SQL queries can sometimes get stored as text in a database. It won't be until the system needs to concatenate strings together, such as names, that the malicious scripts may be executed. Scenarios like this are the reason why developers should be cautious of how they interact with data, opting never to concatenate user data.

It is very important to plan how secure practices will be implemented in a project during development. Waiting until the application is almost ready to be put into production will result in a product that is filled with flaws and vulnerabilities. That being said, it is within the best practice to ensure that security is implemented in the software development lifecycle from the beginning. As stated by SHI, security should be integrated into DevOps [9], allowing developers who often do not get any guidance on writing safe code to interact with security experts, leading to much safer applications in the end. This collaboration can lead to a much more robust end product due to niche edge cases being caught earlier than usual.

It is also important to test regularly, as common sense would suggest vulnerabilities can be caught much earlier with consistent testing. Having security experts test a project to ensure vulnerabilities are caught early on both the application and data layer is very important, allowing every aspect of a product to become more robust. Another lesser-known fact about frequent testing is that testing more often will result in being able to patch issues much faster

as well, as shown in the image below.



[9] Automate testing in the software development life cycle (SDLC) by SHI

This shows that frequent testing can help to identify problems long before they become larger issues. Fixing issues earlier in the development process will most likely be an easier task since there has not been enough time to develop code that relies on parts of these issues. The key point here is to integrate a locked-down security philosophy from the inception of the product, and taking the malice of potential hackers into account while implementing every feature.

Why:

Losing user data could leave companies open to financial litigation at the very least, perhaps even legal litigation in some circumstances or in the future as laws around technology and privacy evolve. As seen in the Fortnite example [6], accounts can be re-sold on black markets causing a wide number of repercussions for users who trusted their private information to companies. This creates a very large incentive for public-facing products to protect their users from a very destructive breach of private information.

Performing SQL injection without permission can lead to severe repercussions and is a form of unethical hacking. Whether or not a hacker intends to release sensitive data or interact with an application or network maliciously, it is not a good idea to try SQL injection on proprietary solutions without permission.

Some companies will hire proficient hackers to perform SQL injection as a form of testing, allowing them to try their best to find any vulnerabilities so that a solution can be patched before someone unintended finds the exploit and uses it maliciously. This is a form of ethical hacking and can be a lucrative career for some.

How:

This vulnerability is usually created when a website allows user input of SQL commands to be passed directly to the database and run on the database [7]. There are several types of SQL injection, including: union-based SQL injection, error-based SQL injection, and blind SQL injection [8]. All forms of SQL injection can become detrimental to a product, resulting in loss of data, or unintended access to sensitive data. SQL injection vulnerabilities are preventable and arise due to negligence of proper security when developing applications that accept user data.

Solution Description - How to prevent SQL injection attacks

Generally, organizations should apply routine automated testing, much like the programs that hackers use, which can be used to test and catch a potential SQL injection before they occur. Paying for a regular security audit by an external firm that specializes in web security and SQL injections is an excellent option as well.

Security measures should be implemented in as many ways as reasonably possible. Like an onion, application security and its many implementations can be broken down into layers, allowing each part of an application to be more resistant to varied attacks. This way, an attacker will have to bypass multiple preventative measures implemented by the developers (ex. A hacker could get find a vulnerability in the application layer, but database security could prevent them from doing anything substantial)

More specifically, organizations can prevent SQL injections by [8]:

- Cleaning/validating data during the process of accepting the user input in the web page's form text field. Data is tested against a list of known illegal values.
- Using prepared Statements with parameterized queries which forces a hard black-and-white distinction between code and data, eliminating the chance of data being run as code.
- Using stored procedures in much the same manner as mentioned above for prepared statements, the difference being that they are stored in the database itself, not in the code, and they are slightly less secure against avoiding SQL injection since implementations may vary. Features like dynamic SQL generation or database owner privileges can be enabled by the database administrator.

- In designing the database, ensure that there is no concatenation of fields that would possibly enable the occurrence of SQL commands to be executed. A string with escape characters in it could use this concatenation to break out of the string and begin executing code from within.

Hiring a white hat hacker to perform penetration testing is another great idea to prevent SQL injection, allowing for experienced hackers to check for exploits in the system while working in a controlled environment.

Solution Demonstration Description

For our solution demonstration, we created a faux web store sign-in using a database and a frontend. The database was created using MySQL with MySQL WorkBench. The frontend had a web form (Figure 1) for that included a series of buttons. Some of the buttons were connected to code that was purposefully unsecured and vulnerable to SQL injection for our demonstration. Others were using our solution that is hardened against injection. In contrast to our proposal, we decided to only create one database (instead of two, one hardened and one not) and just code different avenues to access it.

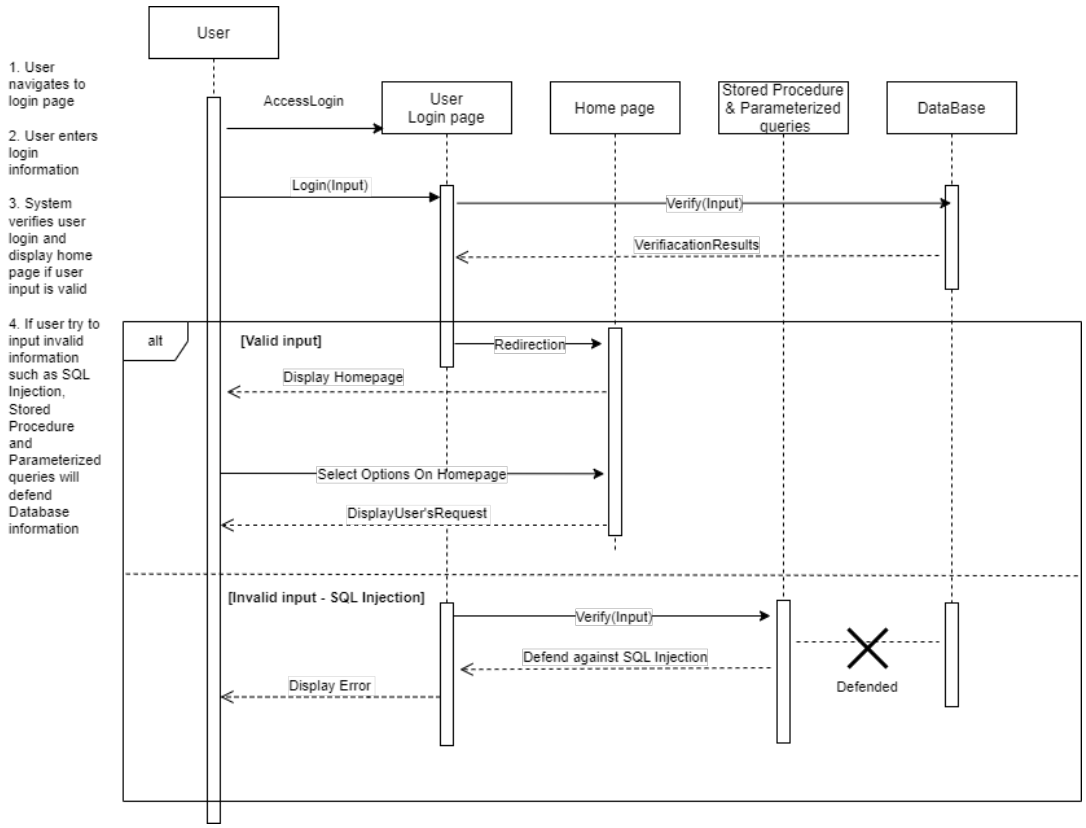


Figure 1. Sequence diagram of our solution to SQL injection.

The web page frontend was created using Python for the underlying code and Tinker for the GUI elements. The frontend code uses MySQL Connector/Python to communicate with the backend database.

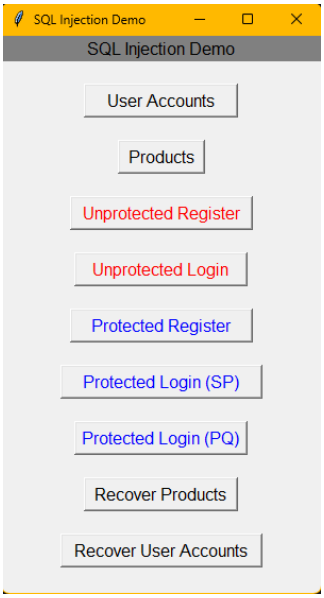


Figure 2. Our frontend main page.

Our mock web form simulates a user either registering a username and password, or signing-in using an existing username and password stored in the database. The registration consists of the user supplying a username, password, first and last name, and phone number. Once a user successfully logs in, the user's information will be displayed on screen. They are not allowed to view the information of another user without providing said user's login credentials.

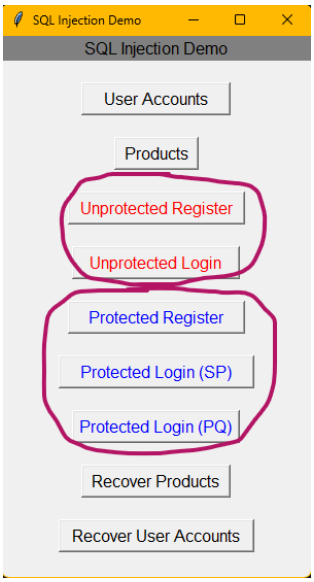


Figure 3. The unprotected and protected applications are circled in pink.

The two applications (unprotected and protected) will write to the same table, storing both user logins and a list of products. The ‘users’ table will store each user’s: username, password, phone number, first and last name, along with a unique ID. The ‘products’ table will store only the product ID and product name. The only difference between the two application paths will be how the user-supplied data is handled. The first part of the solution for the protected application will use a combination of triggers and stored procedures when inserting or retrieving user-supplied data, as well as making sure this data is sanitized and never concatenated. The second part of the solution – the one that ensures that SQL injection cannot happen – uses parameterized queries created with prepared statements.

Baseline Snapshot of Application & Database

Baseline of products in database product table

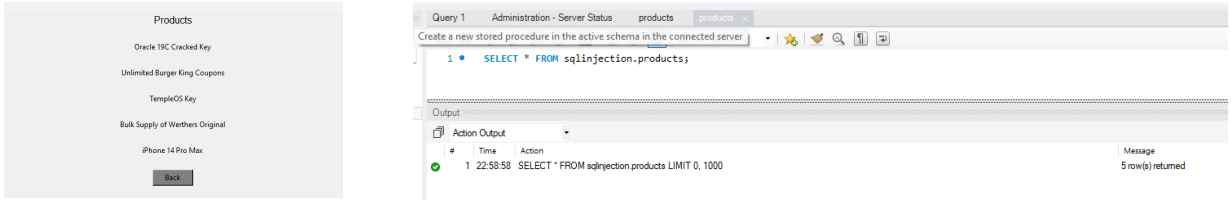


Figure 4 & 5. Screenshots of our products on the webpage and in the database.

Baseline of users in database

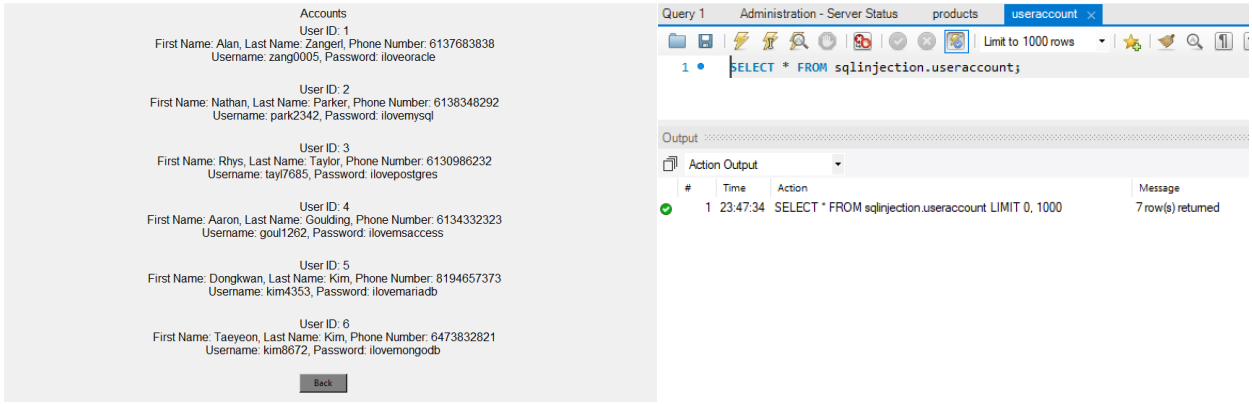


Figure 6 & 7. Screenshots of our users on the webpage and in the database.

The demonstration itself consists of two phases, connecting using the unprotected methods, and then the protected ones.

First, a normal user completes registration and sign-in via the vulnerable/unprotected method to establish a baseline. They are greeted with their first and last name at the top of the page, then a list of all their user credentials currently stored in the database.

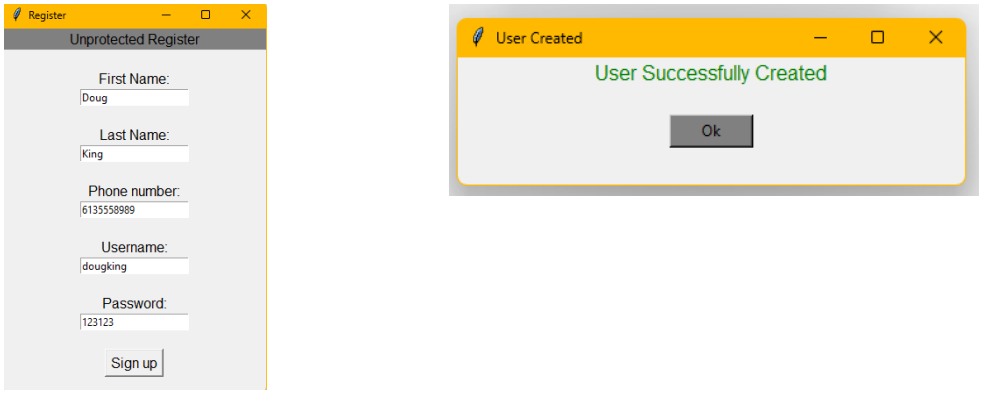


Figure 8 & 9. Screenshots of a benign user registering successfully to our store app.

Below we can see on the updated users list that user “dougking” is added to the store users.

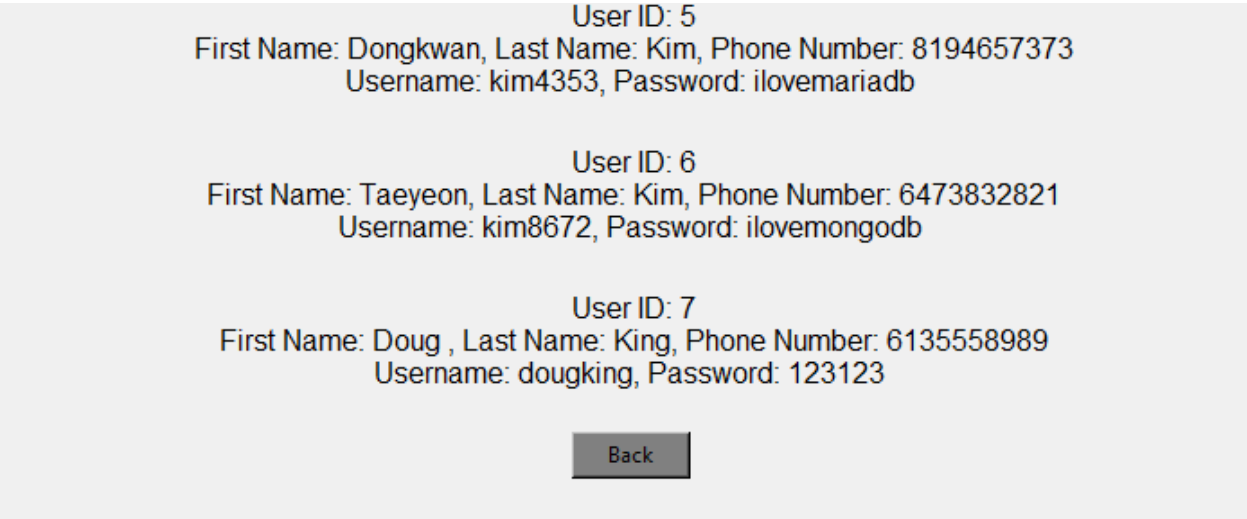
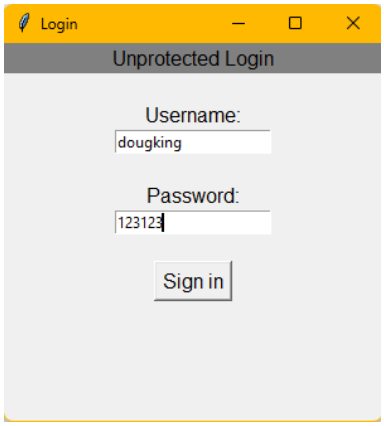


Figure 10. A screenshots of our benign user showing in user list on webpage.

Benign user “dougking” logs into the unprotected application. Everything is good and working normally, as expected.



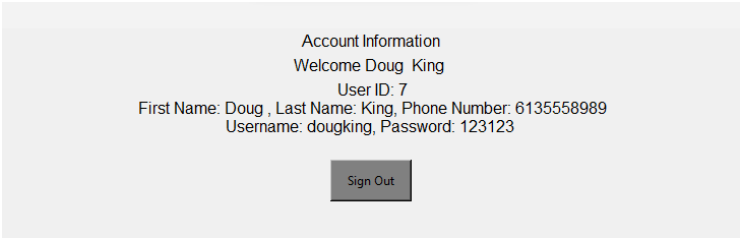
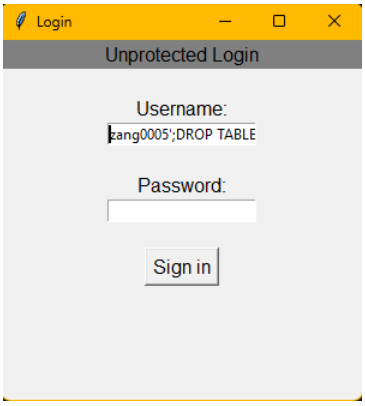


Figure 11 & 12. Screenshots of a benign user logging-in successfully to our store app.

Malicious User Drop Table on Login via Unprotected

Next the malicious user zang0005 signs-in on the unprotected portion of the application.



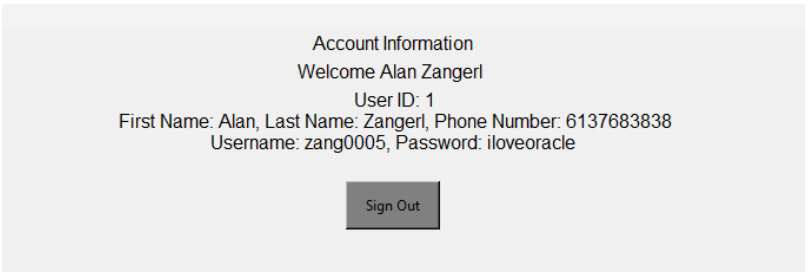


Figure 13 & 14. Screenshots of a malicious user logging-in successfully and executing SQL code.

They use their username in the login to drop products table. No password is needed with this command zang0005';DROP TABLE products;

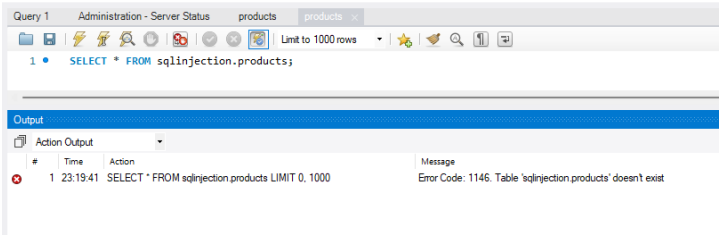


Figure 15 & 16. Screenshots of products webpage and database output after the products table is dropped.

Above we can see that without even supplying a password, after normal registration, the user was able to drop the entire products table by logging into the unprotected application, without even supplying their password. The product list displays nothing, it gets an error trying to read from an empty table on the database.

Malicious User Creates a Stored Procedure via Unprotected

Next the malicious user decides to create a stored procedure to call at some later point in time. They use the command below at the username input at the unprotected login dialogue to create a stored procedure containing a DROP TABLE USERACCOUNT query.

```
zang0005';CREATE PROCEDURE sql_injection2() BEGIN DROP TABLE useraccount;END;
```

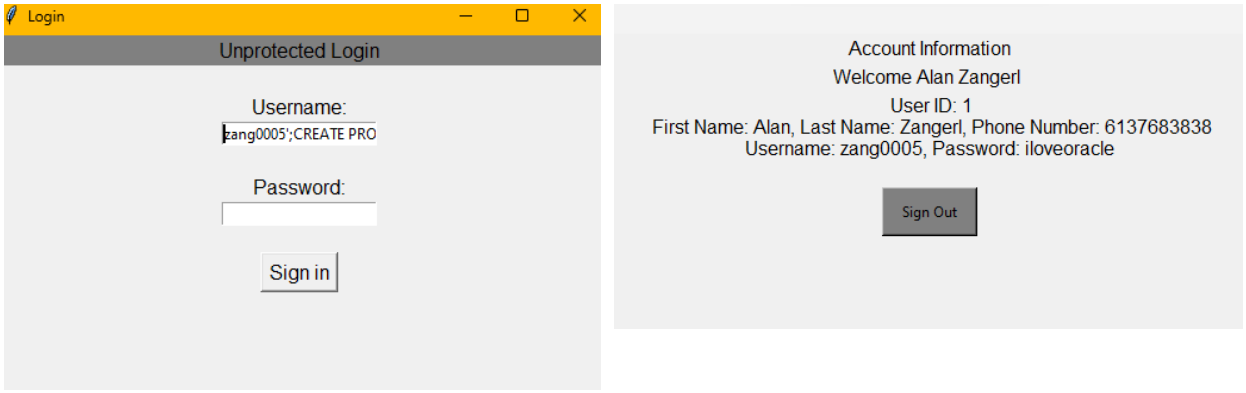


Figure 17 & 18. Screenshots of a malicious user logging-in and executing SQL code to run a stored procedure.

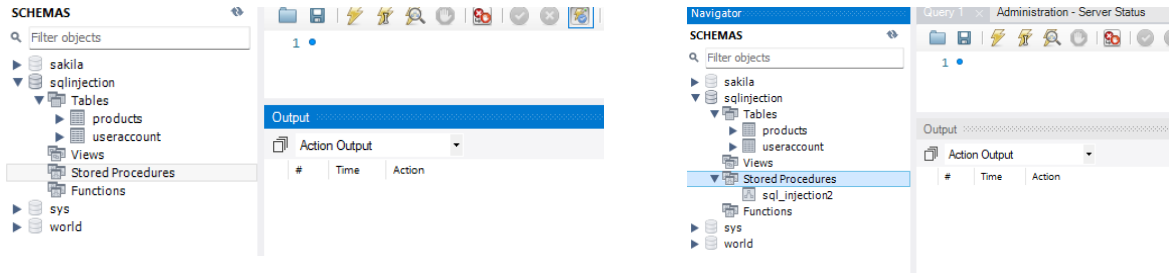


Figure 19 & 20. Screenshots of the Stored Procedures section our database, before (left) and after (right).

The user can now login and call the stored procedure using this: `zang0005';CALL sql_injection2();`

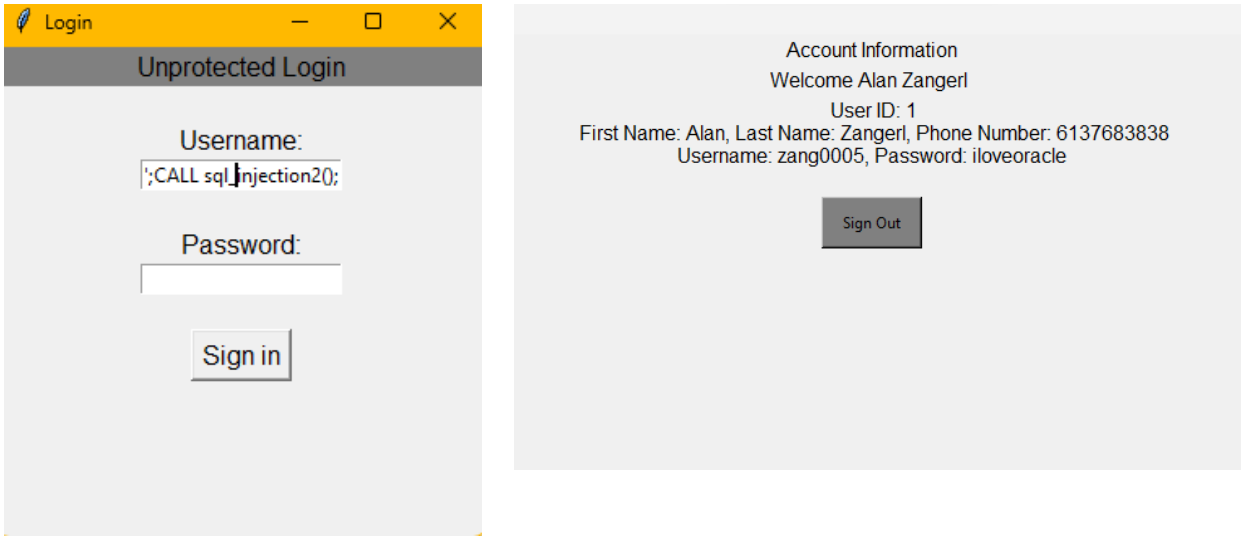


Figure 21 & 22. Screenshots of the Stored Procedures section our database, before (left) and after (right).

As shown below, all the user accounts are gone from the database after the stored procedure is called.

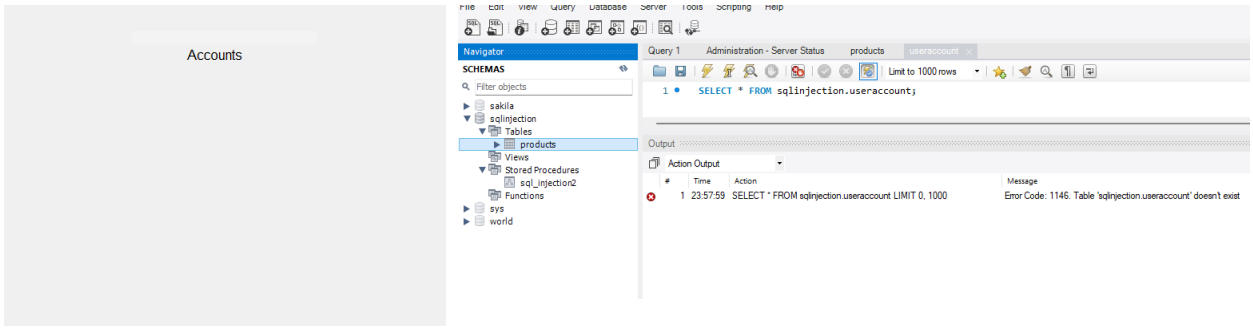


Figure 23 & 24. Screenshots of accounts webpage and database output after the accounts table is dropped.

This method of creating malicious stored procedures on the database has sweeping implications. As demonstrated, data can be maliciously removed causing headaches for the DBA, but it could also be maliciously retrieved if the stored procedure is modified. A large-scale database may have tens if not hundreds of stored procedures contained within it. Who is going to notice one more being created? The malicious user could even drop the stored procedure via another SQL injection, leaving very little trace of an injection even occurring.

Malicious User Drop Table on Login via Protected (SP)

The second phase of the demonstration follows the same order as the first, however, this time the malicious users will be connecting via the protected methods.

The first approach has the login protected by a stored procedure that checks the user input for SQL commands and rejects them if found. This process is known as ‘data sanitization’, stopping malicious user data before reaching the database tables.

```
CREATE DEFINER=`cst8276`@`localhost` PROCEDURE `sql_prevention` (IN un VARCHAR(100), IN pwd VARCHAR(30),   OUT sqlerror int)
BEGIN
		DECLARE checked_un INT;
  DECLARE checked_pwd INT;
  SET checked_un = check_username(un);
  SET checked_pwd = check_password(pwd);
		IF (checked_un AND checked_pwd = 1) THEN
			SELECT * FROM sqlinjection.useraccount WHERE username = un AND password = pwd;
		ELSE
			SET sqlerror = 0;
    SELECT @sqlerror;
  END IF;
CREATE DEFINER=`cst8276`@`localhost` FUNCTION `check_username`(
  un VARCHAR(100))
RETURNS INT
DETERMINISTIC
BEGIN
		DECLARE username_injection INT;
		SET username_injection = 1;
		IF (un like '%''%') THEN
			SET username_injection = 0;
		ELSEIF (un like '%--%') THEN
			SET username_injection = 0;
		ELSEIF (un like '%/*%') THEN
			SET username_injection = 0;
  ELSEIF (un like '%*/%') THEN
			SET username_injection = 0;
  ELSEIF (un like '%@') THEN
			SET username_injection = 0;
		ELSEIF (un like '%@@%') THEN
			SET username_injection = 0;
		ELSEIF (un like '%char%') THEN
			SET username_injection = 0;
		ELSEIF (un like '%nchar%') THEN
			SET username_injection = 0;
		ELSEIF (un like '%varchar%') THEN
			SET username_injection = 0;
		ELSEIF (un like '%nvarchar%') THEN
			SET username_injection = 0;
		ELSEIF (un like '%select%') THEN
			SET username_injection = 0;
		ELSEIF (un like '%insert%') THEN
			SET username_injection = 0;
		ELSEIF (un like '%update%') THEN
			SET username_injection = 0;
		ELSEIF (un like '%delete%') THEN
			SET username_injection = 0;
				...
			SET username_injection = 0;
		ELSEIF (un like '%sys%') THEN
			SET username_injection = 0;
```

```
    END IF;
    RETURN username_injection;
```

The malicious user zang0005 signs-in on the protected (SP) portion of the application. This login uses our SQL Injection stored procedure to check for SQL commands in the text strings.

Figure 25. Screenshot of the Protected Login (SP).

Below we can see that the products table remains unaffected by the malicious user's attempted SQL injection code.

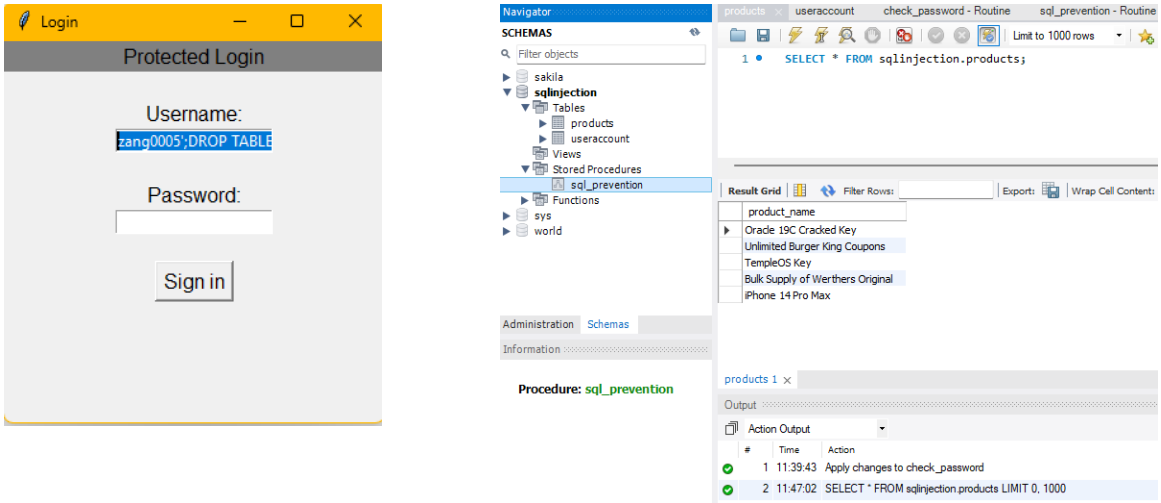


Figure 26 & 27. Screenshots of the malicious user trying to login and run SQL commands (left) and the products table remaining unaffected on the database thanks to our stored procedure solution.

Malicious User Creates a Stored Procedure via Protected (SP)

Like the unprotected section, the malicious user attempts to create a stored procedure upon login by inserting the SQL command into the username field:

```
zang0005';CREATE PROCEDURE sql_injection2() BEGIN DROP TABLE useraccount;END;
```

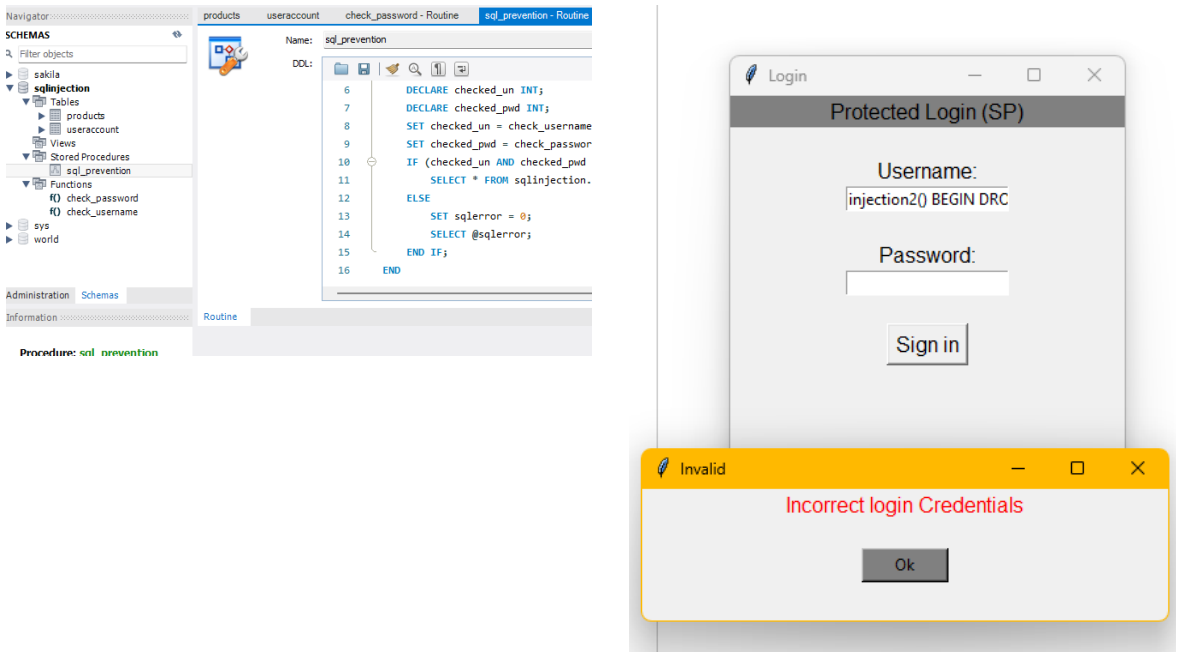


Figure 28 & 29. Screenshots of the malicious user trying to login and create a stored procedure. Our stored procedure solution prevents both the login attempt and creation of the stored procedure on the database.

However, the login attempt will be rejected by the database as the user input fields contain characters present in SQL commands (see above code section). No impact to the database.

Malicious User Attempts Drop Table on Login via Protected (PQ)

Our login parameterized query:

```
def protected_db_login(username, password):
    db_conn = db_Connection()
    cursor = db_conn.cursor()
    sql = "SELECT * FROM useraccount WHERE username = %s and password = %s;"
    cursor.execute(sql, (username, password))
    return cursor.fetchall()
```

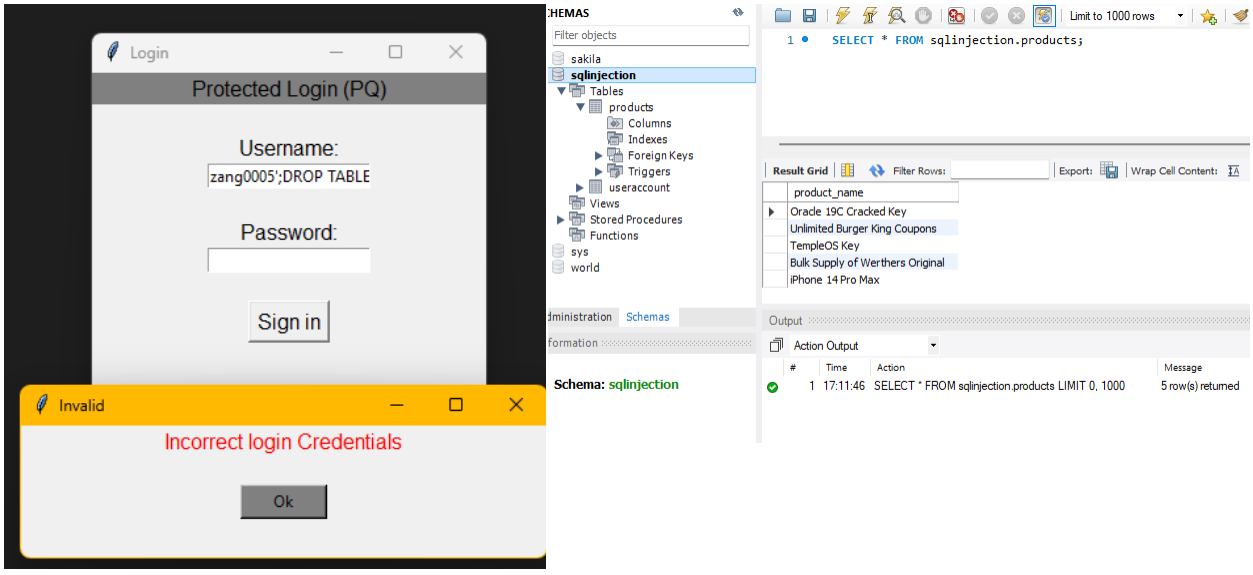


Figure 28 & 29. Screenshots of the malicious user trying to login and run SQL commands (left) and the products table remaining unaffected on the database thanks to our parameterized query.

The SQL injection login zang0005';DROP TABLE products; has no effect as it is handled by the parameterized query which literally looks for a username "zang0005';DROP TABLE products;". Both the login attempt, and the SQL command to drop the product tables are rejected.

Malicious User Attempts Creates a Stored Procedure via Protected (PQ)

Our register new user parameterized query protected code:

```
def protected_db_create_account(lastname, firstname, phone, username, password):
    db_conn = db_Connection()
    cursor = db_conn.cursor()
    sql = "INSERT INTO useraccount(last_name, first_name, username, phone, password) VALUES (%s, %s, %s, %s, %s);"
    cursor.execute(sql, (lastname, firstname, phone, username, password))
    db_conn.commit()
    cursor.close()
    db_conn.close()
```

First, the malicious user registers a user account on our frontend app seen below.

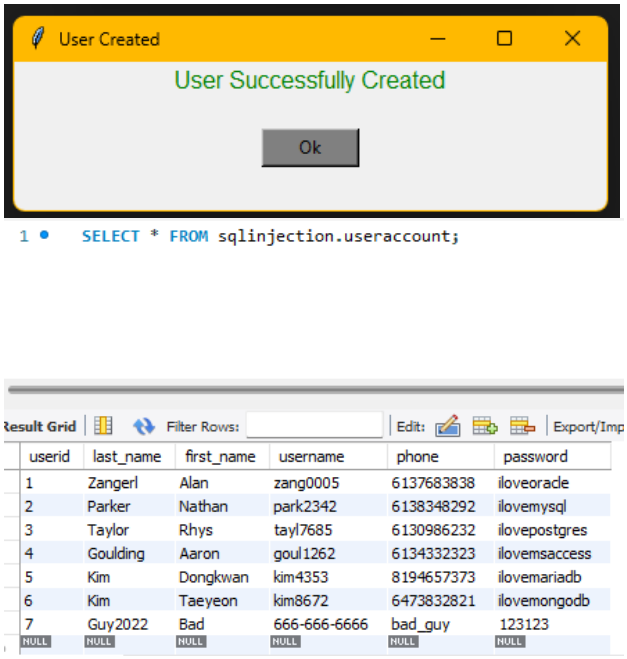


Figure 30,31 & 32. Screenshots of the malicious user registering to our fake store app.

The malicious user then attempts to login and attach the SQL command to create a stored procedure on the database. In doing so, they could call the stored procedure containing harmful SQL commands at their whim. The login user they enter is:
bad_guy';CREATE PROCEDURE sql_injection2() BEGIN DROP TABLE useraccount;END;

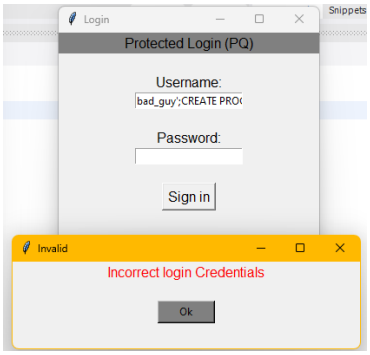


Figure 33. Screenshot of the malicious failing to login.

The login attempt is rejected and the SQL to create a stored procedure is not executed. As we can see below, no stored procedure is created in database:

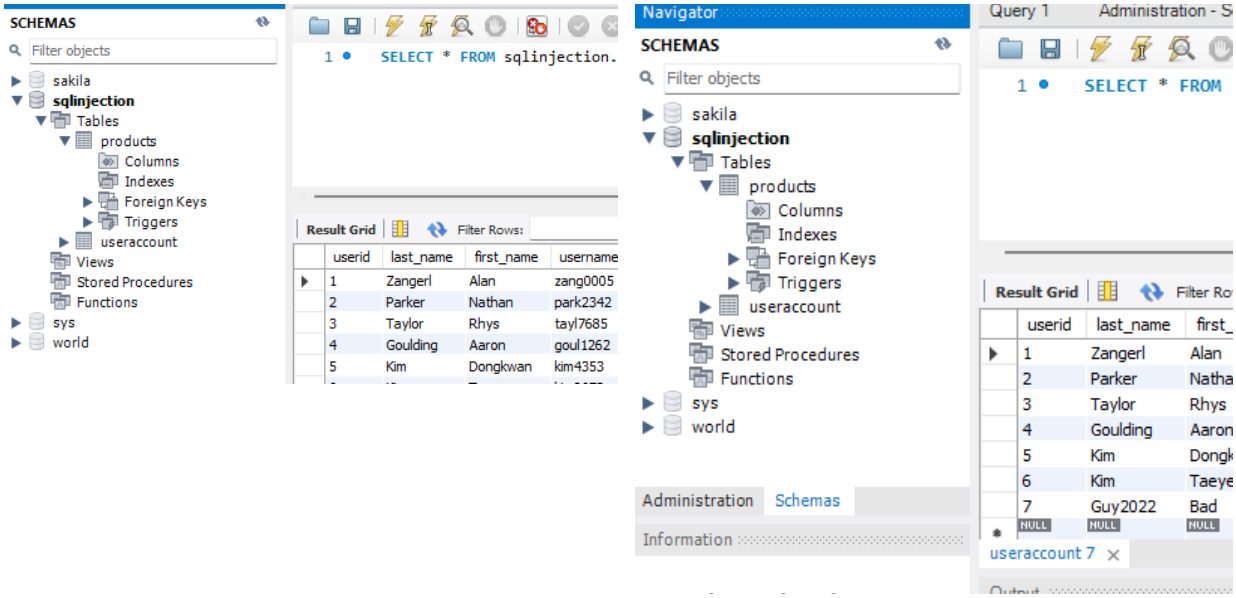


Figure 34 & 35. Screenshot of the stored procedures section of the database before (left) and after (right).

Lessons Learned

The only risk that needed to be mitigated during this project was our lack of knowledge of the subject material. With diligent research and experimentation, we were able to overcome this shortfall to create our solution and report.

Our overall experience with the project was good, as we were able to harness our programming skills to create the solution and learn more about a pressing issue facing DBAs. Our decision to use simple tools such as Python and MySQL Workbench, meant we could spend less time debugging complex systems and more time learning the source material. The big takeaway is that SQL injection is not hard to prevent, but you need to understand the why and how of it happening, and to form a habit of using parameterized queries early in your programming career. Although much SQL injection can be prevented with a well-constructed stored procedure, the best approach is to use parameterized queries to avoid costly disasters down the road that could impact both your career, your employers, and their customers.

References

[1]	R. Rachwald, "Imperva Blog," Imperva, 30 05 2011. [Online]. Available: https://web.archive.org/web/20110629080422/http://blog.imperva.com/2011/05/pbs-breached-how-hackers-probably-did-it.html. [Accessed 27 09 2022].
[2]	Reuters, "Hackers claim Tupac still alive in fake PBS news story," National Post, 31 05 2011. [Online]. Available: https://nationalpost.com/entertainment/hackers-claim-tupac-still-alive-in-fake-pbs-news-story. [Accessed 27 09 2022].
[3]	D. Storm, "eam GhostShell hacktivists dump data from US universities and hundreds of sites," Computer World, 01 07 2015. [Online]. Available: https://www.computerworld.com/article/2943255/team-ghostshell-hacktivists-dump-data-from-us-universities-and-hundreds-of-sites.html. [Accessed 27 09 2022].
[4]	Techie News UK, "Researcher finds SQL injection vulnerability on Tesla website," Techie News UK, 24 02 2014. [Online]. Available: https://techienews.co.uk/researcher-finds-sql-injection-vulnerability-tesla-website. [Accessed 27 09 2022].
[5]	Cyber Security Infotech(P) Ltd, "SQL Injection Vulnerability on Tesla Motors' Website Exposed Customer Records," Cyber Security Infotech(P) Ltd, 24 02 2014. [Online]. Available: https://csinfotechblog.wordpress.com/2014/02/25/sql-injection-vulnerability-on-tesla-motors-website-exposed-customer-records/. [Accessed 27 09 2022].
[6]	S. Khandelwal, "Fortnite Flaws Allowed Hackers to Takeover Gamers' Accounts," The Hacker News, 16 01 2019. [Online]. Available: https://thehackernews.com/2019/01/fortnite-account-hacked.html. [Accessed 27 09 2022].
[7]	N. Dill, "A Complete Guide on How SQL Injection Attacks Work," TestSuite, 30 10 2021. [Online]. Available: https://testsuite.io/how-sql-injection-works. [Accessed 27 09 2022].
[8]	A. Dizdar, "SQL Injection Attack: Real Life Attacks and Code Examples," Bright Security Inc., 08 04 2022. [Online]. Available: https://brightsec.com/blog/sql-injection-attack/#prevention-cheat-sheet. [Accessed 27 09 2022].
[9]	A. Smith, "6 best practices for application security," *The SHI Hub*, 25-Mar-2021. [Online]. Available: https://blog.shi.com/cybersecurity/application-security-best-practices/. [Accessed: 28-Sep-2022].

Appendix A – Work Plan

Deliverable	Subtask	Due Date	Marks Value	Total Work %	num of hours	Contributor 1	Contribution Amt	Contributor 2	Contribution Amt	Contributor 3	Contribution Amt
Initial Proposal		End of Week 4	1	4%							
	Write Draft					Aaron	6hrs				
	Finalize Draft					Rhys	3hrs	Nate	3hrs		
	Formatting/Editing					Rhys	2hrs				
Group Assignment Proposal		End of Week 6	4	16%							
	Rework from Feedback					Nate	5hrs	Rhys	3hrs		
	Formatting/Editing					Rhys	2hrs	Taeyeon	2hrs		
Group Written Report		Week 12-13	15	60%							
	Rework from Feedback					Rhys	5hrs				
	Write Results section					Aaron	5hrs				
	Create Sequence Diagrams					Taeyeon	2hrs	DK	2hrs		
	Formatting/Editing					Rhys	2hrs	Nate	3hrs		
Coding Solution											
	Design Initial DB and Connector					Alan	5hrs				
	Create GUI					Taeyeon	6hrs	Nate	2hrs		
	Write Stored Proc.					DK	6hrs	Nate	2hrs		
	Write Prep. Statement					Alan	3hrs				
	Test					DK	2hrs	Taeyeon	2hrs		
Group In-class Presentation		Weeks 12-13	5	20%							
	Create Presentation					Alan	2hrs	Aaron	2hrs	Rhys	2hrs