The Effectiveness of Parameterized Queries in Preventing SQL Injection Attacks at Go

Rizaldi Fatah Sidik[1], Syifa Nurgaida Yutia[2], and Rana Zaini Fathiyana[3]

[1,2,3] Telkom University, Jl. Telekomunikasi, 40257, Indonesia
[1]zaldisidik@student.telkomuniversity.ac.id

Abstract. SQL Injection attacks are one of the common security risks that occur in applications. SQL Injection cases can lead to data and sensitive information leaks, and even potential application data deletion. This research examines the effectiveness of using parameterized queries in the Go programming language as a method of prevention against SQL Injection attacks. Go provides the feature of parameterized queries by using placeholders such as question marks (?) or parameter names. Parameterized queries separate input values from SQL statements and are executed securely by the database driver. In this study, the use of parameterized queries in Go is evaluated to prevent query manipulation by users in the application. The research is conducted by testing four HTTP request operations: GET, POST, PUT, and DELETE, both before and after the use of parameterized queries. The testing results, based on Acunetix Web Vulnerability scanning, prove that all testing operations are vulnerable to SQL Injection when not using parameterized queries, while successfully mitigating SQL Injection attacks when using parameterized queries in Go.

Keywords: Go, HTTP, Parameterized Queries, SQL Injection, Web Vulnerability.

1 Introduction

SQL Injection ranks 3rd as one of the most dangerous risks identified among 94% of tested applications and 33 Common Weakness Enumeration (CWE) mapped under the Injection category, according to the consolidated findings by OWASP in 2021 regarding the Top 10 Web Application Security Risks [1]. The first documented case of SQL Injection occurred in 1988, primarily targeting retailers and banks [2]. SQL Injection can lead to data and public information leakage and even result in data deletion within applications. Such occurrences can have catastrophic consequences. SQL injection cases should not happen if technology implementers apply early prevention measures in their applications. Systems with vulnerabilities in input parameters allow attackers to retrieve data from the entire database stored on the webserver [3]. The use of PDO Parameterized Queries can prevent SQL Injection and enhance system performance [4]. A study conducted by Santiago et al. (2021) implemented validation filters on input

© The Author(s) 2023
M. D. Sulistiyo and R. A. Nugraha (eds.), *Proceedings of the International Conference on Enterprise and Industrial Systems (ICOEINS 2023)*, Advances in Economics, Business and Management Research 270,
https://doi.org/10.2991/978-94-6463-340-5_18

fields based on OWASP Stinger, a set of regular expressions, and sanitation processes, which achieved an accuracy rate of 98.4% [5].

In Golang, there are several features available to prevent SQL Injection attacks, and one of them is the use of parameterized queries [6]. Golang supports the use of parameterized queries by using question marks (?) or parameter names as placeholders for input values. Parameterized queries allow developers to provide input values as separate parameters in SQL statements, which are executed securely by the database driver. This helps prevent SQL Injection because input values are never directly concatenated into the SQL statement. Various forms of SQL Injection attacks occur when user input is not properly filtered to detect escape characters, and those characters are then passed into the SQL statement on the server [7]. However, it is important to note that while the use of parameterized queries is one preventive measure against SQL Injection, it is not a fully secure solution if not used correctly. In this research, the effectiveness of using parameterized queries in Golang to prevent SQL Injection attacks will be analyzed. This study aims to provide a better understanding of the effectiveness of using parameterized queries in Golang and inspire best practices in developing secure and resilient applications against SQL Injection attacks.

2 Architecture

In this section, the architecture used in the research will be described. The architecture employed consists of only one component, which is the backend. The fundamental architecture for conducting the research utilizes a REST API with the Go (Golang) programming language. In the REST system, four HTTP request methods are implemented, namely GET, POST, PUT, and DELETE operations [8]. The design of the REST API architecture for this research can be seen in figure 1.

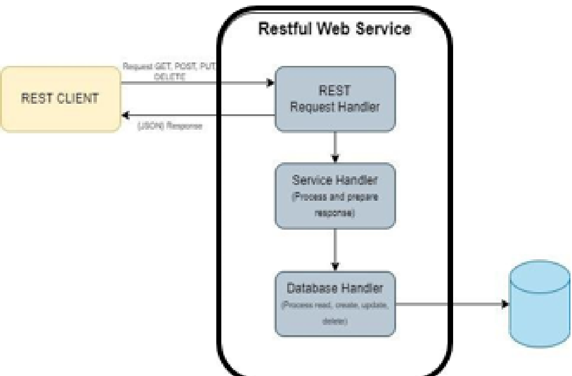

Fig. 1. System Architecture Design.

Figure 1 illustrates the process flow of the system architecture design. The flow begins when a client sends a REST API request to the Restful Web Service. This request can be an operation such as POST, GET, PUT, or DELETE. The request is sent to the Request Handler in the Restful Web Service, which is responsible for receiving and directing the request to the appropriate component for processing. Once the request is received by the Request Handler, it is then forwarded to the Service Handler to perform the business logic associated with the request. If the logic process requires interaction with the database, the process is continued to the Database Handler, which is responsible for processing database-related operations. The result from the Database Handler is returned to the Service Handler and then passed back to the Request Handler in the form of a JSON response to the REST Client.

3 Implementation

The implementation phase is the step where the previously established analysis and design are put into action to achieve the desired outcome. In this research, the implementation process will be divided into several parts:

1. Implementation of queries before parameterized queries
2. Implementation of queries after parameterized queries
3. CURL Rest API

3.1 Implementation Query Before Parameterized Query

In this section, the implementation of queries before utilizing parameterized queries will be performed for each operation (POST, GET, PUT, DELETE) to be used in the subsequent testing phase.

1. POST Operation Code

```
func InsertUser(c context.Context, param *User) (res
string, err error) {
statement := `INSERT INTO tbl_user ( name, address,
birthdate, idnum) VALUES ('`+param.Name+`',
'`+param.Address+`', '`+param.Birthdate+`',
'`+param.IDNum+`') RETURNING id`
err = database.DBConn.QueryRowContext(c, state-
ment).Scan(&res)
if err != nil {
return res, err
}
return res, err
}
```

The code snippet demonstrates the process for the POST operation with an insert query into the "tbl_user" table in the database.

2. GET Operation Code

```
func GetUserWithParam(c context.Context, name, idnum
string) ([]User, error) {
var resp []User
stmt := `select a.id, a.name, a.address, a.birthdate,
a.idnum from tbl_user a where a.deleted_at is null and
(a.idnum LIKE '`+idnum+`') `
rows, err := database.DBConn.QueryContext(c, stmt) if
err != nil {
return resp, err
}
defer rows.Close() for rows.Next() {
var tch User
err :=  rows.Scan(&tch.ID, &tch.Name, &tch.Address,
&tch.Birthdate, &tch.IDNum)
if err != nil {
return resp, err
}
resp = append(resp, tch)
}
return resp, nil
}
```

The code snippet demonstrates the process for the GET operation with a select query on the "tbl_user" table in the database.

3. PUT Operation Code

```
func UpdateUser(c context.Context, param *User) (res
string, err error) {
statement := `update tbl_user set
name='`+param.Name+`',adddress='`+param.Ad-
dress+`',birthdate='`+param.Birthdate+`' where id-
num='`+param.IDNum+`' RETURNING id`
err = database.DBConn.QueryRowContext(c, state-
ment).Scan(&res)
if err != nil {
return res, err
}
return res, err
}
```

The code snippet demonstrates the process for the PUT operation with an update query on the "tbl_user" table in the database.

4. DELETE Operation Code

```
func DeleteUser(c context.Context, param *User) (res
string, err error) {
statement := `delete from tbl_user where id-
num='`+param.IDNum+`' RETURNING id`
err = database.DBConn.QueryRowContext(c, state-
ment).Scan(&res)
if err != nil {
return res, err
}
return res, err
}
```

The code snippet demonstrates the process for the DELETE operation with a delete query on the "tbl_user" table in the database.

3.2 Implementation Query After Parameterized Query

In this section, the implementation of queries after using parameterized queries will be carried out for each operation (POST, GET, PUT, DELETE) that will be used in the subsequent testing phase.

1. POST Operation Code

```
func InsertUser(c context.Context, param *User) (res
string, err error) {
statement := `INSERT INTO tbl_user ( name, address,
birthdate, idnum) VALUES ($1, $2, $3, $4) RETURNING
id`
err = database.DBConn.QueryRowContext(c, statement,
param.Name, param.Address,
param.Birthdate, param.IDNum).Scan(&res)
if err != nil {
return res, err
}
return res, err
}
```

The code snippet demonstrates the process for the POST operation with an insert query into the "tbl_user" table in the database.

2. GET Operation Code

```
func GetUserWithParam(c context.Context, name, idnum
string) ([]User, error) {
var resp []User
stmt := `select a.id, a.name, a.address, a.birthdate,
a.idnum from tbl_user a where a.deleted_at is null and
(a.idnum LIKE $1) `
rows, err := database.DBConn.QueryContext(c, stmt,
idnum)
if err != nil {
return resp, err
}
defer rows.Close() for rows.Next() {
var tch User
err :=  rows.Scan(&tch.ID, &tch.Name, &tch.Address,
&tch.Birthdate, &tch.IDNum)
if err != nil {
return resp, err
}
resp = append(resp, tch)
}
return resp, nil
}
```

The code snippet demonstrates the process for the GET operation with a select query on the "tbl_user" table in the database.

3. PUT Operation Code

```
func UpdateUser(c context.Context, param *User) (res
string, err error) {
statement := `update tbl_user set name=$1, address=$2,
birthdate=$3 where idnum=$4 RETURNING id`
err = database.DBConn.QueryRowContext(c, statement ,
param.Name, param.Address, param.Birthdate, param.ID-
Num).Scan(&res)
if err != nil {
return res, err
}
return res, err
}
```

The code snippet demonstrates the process for the PUT operation with an update query on the "tbl_user" table in the database.

4. DELETE Operation Code

```
func DeleteUser(c context.Context, param *User) (res
string, err error) {
statement := `delete from tbl_user where idnum=$1
RETURNING id`
err = database.DBConn.QueryRowContext(c, statement,
param.IDNum).Scan(&res)
if err != nil {
return res, err
}
return res, err
}
```

The code snippet demonstrates the process for the DELETE operation with a delete query on the "tbl_user" table in the database.

3.3 URL Rest API

URL (Uniform Resource Locator) is a unique address on the internet used to identify a website [9]. CURL (Client URL) Rest API is used as the address for testing in the subsequent phase. The operations used in CURL Rest API are POST, GET, PUT, and DELETE.

Table 1. Rest API CURL.

Operation	CURL
POST	curl --location 'http://127.0.0.1:8024/api/createuser' \ --header 'Content-Type: application/json' \ --data '{ "name":"BAMBANG", "address":"meruyung", "birthdate":"2003-09-09", "idnum":"111111111111111" }'
GET	curl --location 'http://127.0.0.1:8024/api/user?idnum=3275090809940006'
PUT	curl --location --request PUT 'http://127.0.0.1:8024/api/updateuser' \ --header 'Content-Type: application/json' \ --data '{ "name":"BAMBANG", "address":"meruyunggggg", "birthdate":"2003-09-09", "id-num":"111111111111111" }'
DELETE	curl --location --request DELETE 'http://127.0.0.1:8024/api/deleteuser' \ --header 'Content-Type: application/json' \ --data '{ "idnum":"111111111111111" }'

Table 1 shows a list of URLs that will be used in the research process. The research will investigate the effectiveness of using parameterized queries for each URL in the list.

4 Parameterized Queries Effectiveness Testing

In this research, testing will be conducted to measure the effectiveness of using parameterized queries in Rest API. The testing will be performed using Rest API URLs with operations such as POST, GET, PUT, and DELETE. The testing will be carried out twice, before and after implementing parameterized queries. Vulnerability scanning for SQL Injection will be performed using the Acunetix Web Vulnerability Scanner tool. Acunetix Web Vulnerability Scanner is recognized as one of the best tools for detecting SQL Injection security vulnerabilities [10].

4.1 Rest API SQL Injection Scanning Before Parameterized Query

In this section, the results of vulnerability scanning for SQL Injection on the Rest API before implementing parameterized queries will be presented. The scanning process was conducted using the Acunetix Web Vulnerability Scanner tool. The testing results for the POST, GET, PUT, and DELETE operations can be observed in Figure 2 through Figure 5.

Acunetix Threat Level 3

One or more high-severity type vulnerabilities have been discovered by the scanner. A malicious user can exploit these vulnerabilities and compromise the backend database and/or deface your website.

Alerts distribution

Total alerts found	1
High	1
Medium	0
Low	0
Informational	0

Fig. 2. Report SQL Injection Scanning Operation GET Before Parameterized Query.

Figure 2 illustrates the results of vulnerability scanning for the GET operation before implementing parameterized queries. The absence of parameterized queries led to the detection of SQL Injection vulnerabilities in the GET operation, specifically through the "idnum" parameter.

Acunetix Threat Level 3

One or more high-severity type vulnerabilities have been discovered by the scanner. A malicious user can exploit these vulnerabilities and compromise the backend database and/or deface your website.

Alerts distribution

Total alerts found	4
High	4
Medium	0
Low	0
Informational	0

Fig. 3. Report SQL Injection Scanning Operation POST Before Parameterized Query.

Figure 3 displays the results of vulnerability scanning for the POST operation before implementing parameterized queries. Without using parameterized queries, the scanning results indicate the detection of SQL Injection vulnerabilities in the POST operation through parameters such as "name," "address," "birthdate," and "idnum".

Acunetix Threat Level 3

One or more high-severity type vulnerabilities have been discovered by the scanner. A malicious user can exploit these vulnerabilities and compromise the backend database and/or deface your website.

Alerts distribution

Total alerts found	4
High	4
Medium	0
Low	0
Informational	0

Fig. 4. Report SQL Injection Scanning Operation PUT Before Parameterized Query.

Figure 4 presents the results of vulnerability scanning for the PUT operation before implementing parameterized queries. The scanning results indicate the detection of SQL Injection vulnerabilities in the PUT operation through parameters such as "name," "address," "birthdate," and "idnum" when parameterized queries are not utilized.

Figure 5 presents the results of vulnerability scanning for the DELETE operation before implementing parameterized queries. The scanning results indicate the detection of SQL Injection vulnerabilities in the DELETE operation, specifically through the "idnum" parameter.

Acunetix Threat Level 3

One or more high-severity type vulnerabilities have been discovered by the scanner. A malicious user can exploit these vulnerabilities and compromise the backend database and/or deface your website.

Alerts distribution

Total alerts found	1
High	1
Medium	0
Low	0
Informational	0

Fig. 5. Report SQL Injection Scanning Operation DELETE Before Parameterized Query.

4.2 Rest API SQL Injection Scanning After Parameterized Query

In this section, the results of vulnerability scanning for SQL Injection on the Rest API after implementing parameterized queries will be presented. The scanning process was conducted using the Acunetix Web Vulnerability Scanner tool.

The results of the vulnerability scanning demonstrate the effectiveness of implementing parameterized queries in mitigating SQL Injection vulnerabilities. With the usage of parameterized queries, the scanning tool did not detect any SQL Injection vulnerabilities in the Rest API.

The testing results for the POST, GET, PUT, and DELETE operations can be seen in Figure 6 to Figure 9.

Acunetix Threat Level 0

No vulnerabilities have been discovered by the scanner.

Alerts distribution

Total alerts found	0
High	0
Medium	0
Low	0
Informational	0

Fig. 6. Report SQL Injection Scanning Operation GET After Parameterized Query.

Figure 6 illustrates the results of vulnerability scanning for the GET operation after implementing parameterized queries. The utilization of parameterized queries resulted in the absence of SQL Injection vulnerabilities detected in the GET operation.

Acunetix Threat Level 0

No vulnerabilities have been discovered by the scanner.

Alerts distribution

Total alerts found	0
High	0
Medium	0
Low	0
Informational	0

Fig. 7. Report SQL Injection Scanning Operation POST After Parameterized Query.

Figure 7 displays the results of vulnerability scanning for the POST operation after implementing parameterized queries. The utilization of parameterized queries resulted in the absence of SQL Injection vulnerabilities detected in the POST operation.

Acunetix Threat Level 0

No vulnerabilities have been discovered by the scanner.

Alerts distribution

Total alerts found	0
High	0
Medium	0
Low	0
Informational	0

Fig. 8. Report SQL Injection Scanning Operation PUT After Parameterized Query.

Figure 8 presents the results of vulnerability scanning for the PUT operation after implementing parameterized queries. The utilization of parameterized queries resulted in the absence of SQL Injection vulnerabilities detected in the PUT operation.

Acunetix Threat Level 0

No vulnerabilities have been discovered by the scanner.

Alerts distribution

Total alerts found	0
High	0
Medium	0
Low	0
Informational	0

Fig. 9. Report SQL Injection Scanning Operation DELETE After Parameterized Query.

Figure 9 depicts the results of vulnerability scanning for the DELETE operation after implementing parameterized queries. The utilization of parameterized queries resulted in the absence of SQL Injection vulnerabilities detected in the DELETE operation.

5 Conclusion

Based on the test results and research, it can be concluded that the use of parameterized queries in the Go (Golang) programming language is effective in preventing SQL Injection attacks. The effectiveness value is measured based on the results of vulnerability scanning tests before and after using parameterized queries in the application code. The scanning results before using parameterized queries showed the detection of SQL Injection vulnerabilities in all parameters in each Rest API operation. Then, the scanning results after using parameterized queries showed the absence of SQL Injection vulnerabilities in each Rest API operation. The application will automatically protect itself from SQL Injection attacks if it uses parameterized queries. Parameterized queries convert user input values into ordinary data rather than part of the SQL query structure. This ensures that this step prevents query manipulation by users in the application. This research is expected to provide a better understanding of the effectiveness of using parameterized queries in Golang and inspire best practices in developing secure and resilient applications against SQL Injection attacks.

References

1. OWASP, 2021, Top 10 Web Application Security Risk, [Online]. Available: https://owasp.org/www-project-top-ten/. Accessed: Nov. 24, 2022.
2. Geeksforgeeks, 2021, Mitigation of SQL Injection Attack using Prepared Statements, [Online]. Available: https://www.geeksforgeeks.org/mitigation-sql-injection-attack-using-prepared-statements-parameterized-queries/. Accessed: Jun. 16, 2023.
3. A.Djalil, 2020, Analisa Serangan SQL Injection pada Server Pengisian Kartu Rencana Studi Online. Jurnal AIKOM Ternate.
4. Castillo, R. E., Caliwag, J. A., Pagaduan, R. A., & ... (2019). Prevention of SQL injection attacks to login page of a website application using prepared statement technique. Proceedings of the 2019 …. https://doi.org/10.1145/3322645.3322704.

5. Santiago Ibarra-Fiallos et al, "Effective Filter for Common Injection Attacks in Online Web Applications", IEEE Access, vol.9, 2021.
6. GO, 2023, Querying for data, [Online]. Available: https://go.dev/doc/database/querying. Accessed: Jun. 4, 2023.
7. H.F.Herdiyatmoko, "Back-End System Design Based On Rest API", Journal of TEKINKOM, vol.5, no.1, Juni 2022.
8. Vugar Abdullayev, Dr. Alok Singh Chauhan,. "SQL Injection Attack: Quick View", Mesopotamian Journal of Cybersecurity, vol.2023, pp.30- 34, 2023
9. Saputra A, Astuti DY, 2018, Analisis Pengaruh Struktur HTML Terhadap Rangking Search Engine Result Page. J. Mantik Penusa 2: 34-67.
10. M. Ula, "Evaluasi Kinerja Software Web Penetration Testing", TECHSI– J. Tek. Inform., vol.11, no.3, p.336, 2019.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (http://creativecommons.org/licenses/by-nc/4.0/), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.