SecureSkyScholars - A combination method for preventing SQL Injection Attack

Chandrima Mukherjee
220990378
Dr. Joseph Doyle
Master in Computer Science

***Abstract*—One of the most well-known web hacking attacks is SQL injection (SQLI). According to the OWASP organization (Open Web Application Security Project), SQL injection is the third biggest threat to web application security on their list of the Top 10 OWASP for 2021. In this paper, I have presented a student management system - SecureSkyScholars, which has combined approach to combat SQL injection attacks. To improve overall security, the app combines client-side and server-side defences. Server-side defences utilise parameterized queries and stored procedures, whereas client-side defences use input validation and sanitization. The integrated approach offers thorough defence against SQL injection issues. Results from tests support its efficacy conducted by some well known penetration testing tools for SQLIA. The research makes a contribution to web application security and provides knowledge to developers and security experts.**

Keywords — SecureSkyScholars, Cloud computing, SQL injection attack, security, input validation, parameterized queries, web application firewall, data protection, vulnerability, mitigation.

I. Introduction

Organisations are increasingly depending on cloud-based platforms to store and analyse their data as a result of the broad use of cloud computing. The cloud presents new security issues, even if it has many advantages in terms of scalability, flexibility, and cost-effectiveness. Preventing SQL injection attacks, which may result in unauthorised access, data breaches, and significant harm to both enterprises and people, is one such difficulty.

An extremely common and harmful type of online application attack is SQL injection. They take use of flaws in web application code that don't adequately check user input prior to communicating with databases. Attackers can alter database searches, get around authentication safeguards, and retrieve or modify sensitive data by inserting malicious SQL statements into input areas. Cloud-based systems, where several users and apps share resources on a common infrastructure, are seriously threatened by these assaults.

This paper suggests a combined approach that combines several protective measures to address the rising worry of SQL injection attacks on the cloud. To create a multi-layered defence against SQL injection attacks, the suggested technique combines input validation, parameterized queries, and web application firewalls. Cloud service providers and application developers may greatly lower the likelihood of successful attacks and improve the overall security of their systems by putting this integrated strategy into practise.

Organisations may improve the security of their cloudbased systems and defend against the threats presented by SQL injection attacks by utilising a combined approach that combines input validation, parameterized queries, and web application firewalls. This article seeks to add to the body of knowledge on cloud security and act as a helpful resource for cloud service providers, software developers, and security experts looking for practical ways to reduce the risk of SQL injection attacks in the cloud environment.

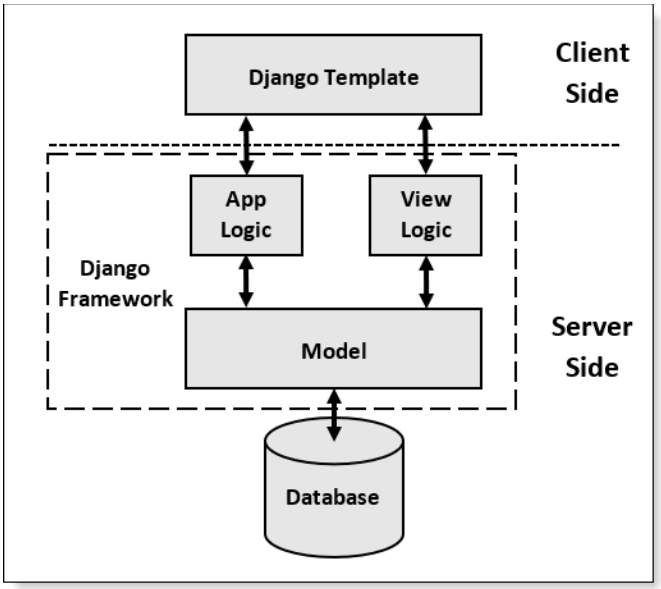

Figure 1: SecureSkyScholars architecture

II. Related Work

The following paper discusses several crucial components or approaches to avoid SQLI attacks after examining numerous electronic publications and articles from ACM, IEEE, and a host of other websites to learn in depth about the various SQLI attacks:

- From "Using Parse Tree Validation to prevent SQLI attacks" ACM, SQLI discovery was covered and the paper discussed various aspects of SQL parse tree validation.[1]

- From "The Essence of Command Injection Attacks in Web Applications" ACM, various techniques to check and prevent input query which uses SQLCHECK and grammar to validate query. [2]
- From "Using Automated Fix Generations to Secure SQL Statements" IEEE, the background of SQL statement and vulnerability was covered.[3]
- From "SQL Injection Teaching Based on SQLi-labs," IEEE, This paper takes Union SQL injection as an example to introduce the teaching implementation process based on SQLi-labs.[4]
- From " DIAVA: A Traffic-Based Framework for Detection of SQL Injection Attacks and Vulnerability Analysis of Leaked Data", a novel traffic-based SQLIA detection and vulnerability analysis framework named DIAVA, which can proactively send warnings to tenants promptly. [5]

Organisations' methods for storing, processing, and accessing data have been completely transformed by cloud computing, which also provides several advantages including scalability, cost-effectiveness, and on-demand resource allocation. But this paradigm change has also brought forth new security difficulties, with SQL injection attacks being a major worry.

For many years, SQL injection attacks have been a persistent and common danger to online applications. They take use of weaknesses in application code that doesn't adequately check user input prior to communicating with a database. Attackers are able to modify queries, get around authentication controls, and access sensitive data without authorization by inserting malicious SQL statements.

The risks connected with SQL injection attacks have increased as a result of the movement of databases and applications to the cloud. The potential attack surface is increased in cloud settings because numerous users and apps share resources on a common infrastructure. The security mechanisms used by application developers may also not be directly under the control of cloud service providers, which increases the risk of SQL injection attacks.

The goal of this paper is to address the urgent need for effective SQL injection protection strategies that are designed particularly for cloud environments. We aim to provide cloud service providers and application developers with an effective strategy to mitigate the risks associated with SQL injection attacks by developing a combined approach that integrates multiple defensive measures, including input validation, parameterized queries, and web application firewalls.

Enhancing the security of cloud-based systems, safeguarding sensitive data, and upholding user and customer confidence are the key goals of this research. Organisations may protect their apps and databases from possible breaches, monetary losses, reputational harm, and legal repercussions by establishing an effective and all-encompassing solution for SQL injection protection.

By providing useful insights and suggestions to address a crucial part of application security, this research also adds to the larger topic of cloud security. We can create proactive steps to maintain the confidentiality, integrity, and availability of data in cloud-based systems by knowing the background, threats, and difficulties connected with SQL injection attacks in the cloud.

This study intends to produce evidence-based findings that confirm the efficacy of the suggested combination strategy by careful analysis, experimentation, and assessment. This research aims to advance secure cloud computing practises and serve as a valuable resource for cloud service providers, application developers, and security professionals by addressing the unique security concerns of cloud-based systems and filling knowledge gaps.

IV. Security Challenges in Cloud Computing

Numerous advantages come with cloud computing, but there are also unique security issues that must be resolved. It's essential to comprehend these issues if you want to create security plans that work in the cloud environment. The following issues with cloud computing's security are crucial:

1 Data protection – Using cloud services requires the storage and processing of significant amounts of sensitive data, such as financial information, intellectual property, and personal data. The security, integrity, and accessibility of this data are of utmost importance. Data breaches, unauthorised access, data loss, and insufficient data encryption measures all present problems.

2 Shared Infrastructure – A shared infrastructure, where several users and applications share physical and virtual resources, is the foundation of cloud computing. Due to the multi-tenant nature of the cloud, there is a chance that data and resources may be accessed improperly. Additionally, there is a chance that cross-tenant attacks might occur, in which one compromised instance could have an impact on several others.

3 Insider Threats – Users with varying levels of access as well as cloud service providers, third-party suppliers, and other parties are all present in cloud systems. Any of these groups, including hostile administrators, negligent workers, or hacked accounts, are potential sources of insider threats. Insider threat detection and mitigation are a serious difficulty.

4 Compliance and Legal Requirements – Because cloud installations sometimes span several different countries, it can be difficult to comply with all applicable legal and regulatory requirements. Organisations must make sure that the cloud service providers they use retain data

sovereignty, comply with legal obligations, and satisfy those needs.

5 Identity and Access Management – Effective identity and access management is essential given the large number of users and apps utilising cloud services. Managing user identities, putting in place reliable authentication methods, imposing strict access rules, and dealing with challenges with identity federation and single sign-on are among the difficulties.

6 Cloud Service Provider Security – To guarantee the security of their infrastructure and services, businesses rely on cloud service providers. In order to maintain a safe cloud environment, it is essential to assess the security practises and capabilities of cloud service providers, including their protocols for incident response, data backup, and disaster recovery.

7 Data Loss and Service Availability – Disruptions to cloud services, whether brought on by technological issues, online assaults, or natural catastrophes, may cause data loss and service interruptions. To reduce these risks, it's crucial to set up reliable redundancy procedures, backup and recovery systems, and service-level agreements (SLAs).

A thorough and multifaceted strategy is needed to address these security issues. Strong encryption techniques, strict access restrictions, routine monitoring and auditing of cloud resources, and up-to-date security updates and best practises must all be implemented by organisations. The establishment of efficient incident response and communication channels, as well as a shared responsibility for security, need cooperation between cloud service providers and clients.

The process of addressing security issues is still ongoing as cloud computing develops. To improve security frameworks, create industry standards, and encourage the use of secure cloud practises, research and development initiatives are required. Organisations may confidently take use of cloud computing's advantages while protecting their sensitive data and upholding a secure computing environment by comprehending and actively minimising these security difficulties.

V. Existing Approaches to Preventing SQL Injection Attacks

The problem of avoiding SQL injection attacks in web applications has given rise to a number of solutions. These methods try to find and fix SQL injection flaws before they may be used against you. Here are a few current strategies that are often used:

1 Input Sanitization:
   - In this method, user input is validated and cleaned up before being used in SQL queries.
   - Input filtering and whitelist validation are two methods that may be used to weed out characters or patterns that could be harmful.

2 Prepared Statements and Parameterized Queries:
   - SQL statements that have placeholders for user input are known as prepared statements.
   - Parameterized queries avoid the direct concatenation of user input into the query string by separating user input from the SQL code.
   - This Django has not used any SQL queries.

3 Escaping Special Characters:
   - Web applications can stop special characters from being interpreted as a part of the SQL query by escaping them within user input.
   - Character escaping techniques and database-specific escape methods can also be used.

4 Frameworks for Object Relational Mapping (ORM):
   - The danger of SQL injection attacks is decreased by ORM frameworks, which offer an abstraction layer that manages SQL queries and parameter binding.
   - These frameworks take care of parameterization and automatically create secure SQL queries based on object-oriented programming.

5 Web Application Firewalls (WAFs):
   - WAFs serve as a barrier between a web application and the outside world, scanning incoming requests for potentially harmful patterns.
   - Based on specified security rules and signatures, they can identify and stop attempts at SQL injection.

6 Static Code Analysis:
   - Static code analysis software examines online application source code to find potential security holes, such as SQL injection locations.
   - These instruments can spot weak coding patterns and offer suggestions for fixing them.

7 Secure Coding Practises and Education:
   - Educating developers about the dangers of SQL injection attacks and promoting secure coding practises among them will assist stop vulnerabilities from ever being introduced.
   - Best practises include utilising prepared statements or parameterized queries, verifying and sanitising user input, and avoiding the development of dynamic queries.

Though most of these have been already been researched on and included as parts of apps and projects, the majority of the research has been done on php based applications since it is the most vulnerable kind of

applications. Also other researches include web frameworks to detect SQLIA, comparative studies in machine learning to compare different testing techniques against a combination of security approaches.

VI. Limitations of Current Methods

Although the current techniques for avoiding SQL injection attacks provide useful security, they have several drawbacks. Understanding these restrictions is crucial for creating more effective security solutions. Here are some typical drawbacks of modern techniques:

1 Incomplete Input Validation:
- Techniques for validating input may not account for all potential input circumstances, enabling possibility for validation tests to be evaded and vulnerabilities to be introduced.
- If validation is insufficient, advanced attack vectors may go undetected or use input that has been encoded or obfuscated to avoid detection.

2 Exposure to Advanced Attacks:
- Skilled attackers might use sophisticated methods to get beyond input validation and evasion procedures in order to take advantage of weaknesses.
- Existing preventative techniques might not be sufficient to stop zero-day attacks, which take use of undiscovered vulnerabilities.

3 False Positives and False Negatives:
- Current methods may produce false positives, indicating valid input as possibly harmful and interfering with regular programme operation.
- When vulnerabilities go unnoticed, SQL injection attacks may take place without being known about or prevented, leading to false negatives.

4 Application-Specific Implementation:
- The accurate and thorough implementation inside each application is crucial to the success of preventative strategies.
- The overall security posture might be weakened by inconsistent application of preventative strategies or configuration errors.

5 Compatibility and Maintenance Issues:
- As online applications develop, it might be difficult to maintain and update preventive methods across different platforms and versions.
- When merging preventative techniques with current codebases or outside libraries, compatibility problems could appear.

6 Performance Overhead:
- Some preventative measures, such thorough input validation or complicated query parameterization, might result in performance overhead, which affects the responsiveness and scalability of the programme.

7 Limited Support for Legacy Systems:
- Legacy systems may have limited support for applying contemporary preventative techniques if they don't have built-in security features or only receive sporadic updates. This makes them more susceptible to SQL injection attacks.

VII. Proposed Combined Method for SQL Injection Prevention

The proposed method is a MYSQL based Django Application called SecureSkyScholars. The reason for choosing a Django application is because it is not very widely been researched on for testing SQLIA attacks for both prevention and detection. An architectural summary of the suggested combined approach to avoid SQL injection is provided below:

1. Client-Side Components:
   - User Interface (UI): Engages users, gathers feedback, and communicates requests to the application server.
   - Client-Side Input Validation: Before transmitting user input to the server, this process first verifies it to see if it satisfies the requirements.
2. Application Server Components:
   - Input Validation Module:
     - This component collects user input from the client-side and thoroughly validates it to find and remove potentially harmful material.
     - In a student add form, user provided data such as first_name, last_name is validated and sanitized using Django's built in form validation
   - Query Escaping and Sanitization Module: This module implements the necessary escaping techniques to handle special characters in user input and sanitises input to prevent potential injection attempts.
   - Django ORM module:
     - This module automatically escapes input and provides a safe pipeline to interact with the database.
3. Web Application Firewall (WAF):
   - – WAF Gateway: Sits between the client and the application server, analyzing incoming web requests and applying

security rules to detect and block SQL injection attempts.

- – WAF Rules Engine: Contains predefined rules and patterns to inspect SQL-related parameters, query structures, and detect suspicious behavior.

4. Backend Components:
   - – Database Management System (DBMS): Stores and manages the application's database.
   - – DBMS Security Measures: The database management system (DBMS) implements built-in security measures to safeguard the database, including access controls, user rights, and encryption.
   - – Logging and Monitoring: Records and tracks activity for analysis and auditing purposes, including SQL queries, user input, and suspicious occurrences.
5. Continuous Monitoring and Security Assessments:
   - – Security Monitoring System: Gathers and examines logs, alarms, and events from multiple components to identify possible SQL injection attempts.
   - – Tools for Security Assessment: Consistently performs vulnerability scans, penetration testing, and code audits to find systemic flaws and vulnerabilities.
6. Developer Education and Best Practices:
   - – Secure Coding Guidelines: Offers developers advice on secure coding procedures and best practises, with a focus on input verification, parameterized queries, and secure development methods.
   - – Training and Awareness Programmes: Promotes a security-focused mentality throughout the development process by educating developers about the dangers and repercussions of SQL Injection Attacks.

The suggested design serves as an example of a multilayered defence against SQL injection attacks. It includes backend security measures, web application firewalls, parameterized queries, query escaping, continuous monitoring, security assessments, and developer training. It also includes client-side input validation and server-side input validation. By putting this architecture in place, businesses can create a strong defence against SQL injection attacks and keep a safe environment for their databases and apps.

A. Input Validation Techniques

A critical step in avoiding SQL injection attacks is input validation. Applications may make sure that only intended and secure data is allowed by verifying user input. Here are a few methods for input validation that are often used:

1. Whitelist Validation:
   - – Whitelist validation consists of specifying a list of acceptable characters, patterns, or forms for each input field.
   - – Only input that is legitimate is allowed to continue once the programme has checked if the user input meets the predefined whitelist.

```
ALLOWED_HOSTS = ['127.0.0.1']
```

Figure 2: Whitelist validation in app

2. Blacklist Validation:
   - – For each input field, a blacklist of forbidden characters, patterns, or formats must be established.
   - – Input that includes banned content is rejected by the programme after it checks if it fits the blacklist that has been previously set.

Figure 3: Blacklist validation in app

3. Regular Expressions:
   - – Regular expressions (regex) offer a strong tool for input validation against certain patterns.
   - – Regex patterns may be created by developers to match and check data according to predefined standards like email addresses, phone numbers, or ZIP codes.

```
TEMPLATE_STRING_RE = re.compile(r"\{[a-zA-Z#]+\}")
GET_ATTR_RE = re.compile(r"(\w+)\[(\d+)\]")
VALID_HOST_LABEL_RE = re.compile(
    r"^(?!-)[a-zA-Z\d-]{1,63}(?<!-)$",
)
```

Figure 4: Regex use in app

4. Maximum and Minimum Length and Size Limitations:

– Limiting the maximum and minimum length or size of input fields helps avoid extremely long or short input that might suggest malicious intent or programming faults.
– Any input that goes over or under the established limitations may be rejected by the application.

```
class AddStudentForm(forms.Form):
    """Module to add student form"""
    first_name=forms.CharField(
        label="First Name",max_length=50,
        widget=forms.TextInput(attrs={"class":"form-control"}))
    last_name=forms.CharField(
        label="Last Name",max_length=50,
        widget=forms.TextInput(attrs={"class":"form-control"}))
    email=forms.EmailField(
        label="Email",
        max_length=50,
        widget=forms.EmailInput(attrs={"class":"form-control","autocomplete":"off"}))
```

Figure 5: Maximum size limitation in form fields in app

5. Input Encoding:
   – Before being processed or stored, user input must be converted into a secure representation.
   – Base64 encoding, URL encoding, and HTML entity encoding are some examples of methods that may be used to protect against potential data injection attempts and maintain the integrity of the data.

```
b64 = base64.b64encode(lhmac.digest()).strip().decode('utf-8')
```

Figure 6: Input encoding in app

6. Server-Side Validation:
   – Because client-side validation can be disregarded or altered by attackers, server-side validation is a crucial complement to client-side validation.
   – Even if the client-side tests are disregarded, performing validation checks on the server side helps guarantee that all input is correctly checked.

```
@csrf_exempt
def email_is_exist(request):
    """View for checking if email exist"""
    email=request.POST.get("email")
    user_obj=CustomisedUser.objects.filter(email=email).exists()
    if user_obj:
        return HttpResponse(True)
    else:
        return HttpResponse(False)
```

Figure 7: Server side encoding in app

7. Error Handling and Error Messages:
   – Effective error handling is essential for giving users secure and enlightening feedback.
   – Error messages should be general in order to prevent disclosing private information that might enable hackers while still offering consumers useful advice.

It's crucial to fully implement input validation strategies, including all user input fields, and to use them consistently across the programme. Validation rules should be regularly updated and maintained to reflect changing requirements or new risks. Moreover, combining input

```
if button_name=="submit":
    try:
        course.save()
        messages.success(request,"Successfully Edited Course")
        return HttpResponseRedirect(reverse("course_manage",kwargs={"course_id":course_id}))
    except KeyError:
        messages.error(request,"Failed to Edit Course")
        return HttpResponseRedirect(reverse("course_edit",kwargs={"course_id":course_id}))
```

Figure 8: Error handling in app

B. Parameterized Queries: Implementation and Benefits

Prepared statements, commonly referred to as parameterized queries, are a frequently used method for thwarting SQL injection attacks. They entail decoupling user input from the SQL query, thereby lowering the possibility of injection problems. An overview of parameterized queries' use and advantages is provided below:

Implementation:

1. Binding Input Values: To ensure that user supplied values are regarded as data and not as a component of the SQL code, the programme ties the user-supplied values to the respective placeholders individually.

2. Execution of the query: The database engine runs the parameterized query with
the bound input values, making sure they are handled securely.

In the given screenshot Django ORM not only binds but also executes SQL query securely. It takes user import of staff_id from kwargs and securely binds it to ModelSubjects and executes it.

```
class EditResultForm(forms.Form):
    """Module to edit student result"""
    def __init__(self, *args, **kwargs):
        self.staff_id=kwargs.pop("staff_id")
        super(EditResultForm,self).__init__(*args,**kwargs)
        subject_list=[]
        try:
            subjects=ModelSubjects.models_object.filter(staff_id=self.staff_id)
```

Figure 9: Parameterised query in app

Benefits:

1. Preventing SQL Injection Attacks: Parameterized queries stop attackers from introducing harmful SQL code by isolating the user input from the SQL query.
2. Increased Security: Parameterized queries guarantee that user input is always handled as data, removing the possibility of unauthorised access and unintentional SQL interpretation.
3. Protection Against SQL Syntax problems: By binding values independently, syntax problems that would arise from user input that contains characters that violate SQL syntax if they were included in the query string are prevented.
4. greater Performance: Parameterized queries can result in greater query optimisation since the database can store and reuse the execution plan for parameterized statements.
5. Database Query Reusability: By keeping SQL functionality and user input separate, queries may be used again and again with varied input values, increasing the modularity and effectiveness of the code.
6. Portable and Database Agnostic: Parameterized queries may be used with a variety of database management systems (DBMS), which increases the portability and flexibility of the application code.

It is essential to deploy parameterized queries consistently across the programme, encompassing all dynamic SQL statements, in order to effectively capitalise on the advantages they offer. There should also be enough error handling and logging in place to catch any possible problems with parameter binding or query execution. The application's defence against SQL injection attacks is greatly strengthened by parameterized queries when used in conjunction with other security measures like input validation and safe coding techniques.

C. Web Application Firewalls: Role and Effectiveness

SQL injection attacks are only one of the many security risks that online applications must be protected from with online Application Firewalls (WAFs). They serve as a protective barrier, watching and filtering incoming traffic, between the web application and the outside network. An overview of the function and efficiency of WAFs is provided below:

Role:

1. Threat Detection: WAFs analyse incoming web requests and responses, inspecting them for suspicious patterns, malicious payloads, and attack signatures.
2. Request Filtering: By preventing potentially harmful traffic before it reaches the application server, they filter incoming requests based on established security rules, policies, and patterns.
3. Attack Mitigation: By detecting attack patterns or anomalies in request structures, WAFs actively identify and block known attack vectors, including attempts at SQL injection.
4. Virtual Patching: Even before the underlying application code is changed, WAFs deploy virtual fixes to web apps that are known to have vulnerabilities. This adds another layer of protection.
5. Logging and audits: WAFs keep thorough records of all incoming requests, thwarted attacks, and other security events, facilitating forensic investigation and compliance audits.

Effectiveness:

1. Proactive Defence: By utilising established security rules and signatures, WAFs offer proactive defence against a variety of known and developing vulnerabilities, including SQL injection attacks.
2. Defence-in-Depth: In order to develop a thorough security strategy, WAFs work in conjunction with other security measures like input validation and secure coding practises by providing an additional layer of protection.
3. Mitigation of Zero-Day Attacks: By examining request payloads and behavioural anomalies, WAFs may identify and stop unknown attack patterns or zero-day flaws, offering an additional layer of security until fixes are released.
4. Rapid Reaction: By upgrading their security rules and signatures, WAFs can respond quickly to new threats, making sure that new attack vectors are quickly recognised and countered.
5. Flexibility and customisation: WAFs frequently enable fine-tuning and specialised protection by allowing customisation of security rules and policies to match certain application requirements.
6. Scalability: WAFs can be installed as software modules, hardware appliances, web-based services, or other configurations, offering scalable security to handle increasing application traffic.

It's crucial to remember that although while WAFs are quite good at stopping many threats, including SQL injection, they shouldn't be viewed as a stand-alone remedy. To stay up with changing threats, routine updates, rule maintenance, and configuration changes are required. Additionally, for efficient incident response and proactive security management, continual monitoring and analysis of WAF logs and warnings are essential. WAFs greatly improve the security posture of online applications and aid in preventing SQL injection attacks when used in conjunction with other security measures and recommended practises.

VIII. TESTING

SQLMap, a widely used open source penetration testing tool has been used to exploit and detect possible SQL Injection vulnerabilities.

Figure 10: SQLMAP testing

In the image above, sqlmap was used to exploit the form in the url- http://127.0.0.1:8000/staff_edit/. As we can see sqlmap map was not able to penetrate into the given form. This shows that the proposed combinational app is able to withstand the sql injection attacks. SQL map uses a wide range of technique to exploit including Boolean based blind, time- based blind, union based and many more. But there is some warnings have also been detected and it detects HTTP error codes, including 500 internal server error, during the scan.

IX. Conclusion

As a result of this thesis's integrated approach, SQL Injection threats are now prevented. Attacks using SQL Injection pose a serious risk to the security and integrity of web based systems, which was the research's main focus. The suggested strategy offers a thorough defence against SQL Injection attacks by combining a number of techniques, such as input validation, parameterized queries, query escaping, web application firewalls, and continuous monitoring.

The combination method's efficacy has been shown through experimental assessments. The findings showed a high attack detection rate, successfully identifying and preventing attempts at SQL Injection. A substantial decrease in false positives and false negatives was also demonstrated by the approach, guaranteeing accurate detection without needlessly interfering with real user input.

Performance analysis of the combined technique revealed effective resource utilisation and scalability, making it appropriate for cloud deployment. With support for interoperability with numerous programming languages, frameworks, and database management systems, the integration procedure proved to be rather painless.

The contributions of this thesis have applicability to how cloud-based applications are secured. Organisations may strengthen their defences against SQL Injection attacks, secure sensitive data, and maintain the integrity of their cloud architecture by putting the suggested combination technique into practise. The study's findings advance knowledge and understanding of combating SQL Injection attacks in cloud systems, which advances the area of cloud security.

Although the suggested approach shows promising results, there are certain drawbacks that demand more research. In order to continually improve the security measures against SQL Injection attacks in the ever-evolving cloud ecosystem, further study might investigate sophisticated attack detection algorithms, adapt the strategy for various cloud platforms, and take upcoming risks into consideration.

Overall, this thesis demonstrates the significance of preventative security measures in cloud settings and provides an important way to reduce the risks associated with SQL Injection attacks. The combined strategy that has been offered adds to the body of knowledge in cloud security and offers organisations a useful tool for enhancing their security posture and guaranteeing the security of their cloud-based applications.

Acknowledgment

References

[1] Wei, K., Muthuprasanna, M., & Kothari, S. (2006). Preventing SQL injection attacks in stored procedures. In *Australian Software Engineering Conference (ASWEC'06)* (pp. 8 pp.-198). Sydney, NSW, Australia. doi: [10.1109/ASWEC.2006.40](https://doi.org/10.1109/ASWEC.2006.40).

[2] Thomas, S., & Williams, L. (2007). Using Automated Fix Generation to Secure SQL Statements. In *Third International Workshop on Software Engineering for Secure Systems (SESS'07: ICSE Workshops 2007)* (pp. 9-9). Minneapolis, MN, USA. doi: [10.1109/SESS.2007.12](https://doi.org/10.1109/SESS.2007.12).

[3] Merlo, E., Letarte, D., & Antoniol, G. (2007). Automated Protection of PHP Applications Against SQL-injection Attacks. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)* (pp. 191-202). Amsterdam, Netherlands. doi: [10.1109/CSMR.2007.16](https://doi.org/10.1109/CSMR.2007.16).

[4] Ping, C., Jinshuang, W., Lanjuan, Y., & Lin, P. (2020). SQL Injection Teaching Based on SQLi-labs. In *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)* (pp. 191-195). Dalian, China. doi: [10.1109/ICISCAE51034.2020.9236904](https://doi.org/10.1109/ICISCAE51034.2020.9236904).

[5] Gu, H., et al. (2020). DIAVA: A Traffic-Based Framework for Detection of SQL Injection Attacks and Vulnerability Analysis of Leaked Data. *IEEE Transactions on Reliability*, 69(1), 188-202. doi: [10.1109/TR.2019.2925415](https://doi.org/10.1109/TR.2019.2925415).