Network Security Final Project Report

Shuo Li [sli53@jhu.edu](mailto:sli53@jhu.edu)

Yuge Gong [ygong@jhu.edu](mailto:ygong@jhu.edu)

# Abstract

DDoS attacks and SQL Injection attacks are two of the most famous attacks. Both of them have variant revisions and multi detection and defense methods have been published. But how to build a system perfectly detects and defenses these attacks are still a big issue for all computer scientists. In this project, a system detects and defenses these two kinds of attacks are built. Our hypnosis is that, the analysis of system information, traffic information and the IP accessing log would detect and defense major DDoS attacks. Meanwhile, storage procedure strategy could be used to prevent SQL Injection. To test our system, we also design two different experiments with practical attack tools. The experimental results supported my hypothesis by showing that attacks are detected and defensed.

# Introduction and Purpose

This project builds a system detects and defenses two kind of attacks: DDoS attack and SQL Injection. The detection of DDoS attack is based on the analysis of system information, traffic information and the IP accessing log. Meanwhile, this system provides a prepared statements (parameterized query) and storage procedure strategy to prevent SQL Injection.

Our group designs a web server architecture based on J2EE framework to simulate the security system. Servlet filters and Runtime Thread are used to support the functional need of IP matching and system monitoring. At the same time, log file are well defined to visualize the traffic condition. JavaScript is included to display the chart of system information and traffic condition

In section 2, we redefine part of hypnosis we mentioned in the project proposal. The purpose of this system keeps unchanged, which detects and defenses the DDoS and SQL Injection attack.

In section 3, we discuss the methods we use to verify our hypothesis order to verify our assumption, we simulate our attacks in two different ways. The first is a Perl script generates packets with faked variant IP address and float packet size. Without cloud environment, this simulation approximate DDoS attack in very high level. The second attack is user input with SQL phases. If the system is Inviolability, it would prevent the user illegally login. In order to examine the result of second attack, we set the experiment group with input with SQL query, while control group not.

In section 4, experiment data and result are given to examine system's robustness. In DDoS detection and defense, we design the experiment with experiment group as the system under attack, while control group run in normal situation. In order to examine the result of SQL Injection attack, we set the experiment group with input with SQL query, while control group not.

In section 5, we give out our conclusion based on our analysis. The system is strong towards these two kinds of attack. Also the shortcut of the system has been discussed.

# Hypothesis and Question

## DDoS Attack

Before explaining the question of DDoS, we describe the question of Dos. A Dos attack may be directed to a specific computer operating system, to a specific port or service on a target system. DoS attack will consume critical resources in a network service. Generally, DoS has two feature components.

1. Attacker can send a variety of packets. The attack traffic can be made arbitrarily similar to legitimate traffic.
2. The volume of traffic must be large enough. Usually, there should be more than one computer, but it can also happen under a single IP address.

DDoS attack is a DoS attack that occurs from more than one source and from more than one location at the same time. The number of coordinated sources in a DDoS attack can vary from dozens to hundreds. The DDoS has two more features making it perform better than simple DoS attack.

1. We can maximize the bandwidth usage.
2. The attack packets come from different sources, which makes it hard to be traced.

By demonstrating the features of DoS and DDos, we can find that tracing attack source(s) efficiently would be a most important method to detect DoS and DDoS.

1. By using an IP log digging to record the IP address and its relevant visit times can help us to judge whether it is a normal visit or an abnormal one.
2. As DoS and DDos would consume critical resources of a network service, keeping monitoring the resources occupation rate can help us to detect the attack in a more efficient way. More specifically, we would monitor the system and network workload.
3. The detection of a DoS and DDoS should be based on the situation without physical layer damages and routing attacks, in which most of attacks should be detected efficiently.

## SQL Injection

A [SQL injection](https://www.owasp.org/index.php/SQL_injection) attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. Thus we can see that the main idea of SQL injection is to modify the SQL query. By doing so, an attacker can then log in one’s account without authorization or modify the data of the database of the application.

Usually, attackers can use evil character like quote (’) to check whether they can do SQL injection on the system. For example, the internal SQL query of the system is “SELECT \* FROM User WHERE username = ‘” + username + “’;” and an attacker’s input is “evil’s”. This would generate a SQL query SELECT \* FROM User WHERE username = ‘evil’s’ which will make a syntax error happen for executing query on database. After the attacker finds the security bug of the system, he can utilizes it to do further SQL injections to modify the system or some thing else maliciously.

Using a blacklist or a whitelist to check the user inputs can help us to detect the evil inputs from attackers. The blacklist contains the character that cannot be use as input while the whitelist contains acceptable input. More specifically, we need to standardize the every input from users. Regular expression will be used in our project to implement this procedure.

Noticing the example mentioned above, letting the user input deal with the incomplete SQL query directly is an other important risk of SQL injection, because this give the attacker opportunity to modify the query without limitation. In order to exclude this situation for our system, further methods we used.

1. Prepared statements can pre-define the SQL query, which reduce the opportunity of being modified by attackers.
2. Storage procedures can provide system using parameters instead of query sentences to do database operations, which avoids the situation of user dealing with query sentence directly.

# Method

The following figure shows how our system works. There are two main functions for the system, DDoS detection and defense and SQL injection detection and defense. In DDoS detection, the system will both monitor the IP address of website visitors and performances of system itself. If one of them reaches a standard limitation we pre-defined already, an DDoS attack will be detected and the server will refuse the IP to visit the website any more. In SQL injection, the system will determine whether the user inputs are legal or not. If any illegal inputs are detected, the visitor can not continuing his website visit any more.

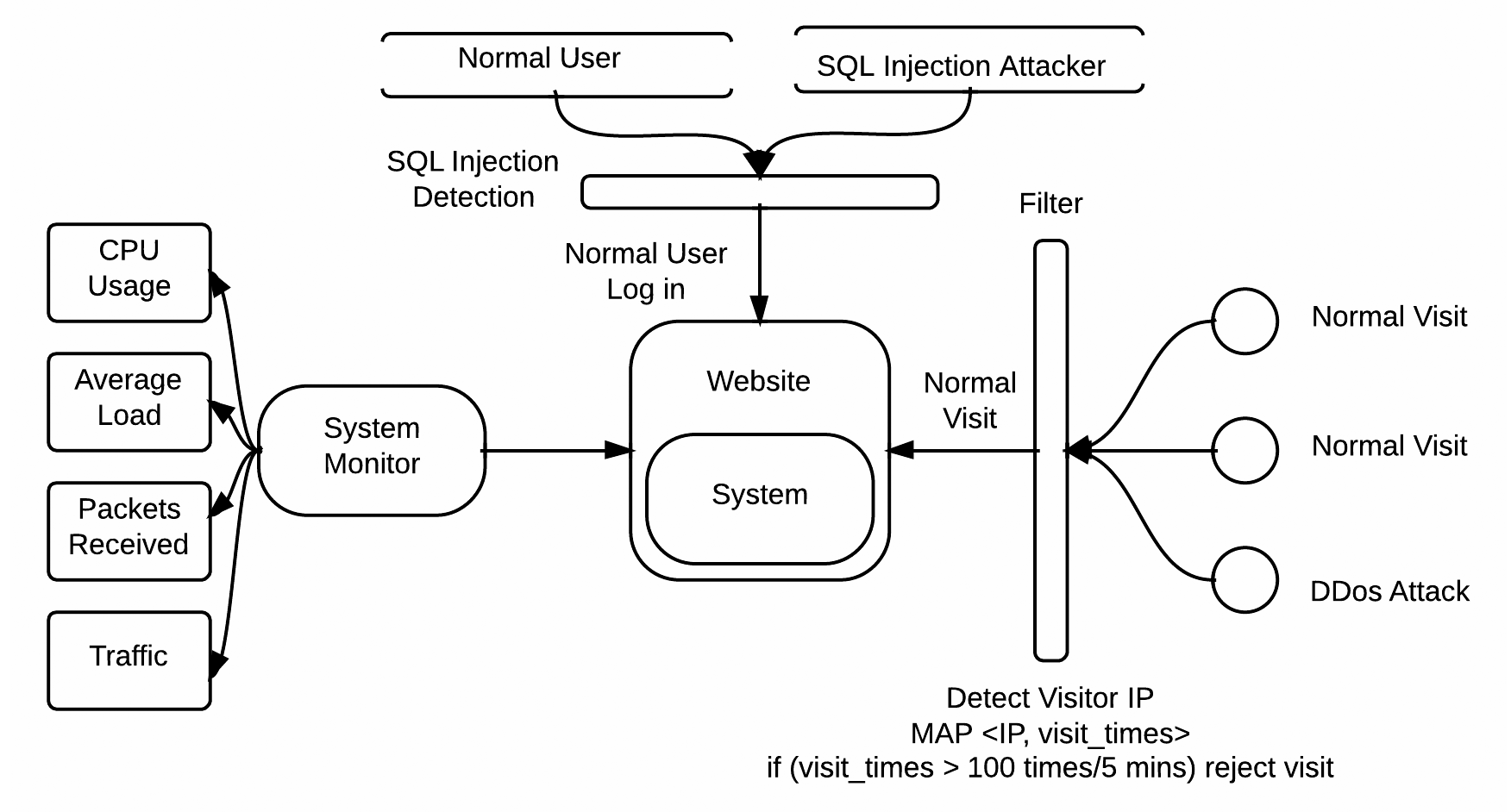


Figure 1 System flow char

## DDoS Detection

In order to detect a DDoS attack, we apply two strategies on this system.

Strategy 1: Filters to count the IP visiting records. A map structure is used to save the records. The map holds the IP visiting records for past 30 minutes. If one single IP address requests the response from server for more than a certain threshold, then block the IP and report as a DoS attack.

Strategy 2: A Java Runtime thread is used to monitor the system performance information, included CPU usage, average system load, packets income for last 30 seconds, and traffic income for last 30 seconds. The significant unusual waste of system resource will raise the alert to announce the administrater there may be a DDoS attack.

According to the definition of DDoS attack, we could prove our system is immune to DDoS attack if it’s sensitive to DoS attack, except the zombie network. Given the common attack methods of DoS attack. We ensure the system is immune to DoS by Filter method, detect the DDoS by the report of unusual system and network resource waste.

For DoS attacks, we defense them by prove our system has ability to defense common DoS attacks:

1. Consumption of computational resources, such as bandwidth, disk space, or processor time.
2. Disruption of configuration information, such as routing information.
3. Disruption of state information, such as unsolicited resetting of TCP sessions.
4. Disruption of physical network components.
5. Obstructing the communication media between the intended users and the victim so that they can no longer communicate adequately.

For attack 1, 4 and 5. if the communication media is broken on the physical layer, there is no way for us to get rid of it. Otherwise if the concern of physical layer damage has been ignored, the monitoring thread will detect this unusual behavior of server, then alert this to be a DoS attack. For a DoS attack, the IP filter strategy would blcok the IP address directly. For an faked UP attack, even there is no method to distinguish the attckers or zombies in the network, other actions such as notification to users could reduce the damages.

For attack 2 and 3. This kind of attacks could be applied by faking the legal routing information, then the expedition of the routing table would cause significant wast of bandwidth. Or lead the stream to other nodes in the network then exhaust the server's resources. These attacks are started from the physic level, our system detect this attack by monitoring the traffic average situation. If the traffic and packets drop to low level in daily working time, there could be possibility such attacks happens. System will alert the unusual status to the administrator.

However, without rigorously proof, the conclusion that routing attack is the only reason causes the unusual decrease of traffic is not stable. The random user behavior and lack of routing information lead to a result, we can only report this exception rather than taking some real actions automatically.

Thus, our system is stable toward a DoS attack. DDoS Defense

## DDoS Defense

## SQL Injection

In SQL injection part, a website written by JSP was build to implement SQL injection. This website is a one with simple functions that normal users can log in this website by using their own user name and password. If the user name and password match with each other, then the web server will redirect the user to a new web page showing that "Log in successfully!", otherwise, the web server will feedback an error message.

Attackers, on the other side, are trying to log in this website by malicious input of user name or password. What we would do is finding out the method to detect attackers' attacks and defend them. In order to show the effects of the detection and defense of SQL injection for this website, we build two login pages for comparison. One is a naive website checking only the length of the input strings, which cannot be zero or larger than 15. The other is equipped with several methods of SQL injection detection and defense.

### SQL Injection Detection

Before we detect SQL injection, we need to understand how a SQL Injection happens. Let's first have a look at how a website without detection works when someone input his user name and password to log in.

In this project, we have used MySQL to build a database for the website and keep a table named “Tbl\_User” for user informations including user name and matching password.

Suppose one user Tom wants to log in the website, he would fill the log in form with his user name “Tom” and the matching password “Tom1234”. After he submits his information, the web server would then begin to parse these informations. Actually, in the JSP page of log in, there are following java sentences to explain this procedure.

1. String user = request.getParameter(“userid”);
2. String pwd = request.getParameter(“pwd”);
3. String query = “SELECT \* FROM Tbl\_User WHERE u\_id = ‘” + user +“’ AND u\_password = ‘” + pwd +“’;”;
4. ResultSet rs = st.executeQuery(query);

The first and second sentences are used to get the user name “Tom” and the password “Tom1234” input from user Tom. The third sentence combines the input strings with the incomplete SQL query. After we get the query, “SELECT \* FROM Tbl\_User WHERE u\_id = 'Tom' AND u\_password = ‘Tom1234’;” (setence 3), we would use it to do searching on database based on the informations provided in above sentences (sentence 4).

Normally, a good person would provide legal information, that is, a username and a password that had already been stored in database before. Then the server would use this information to do SQL query to verify whether these information is matching that in the database. However, a website attacker can utilize this verifying procedure to log in the website without verification. We will explain it then.

The main idea of SQL injection is to modify the SQL query of web server want to deal with. As the website without detection will not check the user inputs, any input will be accepted by server as long as the length of the input satisfying the requirements. Take a malicious input of username as a SQL injection example. An attacker cannot just input ``Alice'' as his user name because the database does not store such a user name or the attacker cannot provide a matching password for the existing user name “Alice”. To continuing his attacking, he can input “1’ OR ‘1’ = ‘1” instead of “Alice” and password. This is because the server is now doing the SQL query “SELECT \* FROM Tbl\_User WHERE u\_id = ‘1’ OR ‘1’ = ‘1’ AND u\_password = ‘1’ OR ‘1’ = ‘1’ ”. As ‘1’ = ‘1’ will be true under any condition, the attacker can always log in the website without verification.

After knowing how an attacker user SQL injection to attack the website, we can then develop the method to detect the situation. In our case, we use regular expression ([^0-9a-zA-Z]) to check user's input. Relevant java sentences are shown as follows.

1. String regEx = “[^0-9a-zA-Z]”;
2. Pattern pattern = Pattern.compile(regEx);
3. Matcher userMatcher = pattern.matcher(user);

Matcher pwdMatcher = pattern.matcher(pwd);"

1. if (userMatcher.find() $||$ pwdMatcher.find())

out.print(“<script> alert(‘illegal characters!’);

window.location=‘Login.jsp’;</script>”);

If a user name or password contains any character that is not a letter or figure, then the server would determine that the input contains illegal character(s) and then redirect the user to the original login page. Thus, the username or password is invalid and cannot be used to log in the website.

### SQL Injection Defense

Using regular expression can standardize the input from users, which means implementing a detection of SQL injection can filter the malicious input from attackers to a large extent. Detection, however, cannot promise server a prevention of a SQL injection. A more efficient and safer way to defense attacker from SQL injection is being used here.

Instead of using simple SQL query to do search on database, we prefer prepared statements and storage procedure here. By implementing prepared statements, we can first define our SQL query, which make attacker cannot modify our SQL query later. By implementing a storage procedure, server will not just patch the incomplete SQL query sentence with input strings from users to get a completed one. On the other hand, we will regard these input strings as input parameters of a single storage procedure stored in database before.

1. String user = request.getParameter(“userid”);
2. String pwd = request.getParameter(“pwd”);
3. CallableStatement cs = con.prepareCall(“{CALL verifyUser(?,?)}”);
4. cs.setString(“user”, user); cs.setString(“pwd”, pwd);
5. ResultSet rs = cs.executeQuery();

In the first two sentences, the web server can get the input strings of user name and password as before, but then we will not use the simple query like “ SELECT... ” sentence any more. Instead, we would first define our parameterized query (sentence 3), then we set parameters for this prepared statements (sentence 4). After this, we call the storage procedure “verifyUser”(sentence 5). By doing so, we can avoid using the input strings from users directly, which can reduce the possibility of SQL injection on server.

# Data and Results

## DDoS attack

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Cpu\_usage** | **Avg\_load** | **Packets** | **Traffic** |  | **Cpu\_usage** | **Avg\_load** | **Packets** | **Traffic** |
| 3.12% | 0.62% | 194 | 0 MB |  | 3.12% | 0.69% | 163 | 0 MB |
| 3.33% | 0.66% | 270 | 0 MB |  | 3.12% | 0.71% | 280 | 0 MB |
| 5.88% | 0.71% | 266 | 0 MB |  | 6.25% | 0.75% | 165 | 0 MB |
| 6.45% | 0.67% | 286 | 5 MB |  | 3.22% | 0.80% | 7632 | 5 MB |
| 6.25% | 0.78% | 274 | 0 MB |  | 10.25% | 0.74% | 315 | 0 MB |
| 8.82% | 0.775 | 267 | 0 MB |  | 8.33% | 0.78% | 177 | 0 MB |
| 6.45% | 0.715 | 354 | 0 MB |  | 6.605 | 0.87% | 8809 | 7 MB |
| 30.0% | 0.90% | 358 | 0 MB |  | 9.90% | 0.81% | 8933 | 6 MB |
| 14.7% | 0.73% | 291 | 0 MB |  | 8.33% | 0.90% | 8878 | 6 MB |
| 6.45% | 0.85% | 306 | 0 MB |  | 13.15% | 0.91% | 16250 | 12 MB |
| 6.60% | 0.88% | 364 | 0 MB |  | 12.50% | 0.99% | 16388 | 12 MB |
| 9.90% | 0.84% | 270 | 0 MB |  | 11.76% | 0.99% | 16810 | 12 MB |
| 8.82% | 0.82% | 345 | 0 MB |  | 10.255 | 0.91% | 17738 | 13 MB |
| 9.37% | 0.98% | 494 | 0 MB |  | 9.37% | 1.00% | 17859 | 13 MB |
| 9.37% | 0.78% | 378 | 0 MB |  | 10.81% | 1.99% | 16841 | 13 MB |
| 75.0% | 1.07% | 637 | 1 MB |  | 70.315 | 1.00% | 13009 | 9 MB |
| 30.0% | 1.00% | 459 | 0 MB |  | 9.90% | 0.99% | 10418 | 8 MB |
| 30.95% | 1.00% | 359 | 0 MB |  | 13.51% | 1.15% | 11876 | 9 MB |
| 23.7% | 0.73% | 315 | 0 MB |  | 10.81% | 1.14% | 9329 | 7 MB |

Table 1 Website without DDoS attack Table 2 Website with DDoS attack

## SQL Injection

The following tables shows the differences between website reaction with or without SQL injection detection and defense.

|  |  |  |
| --- | --- | --- |
| **Reaction** | **Input Password** | **Reaction** |
| null | null | Username cannot be empty! |
| Alice | null | Password cannot be empty!  Wrong password! |
| Yuge | admin | Log in successfully! |
| 1' OR '1' = '1 | Any password | Wrong password! |
| 1' OR '1' = '1 | 1' OR '1' = '1 | Log in successfully! |

Table 3 Website without SQL injection detection

|  |  |  |
| --- | --- | --- |
| **Reaction** | **Input Password** | **Reaction** |
| null | null | Username cannot be empty! |
| Alice | null | Password cannot be empty!  Wrong password! |
| Yuge | admin | Log in successfully! |
| 1' OR '1' = '1 | Any password | lllegal characters! |
| 1' OR '1' = '1 | 1' OR '1' = '1 | lllegal characters! |

Table 4 Website with SQL injection detection

# Conclusion

At the ideal scene, the system is immunity towards these two kinds attacks. But in practical, it may face such a challenge. If the attacker use the zombie network to fork the attack program to many users, at the same time set the packet to be significant large, our system would report that it is under DDoS attack. But there is no automatic way to solve such a situation because it is almost impossible to filter the malicious attacks from regular requests at such users number.

## DDoS attack

### DDoS detection

### DDoS defense

## SQL injection

### SQL injection detection

### SQL injection defense

# Reference