DIGITAL COMMONS
@ UNIVERSITY OF SOUTH FLORIDA

University of South Florida
Digital Commons @ University of South Florida

USF Tampa Graduate Theses and Dissertations
USF Graduate Theses and Dissertations

10-27-2015

BlindCanSeeQL: Improved Blind SQL Injection For DB Schema Discovery Using A Predictive Dictionary From Web Scraped Word Based Lists

Ryan Wheeler
University of South Florida, ryanwheeler@mail.usf.edu

Follow this and additional works at: https://digitalcommons.usf.edu/etd

Part of the Computer Sciences Commons

Scholar Commons Citation
Wheeler, Ryan, "BlindCanSeeQL: Improved Blind SQL Injection For DB Schema Discovery Using A Predictive Dictionary From Web Scraped Word Based Lists" (2015). *USF Tampa Graduate Theses and Dissertations.*
https://digitalcommons.usf.edu/etd/6050

This Thesis is brought to you for free and open access by the USF Graduate Theses and Dissertations at Digital Commons @ University of South Florida. It has been accepted for inclusion in USF Tampa Graduate Theses and Dissertations by an authorized administrator of Digital Commons @ University of South Florida. For more information, please contact digitalcommons@usf.edu.

BlindCanSeeQL: Improved Blind SQL Injection

For DB Schema Discovery Using A Predictive Dictionary

From Web Scraped Word Based Lists

by

Ryan Wheeler

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Yicheng Tu, Ph.D.
Jay Ligatti, Ph.D.
Yao Liu, Ph.D.

Date of Approval:
October 6, 2015

Keywords: Security, Web Attacks,
Microsoft SQL Server, Inferential, Bisection

Copyright © 2015, Ryan Wheeler

DEDICATION

I would like to dedicate this thesis to John Draper (aka Captain Crunch). His curiosity in seeing how technology worked was an inspiration to me. As a kid growing up in the 80’s, I was lucky to have access to personal computers, starting with a C64 and a 300-baud modem. I was incredibly curious about computer and phone systems and how they worked, and therefore John Draper was one of my early heroes.

> “I do it for one reason and one reason only. I'm learning about a system. The phone company is a System. A computer is a System, do you understand? If I do what I do, it is only to explore a system. Computers, systems, that's my bag. The phone company is nothing but a computer.” [1]
>
> *- Secrets of the Little Blue Box, Esquire Magazine, October 1971*

During my early twenties I was lucky enough to meet him while he lived in Florida, and have kept touch with him ever since. I see him every year at the DefCon Security Conference in Vegas, and the love and passion for how things work burst forth from him in every conversation. John Draper is still very active in the tech community, attending conferences and inspiring young people to look both inside and outside the box when it comes to technology.

Thanks John – 2600 Hz forever.

ACKNOWLEDGMENTS

Every professor I've worked with while attending USF has been an inspiration, not only for the course material they have taught but with important real life scenarios as well. Thank you all so much!

I would like to acknowledge Dr. Yicheng Tu for all of his support during my time at USF. He has not only been an incredible teacher, but also been a friend and colleague. He has encouraged me to work on various projects, attend database seminars, and help others with database and coding.

Along with Dr. Tu, I would like to recognize Dr. Rollins Turner for pushing me forward with desktop and web application coding. He actively listened to my security concerns for things like Connection String Injection, and quickly added it to his curriculum, teaching classes that they need to sanitize all inputs, not just database related inputs.

I would also like to acknowledge the speakers and special interest groups at the yearly security conferences in Las Vegas: Black Hat and DefCon. Thank you all for continuing to bring security issues to the forefront and inspiring me for this thesis topic. I also want to acknowledge James Gibson at UF for assisting with any math questions I had while working on my thesis.

Lastly, I want to give thanks to my parents. My dad has always been a technology and gadget fan, which gave me early access to computers. My mom is an educator and fueled my drive for always striving to learn more. Thank You!

TABLE OF CONTENTS

LIST OF TABLES ii

LIST OF FIGURES iii

ABSTRACT iv

CHAPTER 1: INTRODUCTION 1
1.1 SQL Injection Overview 1
1.2 Classification and Techniques for SQL Injection 2
1.3 Summary of Discussion and Contribution 5

CHAPTER 2: RELATED WORK 8
2.1 SQL Injection Attacks (SQLIAs) 9
2.2 Code-Injection Attacks on Outputs (CIAOs) 10
2.3 Existing Automated Tools for SQL Injection 11
2.4 Related Work Summary 12

CHAPTER 3: BLIND SQL INJECTION 13
3.1 Anatomy of the Blind SQL Injection Technique 13
3.2 Using Bisection 14

CHAPTER 4: BLINDCANSEEQL – BCSQL – BLIND INJECTION TOOL 17
4.1 Web Crawler 17
4.2 BCSQL DB Class Design 18
4.3 BCSQL Differences and Improvements 20
4.3.1 BCSQL Differences Overview 20
4.3.2 BCSQL Improvements In Detail 21
4.4 BCSQL Examples 24

CHAPTER 5: SUMMARY AND CONCLUSION 28
5.1 Prevention 28
5.2 Improvements 31
5.3 Conclusion 32

REFERENCES 33

APPENDICES 36
Appendix A Copyright Permissions 37

ABOUT THE AUTHOR ..... END PAGE

LIST OF TABLES

Table 1 Required Database Functions 16

Table 2 BCSQL Statistical Methods By Object 19

Table 3 Comparison of Requests for Objects Between Tools.............................27

LIST OF FIGURES

Figure 1 Visual Error Based Attack 2

Figure 2 Example of True and False Blind SQL Injection 14

Figure 3 Sample of BCSQL Running - Console Output 24

Figure 4 Database Statistics At End of Run...25

Figure 5 Savings Using Auto Complete 26

Figure 6 Obfuscation Through Unicode...31

ABSTRACT

SQL Injections are still a prominent threat on the web. Using a custom built tool, BlindCanSeeQL (BCSQL), we will explore how to automate Blind SQL attacks to discover database schema using fewer requests than the standard methods, thus helping avoid detection from overloading a server with hits. This tool uses a web crawler to discover keywords that assist with autocompleting schema object names, along with improvements in ASCII bisection to lower the number of requests sent to the server. Along with this tool, we will discuss ways to prevent and protect against such attacks.

CHAPTER 1: INTRODUCTION

1.1 SQL Injection Overview

SQL Injections are not a new attack vector for discovering data, surpassing authentication information, or even gaining control of a system, yet they are still a prominent means to compromise a system and the data within. [2-4] SQL Injections refer to the insertion or injection of extra information of a SQL query, typically from input data from a client to the server into a poorly validated application, which will execute the code. Most commonly this means a web page that dynamically displays content or updates the SQL database in some way, but could also be used to refer any client/server type application. Since the most common SQL Injection attacks occur through the web, it will be the focus of this paper.

A typical attack through a web application is often done in the query string. However, any type of input may be used to compromise the system, such as header data, POST data, cookies, referrer URLs, user-agent, and more, if the system uses any of these to read or write to the SQL database. By appending data to the end of one of these input vectors, one may see a change in the page's behavior. For instance, if the database powering a website is Microsoft's SQL Server, appending a single quote to the end of the website's URL and query string could cause an error message to appear.

As an example, a website that displays product data might pull up a product page for product number seven using the "id" parameter in the following query string: http://www.somesite.com/target/target.asp?id=7. If the data is not validated and errors are visible, appending a single quote causes the message in Figure 1 to appear: http://www.somesite.com/target/target.asp?id=7' This is a classic example of SQL Injection in web applications. [5]

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before the
character string ''.
/target/target.asp, line 113
```

Figure 1: Visible Error Based Attack. This figure shows a visible error message that would result from using an unclosed quotation.

1.2 Classifications and Techniques for SQL Injection

SQL Injections come in many variations and flavors. According to Open Web Application Security Project (further referenced as OWASP), there are three classes of SQL Injection Attacks: [5]

1. In-band: data is visible in the same channel used for the injection.
2. Out-of-band: data is retrieved outside of the band used for the injection.
3. Inferential or Blind: no data is visible or transferred. However, the attacker is able to determine the data by observing behavior changes.

For obvious reasons, In-band attacks make it easy to reconstruct how the query within the code works (for instance, many Cold Fusion sites show the entire query). If no error message is shown, it is still possible for an attacker to reverse engineer the logic behind the scenes, and many automated tools exist to exploit this.

Within these three classes, five common techniques are used to glean information from the database [5]:

1. Union: appending data using the UNION operator.
2. Boolean: determining data through true/false conditions.
3. Error: visible error data, which can include query logic, data, and more.
4. Out-of-band: uses a different channel to display information.
5. Time Delay: cause the server to delay or sleep in a conditional query (most useful when none of the other methods provide results).

Attacks can be combinations of the above techniques; such as using the UNION operator to force Error based data over an Out-of-band connection. I would also like to mention another technique, although not defined by OWASP in the common techniques. Stacked Queries, which are done by putting multiple queries within the injection, allow the attacker to execute multiple queries in one call. An example of a stacked query could be *"(background query)'; INSERT INTO users (username, password, IsAdmin) values ('attacker','password', 1);--'"*. This would hijack the query to insert the user as a valid admin user, provided that the table names and column data types are correctly used. It should be noted that MySQL does not allow more than one query within a call, providing protection against this type of attack (Oracle and MS SQL allow stacked queries).

If a detailed visible error message is shown, it is extremely easy to get the database schema and data. Most sites now protect against visual error based attacks, often with a redirect to a generic error page. There are many SQL injection utilities that already take advantage of this, so Error based attacks will not be covered during this discussion.

One might also be able to have a page display data using a UNION attack. By doing a UNION command, one appends the data to the result set, which then gets displayed on the page. While not as convenient as a direct error message result, this is a very convenient way to get data out of the database, or even other sensitive system data if the database has access to the file system or other systems on the network. Again, there are many tools out there for gathering visible data, and will not be covered in this paper.

Out-of band attacks are both visible and blind, meaning that you cannot see the data In-Band (thus is blind In-band) but sends the data to a remote server (thus is visible In-band). Out-of band uses specific database commands, such as OPENROWSET in MSSQL or UTL_HTTP in Oracle, to send data through another channel. This technique can use any non-hardened vector, such as FTP, DNS, HTTP, or other protocols that may not be blocked. This requires one to setup and have control of another system that the database server can connect to, along with timestamps or other ordering data since the responses can come out of sequence. Since most servers and firewalls do not block DNS requests this becomes a viable option to sending data from the system to an outside source. This type of attack is not discussed in this paper.

Even if a visible data response isn't shown, an attacker can take advantage of the system using Blind SQL Injection Attacks. One of those techniques is a time-delay attack using built in database commands (e.g. sleep) that cause a delay in the response of the page. However, since other criteria may also cause a delay, such as website or network traffic, this should be the last attack vector to use. It is not covered in this discussion either.

The final blind attack technique is the Boolean technique. By watching the behavior of the application, an attacker is able to reconstruct information by sending particular requests and observing the behavior of the application and the database responses. This attack uses Boolean conditions to determine if the data is true or false. Think of it as playing Twenty Questions with the server, typically checking the ASCII byte value of a single character in the position of the output. While tedious, this type of attack is still one of the most common. Developers tend to think that because you don't see an error, the database is secure. This paper will focus on this Boolean style of attack, providing a tool that uses fewer requests to determine DB Schema objects.

1.3 Summary of Discussion and Contribution

The work in this thesis is based on .NET (ASPX) and ASP websites utilizing a Microsoft SQL (MSSQL) backend. Both are readily available tools from Microsoft and represent the largest installed base of URLs. According to WhiteHat Security, ASPX and ASP total 44% of URLs by language. [6]

Java, PHP, and other languages can connect to MS SQL (as well as .NET and ASP sites connecting to other database (DB) servers such as MySQL and Oracle), but common practice is to use the same DB server vendor as the web site vendor. With Microsoft technologies being the most widely used, and holding the most combined SQL Injection vulnerability, the decision was made to utilize these technologies in development and testing of this thesis.

Many tools exist to automate blind SQL injection, including new tools using threading to gather data quickly. All of the tools for blind injections find data by each character. The downside to this volume of queries is that the attack is highly

visible as the number of requests to a website jumps dramatically. An admin or security device might notice a large jump in the number of requests based on the use of such a tool. A small database with seven tables and 32 columns could take several thousands of requests to determine the schema, and many more than that if using threading, because threading requires the length of each database object to be known in advance.

It is common practice for programmers to use the same table and column names from the database in the website forms and fields when developing a web application that connects to a database server. By building a tool that would intelligently discover a dictionary of key words from a website, one could shorten the number of requests to a server by autocompleting database schema objects utilizing these key words. This could help skip detection triggered by an unsual volume of requests by both slowing down how quickly it sends the requests, and at the same time lowering the number of requests required to determine the schema. This is a framework discovery tool; the tool is not efficient for retrieving data contained within the columns, but instead would return the schema, which the user can then look over for important columns. For instance, one would most likely want to know which tables contain username and password information, but would want to ignore tables with tax information or product weights.

This thesis will cover related work and formal definitions in Chapter 2, as well as a list of some of the current automated tools. Chapter 3 will focus on current, in-use blind injection techniques. In Chapter 4, I will discuss the tool I wrote, BlindCanSeeQL, which automatically crawls a website and grabs any input or form field IDs to build a dictionary file. It then uses a previously discovered blind

injection point and special bisection rules along with the predictive dictionary to exploit DB schema discovery at a more efficient hit rate than other current automated tools. The improvements will be discussed and compared against one of the most popular tools, SQLMAP. [7] Chapter 5 will discuss ways to prevent such attacks, and draw final conclusions on the topic.

CHAPTER 2: RELATED WORK

This chapter discusses other research related to this thesis. SQL Injection Attacks, or SQLIAs, have been formally defined in previous work, including classification, detection tools, and prevention. [8-12] While a majority of attacks are on the web, due to the prevalence of websites being database driven applications that anyone with a browser might have access to, it should be noted that this could apply to any database driven application. For instance, there has been analysis of utilizing SQLite attack vectors in Android OS as a possible way to gain control of a system. [13]

Considering the existence of disassemblers and live debuggers, such as IDA Pro, one should not assume that a carefully coded client software is sending validated data to a server side application. [14] It is possible to decompile a desktop client application, or alter outgoing traffic to a server at the network level, and send additional data that would be accepted at the security level of the server application. Therefore, validation should be performed at both the client and the server level. For instance, GreyWolf is a reverse engineering tool for .NET Framework application that can de-obfuscate and edit the common language runtime, adding payloads or editing attributes. [15]

SQLIAs are more specifically defined by Donald Ray and Jay Ligatti as "code-injection attacks on outputs", or CIAOs. [16-17] This definition is well suited for this thesis as the focus is Boolean based blind SQL injections, in which the output of

the program is different based on the data passed to the server. CIAOs will be discussed in section 2.2.

2.1 SQL Injection Attacks (SQLIAs)

The general definition of SQLIAs is when an attacker changes the intended logic of a SQL query by inserting new data, whether that is SQL keywords, data, operators, or non-code injections. [10] This can be a first-order injection, which directly trigger the SQLIA, or second-order injection, in which the data is saved to the database to be triggered at later time. [11] For instance, a website might allow a new user to be created, parsing the input correctly and saving it to the database using an INSERT function. In another part of the application that data might be retrieved to do an UPDATE function that is not validated correctly, causing the second-order injection to occur.

While many SQLIA types are defined [8-12], the one that this paper focuses on is known as Inference, or Blind Injections. As discussed in chapter one, errors may be hidden from the user, but the output of the application changes based on the input. By using true/false questions about the data values, typically character by character, attackers can extract data from the database.

Although some researchers suggest that parameterized statements, least privilege, or stored procedures can defend against these attacks [19-20], one should not rely on these alone. Incorrectly written stored procedures that call dynamically built SQL would still be vulnerable to SQLIAs. [12] Techniques like character encoding might bypass parameterized statements if not validated, and while least privilege might protect against a first-order injection, it may not protect against a second-order injection, if the code that runs against the data written by

the least privilege user has different privilege. For instance, a reporting tool might be run as a site administrator, and in that report it might grab and execute a SQLIA unknown to the user running the report.

Many SQLIA detection methods and frameworks have been discussed in other works [18]. SQLIA countermeasures exist, such as AMNESIA [21], CANDID [22], and SQLCHECK [23]. However, none of these are catch-all and should be combined with manual and automated approaches, such as defensive programming, code reviews, and scanning frameworks.

2.2 Code-Injection Attacks on Outputs (CIAOs)

In more recent work, it has been shown that the formal existing definitions of code-injection attacks, of which SQLIA is part of, is lacking. The work by Donald Ray and Jay Ligatti [16-17] distinguishes between "code-injection attacks" (CIAs) and "code-injection attacks on outputs" (CIAOs). CIAs are used to define code injected into memory used by an application, while CIAOs are on outputs. By separating and formalizing the definitions, it allows us to more effectively develop and analyze CIAOs.

That work has been further expanded with the discussion of BroNIEs, or Broken Non-code Insertion or Expansions. [25] Similar to parameterized queries, in which placeholders are manually used for untrusted inputs, BroNIEs looks to automatically fill all untrusted inputs to be confined to non-code. Considering all applications that are vulnerable to CIAOs are also vulnerable to BroNIEs, tools could be developed to help stop attacks without requiring developer compliance such as defensive programming.

2.3 Existing Automated Tools for SQL Injection

Doing SQL Injection testing by hand can take a great deal of work. Luckily, there are quite a few automated tools out there. Not all tools are created equal, and some generate many unnecessary requests. EFYTimes.com lists ten of the most powerful tools as of March 2014 [26]:

- BSQL Hacker (MySQL support is in beta)
- The Mole (Union or Boolean only)
- Pangolin (developed by NOSEC)
- SQLMAP (Python code base, endorsed by OWASP, has the most functions)
- Havij
- Enema SQLi
- SQLNinja (focuses on Microsoft SQL Server)
- Sqlsus (Perl based tool for MySQL)
- Safe3 SQL Injector
- SQL Poizon

There are many more tools than those listed here (Absinthe, SQL Power Injector, and Acunetix to name a few). I chose to test against SQLMAP because it is free and open source, and is known for its full support for all SQL injection techniques on eleven major database management systems [7]. SQLMAP has been used in other research papers [27-28] as the main tool to compare results against.

When using SQLMAP, I forced the technique to be Boolean based, and flushed each session and used fresh queries. By default, SQLMAP stores session data so that it does not have to retrieve already known data. SQLMAP also has some great features such as searching for specific strings within database names,

tables, and columns. For instance, it can identify tables and columns that contain the string "name" or "pass", without having to grab the entire schema.

SQLMAP schema detection sorts by sysuser.name+'.'+sysobjects name for tables, skipping the need to read table owners separately. By sorting this way, it can check the current table owner name and characters of the previous table, starting with the first character of the last known table owner. For the default owner, "dbo.", it will do a test for "d" then "b" then "o" then ".". Suppose you have "dbo.Product" then "dbo.ProductVariant" in sort order. On the second table, it would find the first 12 characters (dbo.Product) using 12 requests, and start testing on the 13th character. This saves quite a few requests. It also sorts columns by name, but doesn't seem to do any checks from the previous column data like it does for tables.

2.4 Related Work Summary

SQLIAs have been generally defined, and further defined into CIAOs for this paper. These definitions help create detection frameworks and countermeasures for SQL injections, with references to existing tools for both attacking and protecting against such attacks.

Ideally, server side code needs both defensive programming and automated testing utilities to recognize attacks. This can be strengthened through the use of frameworks and countermeasures created by the definitions of SQLIAs and CIAOs.

It is important to remember that this applies to all database driven applications, not only web based applications, as the client side may not be as secure as one would believe.

CHAPTER 3: BLIND SQL INJECTION

3.1 Anatomy of the Blind SQL Injection Technique

SQL Injection of any type requires an input of some sort: query string, form field, HTTP header (e.g. cookies), and more. When this is passed to an application that does not sanitize inputs, the attacker can execute unintended commands or access data they normally would not be given access to.

For instance, the backend code for a SQL query might be something like this:

SELECT ProductName, ProductDescription FROM Products WHERE ProdID = 1

In this example, "SELECT ProductName, ProductDescription FROM Products WHERE ProdID =" is the query, and "1" is the data. The web application gets the data from the URL, e.g. http://samplesite.com/info.asp?pid=1, where "http://samplesite.com/info.asp?pid=" is the resource, and "1" is the data. An example of a vulnerable code behind could be:

String query = "SELECT ProductName, ProductDescription FROM Products WHERE ProdID = " + request.getParameter("pid");

By manipulating that data part of the URL we can add on to the query being passed to the database. It is becoming more and more common for developers to hide any errors (e.g. Try{} Catch{} statements, or error page redirects). But what about when a query is valid? This is where Blind SQL Injections come into play.

If we were to change the data part of the URL to "1 and 1=1", which is a true statement, the query is still valid and therefore would return the same page content

as sending "1". Changing the data to "1 and 1=2" makes the query false. If the content of the true and false pages are different, then the attacker is able to do a Blind SQL Injection, as seen in Figure 2.

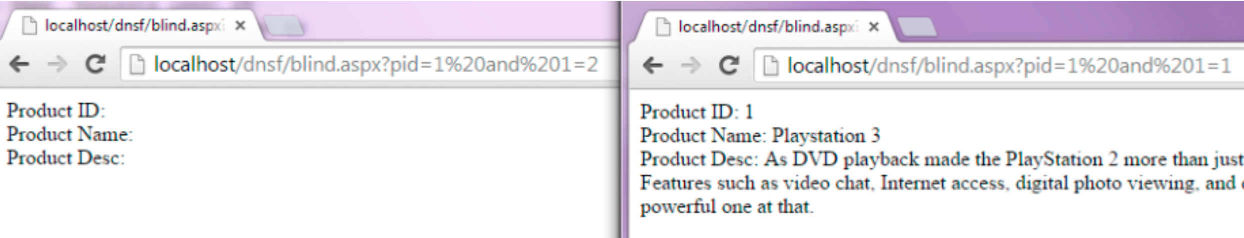


Figure 2: Example of True and False Blind SQL Injection. The query on the left is false ("and 1=2"), while the query on the right is true (" and 1=1"). Since the right query is true, it displays content normally.

Since we can now start asking the server true and false questions, we can find out the values of data, character by character. It would be both time and request intensive to check each position against each possible value, so attackers typically convert the character to its ASCII value and perform bisection queries to reduce the number of requests to find each character. Once the first character is discovered, we move on to the next, until we reach the end.

3.2 Using Bisection

Blind SQL Injection is done one character at a time. In an ideal world, tables and columns would consist only of letters. One would then define the set or range of characters to be [A-Z], or 26 characters, not including lower case. Instead of checking a position in a string against each character (in which case Xylophone would take quite a while if we started at A, then B, etc., for each character), we bisect the list into two lists. We can then ask the server "is the first character in [A-M]"? If the page displays the same content as a false query (e.g. 1=2 did not return the product name of Playstation in Figure 2), then we know the first

character is in the set [N-Z]. We split the set in two each time until we are left with one character, which is the answer. This is bisection of sets, taking a set of size *n* and making two sets, each of size *n/2*.

Therefore the number of steps it takes to find the answer using bisection is $log_2 N$, where *N* is the size of the set. For upper case letters, this would mean we could find the answer within five requests ($log_2 26 = 4.7$, round up to 5). Typically most automated utilities consistently ask "is it greater than" until one character is left in the set.

There are two ways to know when to stop asking for the next character. One way would be to first find the length of the object name, but typically this approach is not used, as it generates an unknown amount of requests, depending on the length of the string. Other tools check for a NULL character instead, so the NULL character would add one more character within a set. If the set A is all upper case alpha characters, adding a NULL character makes the set size 27, $log_2 27 = 4.75$, which is trivial and nearly the same as a set of 26.

Since database objects can contain characters such as numbers, spaces, and other special characters, we use the ASCII character value when bisecting. Since there are 128 characters in the ASCII set, this means the worst-case scenario to find the value is seven requests ($log_2 128 = 7$). Since NULL is included in the ASCII table, it should be found within the seven requests.

In order for character checking, the database must allow for SUBSTRING and ASCII conversion. If determining length, a LENGTH function must exist – this is especially true in multithreaded automated tools to intelligently retrieve data. It is also useful to have conversions functions, such as CAST, which allow you to change

one output type to another. Doing a COUNT(name) for number of tables and using CAST to change it to a string, then finding the numbers by each character, is a lot more efficient than asking the server COUNT(name)=0, then COUNT(name)=1, etc. For instance, a database with 149 tables, "149" can be found in as little as 13 requests by treating it like a string.

Practically every database has some variation of SUBSTRING, ASCII, and LENGTH functions, as shown in Table 1. These functions are critical in every automated tool for SQLIAs.

Table 1: Required Database Functions. Although BlindCanSeeQL works without determining the length of the text, it uses LENGTH to perform sort order of objects.

Function Name	**Function Definition**
SUBSTRING(text, start, length)	This function returns a portion of the "text", with the position starting at the value for "start" and length of "length". Typical use of length is 1, which provides a single character.
ASCII(character)	Returns the ASCII (decimal) value of the given character
LENGTH(text)	Returns the length of the text

CHAPTER 4: BLINDCANSEEQL – BCSQL – BLIND INJECTION TOOL

BlindCanSeeQL (BCSQL) was developed in C# using Visual Studio 2012, .NET Framework 4.5.1. It utilizes the open source Abot C# Web Crawler package [29] available through NuGet, a package manager for Visual Studio. BCSQL specifically targets Boolean based injections for MS SQL Server, although it could be adapted to work with other databases in the future. It features a web crawler that builds a dictionary file out of HTML input tags, which is then used for autocompleting words during blind injection. A non-autocomplete injection option exists as well, to be used as a baseline against autocomplete queries, although it does autocomplete the owner name so that we can compare it SQLMAP's performance.

4.1 Web Crawler

The web crawler functionality of BCSQL has the low level plumbing handled by Abot (multithreading, link parsing, etc.). This includes all open source dependencies such as AutoMapper, CsQuery, HtmlAgilityPack, log4net, Moq, nUnit, and Robots. [29]

Log4Net [30] is used to write out a text log file and the console display. Log4Net allows different levels of logging (Debug, Info, Warn, Error, etc.), which can be set in the *app.config* file for the executable. Debug will show payloads and low-level messages, while Info displays the current table and column being fetched. It is recommended to run at Info level, which shows the levels above as well

(warning and errors). Log4Net was also used throughout the entire project, not just the web crawler.

CsQuery, which is also part of the Abot package, is a complete CSS selector engine, HTML parser, and jQuery port for C# [31]. It is utilized to scrub the page for input tags of type "text", "hidden", and "password". The ID field of the input tag is then passed into a CleanUpInput function, which removes ASP.NET system variables, ClientState variables, ID's that are numbers only, and any starting text declared in the *app.config* file (e.g. "txt,popup" in the *app.config* would change txtFirstName to FirstName, popupImageURL to ImageURL, and so on). The cleaned ID is then put into a SortedSet object, a .NET object which sorts alphabetically and does not allow duplicate elements.

From the sorted set a dictionary file is built, using the path and filename defined in the *app.config*. This file is loaded (or created if it does not exist) every time the program runs, and then saved when the program ends.

4.2 BCSQL DB Class Design

BCSQL is an object oriented console application that utilizes a custom built class called DB, which mimics database structures. The DB class object contains Table objects, which contain Column objects. A tracking object, called BisectionOutput, is used for any requests made, whether it a request for an autocomplete, or the set of requests of the ASCII byte order used to find the current character. This object allows for the history tracking and statistics, including number of requests made, if an autocomplete guess made, and if it correct or not. Each set of objects contains various tracking objects, automatically generated from the BisectionOutput data, as listed in the table 2.

Table 2: BCSQL Statistical Methods By Object. These statistics are generated by parsing the BiscetionOutput objects that are related to main object in the Object column. All statistical objects listed are of type integer.

Object	**Statistical Method Name**	**Description**
DB	TotalRequests	Total requests for schema discovery (does not include initial blind injection test).
	TotalGuessesColumnInDB	Number of all autocomplete attempts made (column names).
	TotalGoodGuessesColumnsInDB	Number of all correct autocompletes (column names).
	TotalGuessesTableNamesInDB	Number of all autocomplete attempts made (table names).
	TotalGoodGuessesTablesNamesInDB	Number of all correct autocompletes (table names).
Table	TotalRequests	Total requests for the table (roll-up of the next three methods).
	–TotalReqForName	Total requests to get the full name (includes owner and table name, e.g. "owner.name").
	–TotalReqForNumberOfColumns	Total requests to find the number of columns in the table.
	–TotalReqForColumns	Total requests to find all column names in the table.
	TotalGuessesForOwner	Total autocomplete requests (owner portion of the full name).
	TotalGoodGuessesForOwner	Total correct autocompletes (owner portion of the full name).
	TotalGuessesForName	Total autocomplete requests (name portion of the full name).
	TotalGoodGuessesForName	Total correct autocompletes (name portion of the full name).
	TotalGuessesForColumns	Total requests for autocompleting the column names in this table.
	TotalGoodGuessesForColumns	Total correct column name autocompletes in this table.
Column	TotalReqForName	Total requests made to get the column name, including any autocomplete attempts.
	–TotalGuesses	Total autocompletes attempted.
	–TotalGoodGuesses	Total correct autocompletes.
Bisection Output	numOfRequests	If it was an autocomplete, returns 1, otherwise returns count of the ASCII Byte Order (list of bytes used to find the current character).

The BisectionOutput (BO) class is built to show the ASCII byte order used to find the current character, or track when an autocomplete attempt is made. The autocomplete attempt is recorded, and if found correctly, populates the *BisectionOutput.Found* string part of the object. This allows us to perform a few methods against the BO objects, such as *DidWeGuess* and *DidWeGuessCorrectly*, which are used in the statistical objects mentioned above.

4.3 BCSQL Differences and Improvements

4.3.1 BCSQL Differences Overview

BCSQL was written for Microsoft SQL Server (MSSQL), referencing the system information tables instead of information schema. Per the Microsoft developer network and MSSQL documentation: "Do not use INFORMATION_SCHEMA views to determine the schema of an object. The only reliable way to find the schema of an object is to query the *sys.objects* catalog view." [32] Therefore, the key objects used in BCSQL are *sys.users* to find the table owner, *sysobjects* for the table name, and *syscolumns* for the column names.

Four differences were implemented in BCSQL for Blind SQL Injections:

1. Sorting by length of the object names, which allows us know the minimum length of the next object without making requests to query the length for each object. This keeps us from autocompleting smaller words once we know the length of last completed object.
2. Tracking current owner and autocompleting it when discovering table objects in the database. This allows us to reset the length and autocomplete word list when the owner changes.

3. Bisection rules check the upper ASCII ranges first, giving priority to the upper and lower case alphabetical ranges; if this check is false, then it does a NULL / end character request. If not in the upper ASCII range or NULL, it checks numbers or underscore range, then for a space. NULL checks for most other bisections go over the control character range from 1 to 31, which is not used for schema objects, requiring other utilities to take seven requests to find NULL while BCSQL takes two.
4. Autocomplete of words based from the dictionary file, which is built from the web crawler or added to by hand. The concept is that web developers tend to use the same ID names as schema object names.

These four concepts are the core to BCSQL usage and how it can provide improvements in the number of requests when compared against other blind injection tools.

4.3.2 BCSQL Improvements In Detail

Table names in a database are unique by owner, and are referenced using "(*sysusers.name)*(dot)*(sysobjects.name)*", such as dbo.Product, guest.Product, etc. While SQLMAP checks the current table against the previous table for the first few characters, BCSQL sorts by length of *sysusers.name* and *sysobjects.name*, in alphabetical order. By doing this we can check if the current owner (*sysusers.name*) is the same, including the dot before the table name. If the last known owner is guest, on the next object we would autocomplete the owner portion using the same owner. This means we have found the first six characters ("guest.") with one request, instead of the six requests done by SQLMAP.

The decision to sort table full names by both length of sysusers.name and sysobjects.name was also made so that we would know the length of the last found table name object. Since it would take several requests to find the length of any object, BCSQL instead tracks the length of the last known *sysobjects.name* table name for the current owner, meaning, we do not keep track of the length of the owner and the dot after the owner's name. Thus we can use this in the autocomplete computation for minimum word completion length. Column names in BCSQL are sorted by the length and then alphabetical order.

Bisection rules are slightly modified for BCSQL. Instead of taking an exact bisection of cutting the sets in half, a few modified rules occur. For alpha characters, the upper range (A-Z) is from (65-90) and the lower alpha range (a-z) is from (96-122). All bisection checks start at greater than ASCII value 64. If it is in the alpha range to start (>64), we check if it is in the lower alpha range (>96). If it is, we bisect into the middle of the lower alpha range (>109, or m). If it is not, we bisect into middle of the upper alpha range (>77, or M).

If it is less than ASCII value 64 to start, we next check for NULL (<1). This makes two checks for NULL, where as most programs would bisect to (<32, <16, <8, <4, <2, <1), which takes seven checks for NULL, or end of object name. If it is not NULL, our next bisection check is in the number range (ASCII range for numbers 0-9 are 48-57), so we check at >47. If it is greater than that, we bisect into >53 (which is 5). If it is not in the number range, we make a special check for the SPACE character (ASCII 32), which is found in five requests. Otherwise the list in bisected normally. Therefore, instead of everything taking seven requests, NULL checks take two requests, spaces take five requests, and some other characters will

take six requests. Some take six requests giving a tiny boost (for instance, the letter 'A' or 'a' would only take six requests).

Lastly, we come to the autocomplete (AC) functions. The AC functions are kicked off once we reach a minimum length, set in the *app.config* settings of the program. They are passed in the string of the word we have so far, e.g. "Pas", the minimum length to guess (determined by the sorting structures mentioned above), and a list of words already guessed (for columns and table names), as well as objects already known (resets for table names). Recall that for columns, names are unique, but for tables, names are only unique by each owner, so we must reset known objects if the owner changes.

BCSQL first finds the current character, and if the character count is greater than the setting to start using AC, it does one test. If the test fails, it will find the next character before testing again. It does the tests in alphabetical order from the dictionary.

For instance, let us assume the following conditions in the program when searching for column names; word found so far: "Pas", length of last word: 7, dictionary words starting with "Pas": Passed, Passport, Password. When hitting the function, the word list would be trimmed to Passport and Password, since Passed is shorter than the current known length. It would test for Passport, and fail, adding Passport to the list of guessed words (parameter 3). After it finds the next character, it would pass in "Pass" to the function, and the only word left would be Password. After autocompleting the object name correctly, it would set the length of last word to 8, add "Password" to the list of known words (parameter 4).

4.4 BCSQL Examples

BCSQL runs on the console with the default in INFO display mode. This will display the progress without crowding up the screen. If detailed payload data is needed, the *app.config* setting can be set to DEBUG. In Figure 3 is an example of starting up the program with autocomplete running. One can see it attempting the autocomplete of "Pro", missing on that attempt, finding the next letter "d", and then on the next autocomplete attempt it finds the correct word.

```
[INFO ] - BlindCanSeeQL started at 2/16/2015 12:55:55 PM
[INFO ] - BaseBlindURL is: http://localhost/dnsf/smallsite/smallblind.aspx
[INFO ] - Query string is: pid=1
[INFO ] - True search string is: Playstation
[INFO ] - Dictionary file is: dict.txt
[INFO ] - Dictionary file # of entries: 110
[INFO ] - Will start autocomplete after X characters: 2
[INFO ] - Checking to see if we have blind injection point...
[INFO ] - Blind injection works!
[INFO ] - Fetching number of tables...
[INFO ] - Found: 7. Byte order: 51,54,55
[INFO ] - Found: 7 (end). Byte order: 51,48,1
[INFO ] - Number of requests for # of Tables: 6
[INFO ] - Table count is: 7
[INFO ] - Starting to find table #1 of 7
[INFO ] - Found: dbo.P. Byte order: 64,96,77,86,81,79,80
[INFO ] - Found: dbo.Pr. Byte order: 64,96,109,118,113,115,114
[INFO ] - Found: dbo.Pro. Byte order: 64,96,109,118,113,111,110
[INFO ] - Trying Autocomplete: dbo.Produce
[INFO ] - Found: dbo.Prod. Byte order: 64,96,109,102,99,100
[INFO ] - Trying Autocomplete: dbo.Product
[INFO ] - The table name is: dbo.Product
[INFO ] - Number of requests: 30
[INFO ] - Starting to find table #2 of 7
```

Figure 3: Sample of BCSQL Running - Console Output. By default BCSQL displays the byte order of the requests as well as attempted autocompletes (ACs).

At the end of running BCSQL, each table is listed with the statistics from that table, including columns found, the total ACs attempted, along with total good ACs. These stats are shown for each column, and then a summary at the end of all the columns. The total requests used for that table are also shown. After all tables are displayed, the entire database statistics are shown, along with correctly completed

words, and the total requests from start until finish (including table and column counts), as can be seen in Figure 4.

```
[INFO ] - ------------- Table Info -------------------------
[INFO ] - -guest.Product  Requests for Table Name: 30
[INFO ] - --# of Columns: 3  Req for # of Columns: 7
[INFO ] - ---Total ACs for owner: 1  Total good ACs: 1
[INFO ] - ---Total ACs for name: 2  Total good ACs: 1
[INFO ] - --Columns--
[INFO ] - ---Name  Requests used: 21
[INFO ] - ----Total ACs: 1  Total good ACs: 1
[INFO ] - ---ProductID  Requests used: 38
[INFO ] - ----Total ACs: 3  Total good ACs: 1
[INFO ] - ---Description  Requests used: 21
[INFO ] - ----Total ACs: 1  Total good ACs: 1
[INFO ] - --Total Req for columns: 80
[INFO ] - --Total Column ACs: 5  Total good column ACs: 3
[INFO ] - -Total Requests for table: 117
[INFO ] - --------------------------------------------------
[INFO ] - --------------------------------------------------
[INFO ] - Total TableName ACs: 6  Total Good TableName ACs: 3
[INFO ] - Total Columns ACs: 19  Total Good Column ACs: 15
[INFO ] - -Correcctly Autocompleted Words:
[INFO ] - --City
[INFO ] - --Customer
[INFO ] - --CustomerID
[INFO ] - --Description
[INFO ] - --Email
[INFO ] - --FirstName
[INFO ] - --LastName
[INFO ] - --Name
[INFO ] - --Password
[INFO ] - --Phone
[INFO ] - --Product
[INFO ] - --ProductID
[INFO ] - --State
[INFO ] - --Zip
[INFO ] - Total Requests: 1480
[INFO ] - --------------------------------------------------
```

Figure 4: Database Statistics At End of Run. This summary lists the requests made for each object, as well as correctly autocompleted words.

Before we compare BCSQL to SQLMAP in a database, here is a quick comparison of using BCSQL against the same database, with a dictionary file of one hundred ten database object names. This dictionary file was built parsing against the sample ASPDOTNETSTOREFRONT (ASPDNSF) website and database. [33] The ASPDNSF database has 149 table objects and 2,040 columns across those tables,

for a total of 2,189 objects in the database. The total length of the objects, including the null character at the end, is 26,782 characters. At seven requests per character, that would be 187,474 requests without any optimizations. The final statistics can be seen in Figure 5, saving us 47,300 requests, which is nearly 27% fewer requests than running without using the BCSQL AutoComplete feature.

```
Total # of Tables: 149            Total # of Columns:   2,040

Length of Objects (including null but not owner names): 26,782
Total Requests for objects at 7 requests / character:  187,474

BCSQL without AutoComplete (only owner name completion)
---------------------------------------------------------------
- Total Requests: 174,872                                     -
---------------------------------------------------------------

BCSQL with AutoComplete and dict.txt of 110 words
---------------------------------------------------------------
- Total Table Name ACs: 51      Total Good Table Name ACs: 9 -
- Total Column Acs:  1,373      Total Good Column ACs: 1,105 -
- Total Requests:  127,572                                    -
---------------------------------------------------------------
```

Figure 5: Savings Using AutoComplete. The total number of requests is significantly lower by utilizing the web scraped list and auto completion.

Even if the attempted ACs did not work, we would have only added 1,424 requests. Considering the savings of checking for NULL in BCSQL is two requests compared against other tools taking seven requests (for a savings of five requests per object), the savings made on the NULL check alone allow for up to five requests per schema before we start to lose any performance.

In Table 3, I make the comparison of SQLMAP, BCSQL without AC, and BCSQL with AC being used against a smaller database, using the same dictionary file in the previous tests. Certain full table name request counts may seem higher than others; this is due to either BCSQL or SQLMAP switching owners, which gets rid of each tools' optimization when finding the new owner in that circumstance.

Table 3: Comparison of Requests for Objects Between Tools. This lists the total requests made to the server to find the database schema's object name.

Owner.TableName	Column Name	SQLMAP	BCSQL no AC	BCSQL w/AC
dbo.Product		61	52	30
	Name	37	29	21
	ProductID	65	64	38
	Description	83	78	21
	(requests total)	**246**	**223**	**110**
dbo.Customer		91	60	24
	Fax	28	22	22
	Zip	28	23	23
	City	35	30	22
	Email	42	36	21
	Phone	42	37	22
	State	42	36	21
	Gender	49	42	42
	Street	49	44	44
	LastName	63	56	21
	Password	63	56	21
	FirstName	70	64	22
	CustomerID	77	72	23
	(requests total)	**679**	**578**	**328**
dbo.Has Space		75	62	62
	has.dot	56	49	49
	has,comma	70	66	66
	(requests total)	**201**	**177**	**177**
guest.Short		49	80	81
	data	27	27	27
	GuestID	59	50	50
	(requests total)	**135**	**157**	**158**
guest.Product		61	52	30
	Name	37	29	21
	ProductID	65	64	38
	Description	83	78	21
	(requests total)	**246**	**223**	**110**
guest.DifferentUser		141	94	94
	test	34	30	30
	customer	62	59	24
	(requests total)	**237**	**183**	**148**
	Total Requests:	**1744**	**1541**	**1031**

CHAPTER 5: SUMMARY AND CONCLUSION

If you know that a company develops similar sites for many clients, you can parse each site in the expectation that they may use similar backend objects. This could provide the tool with a larger and smarter dictionary file. When explaining my tool (and during assistance of proofreading), many programming colleagues have said that it is policy to name objects on the web the same as the database and that they are going to start recommending that their companies do things differently.

5.1 Prevention

The only real protection against SQL Injection attacks is very carefully sanitized input. Any data coming from an untrusted source should be treated with the utmost care, treated as unsafe, and any results being shown should be limited to what the end user needs to see. Any unseen data, such as updates, inserts, deletes, or ability to be sent out-of-band, should be limited and audited. There are a number of software plug-ins and hardware devices out there that try to assist with doing sanitization and limitation, but they are not foolproof, and people may find a work around for tool that you believe is protecting you. Assume that you are not safe and consistently test on a regular schedule against injection attack vectors. The ultimate responsibility comes down to the developer.

Many papers are written for defending against injection attacks [8-13], but here is a summary of defensive coding for developers to use: [34]

- Parameterized Queries
- Stored Procedures
- Least Privilege
- Escaping User Supplied Input
- While List Input Validation

Parameterized queries are written into most languages, allowing you to define the type of input expected. For instance, "test' or 'x'='x" injected into a parameterized query as a user ID would actually look for an entire user ID matching that string. This allows a separation of code and data. These are often used in conjunction with stored procedures. Parameterized queries may not protect against second-order injection attacks as discussed in chapter 2.

Stored procedures are database commands that take in parameters. Thus they are like parameterized queries when it comes to taking in arguments, but the database language interpreter handles the separation of code and data instead of the front-end language interpreter.

It should be noted however that both parameterized queries and stored procedures can be unsafe if the stored procedure itself contains a dynamic query that is not sanitized! Also, most databases do not allow table or column names or other code before the WHERE statement to be parameterized, resulting in the need of a dynamic query. [12]

Least privilege is an excellent idea no matter what type of system you are running. One should never make the user connecting to the DBMS system administrator or root! Creating different users with limited privileges helps protect the system. This doesn't stop the injection from happening, but may lower the

damage than can be caused due to injection. It may not protect from second-order injections as well.

Escaping user-supplied input is a type of input validation, especially used when you need certain control characters such as a single quote. For example, "O'Neil" may be a valid last name but could break a dynamically written query and cause an injection point. Escaping the character uses other control characters, such as a second single quote or a backslash, to wrap the control character within the string. However most languages allow other types of escaping, such as hex encoding or character concatenation, which can bypass escaping input strings. Therefore this is not recommended as the main type of protection, and one should use parameterized queries or stored procedures before using escaping, or at least along with escaping.

While list input validation is where you pass every SQL input through a function that sanitizes the input. Due to the many types of encoding and many variations of user input, it should only be used in conjunction with parameterized queries and stored procedures. It should not be used by itself, unless the input is very limited, such as a white list of only a few options (e.g. red or black are the only options and if any other text is detected, raise an error).

One thing I noticed while working on this thesis was the idea to use Unicode in schema object names. Both SQLMAP and BCSQL did have issues with Unicode characters. However, this would cause a huge problem with developers using the wrong names, not to mention making the code for future developers very difficult to read and program against. It does not protect on tools using object IDs within a

database, such as Absinthe. The only protection it provides is security through obfuscation, making it less human readable, as seen in Figure 6.

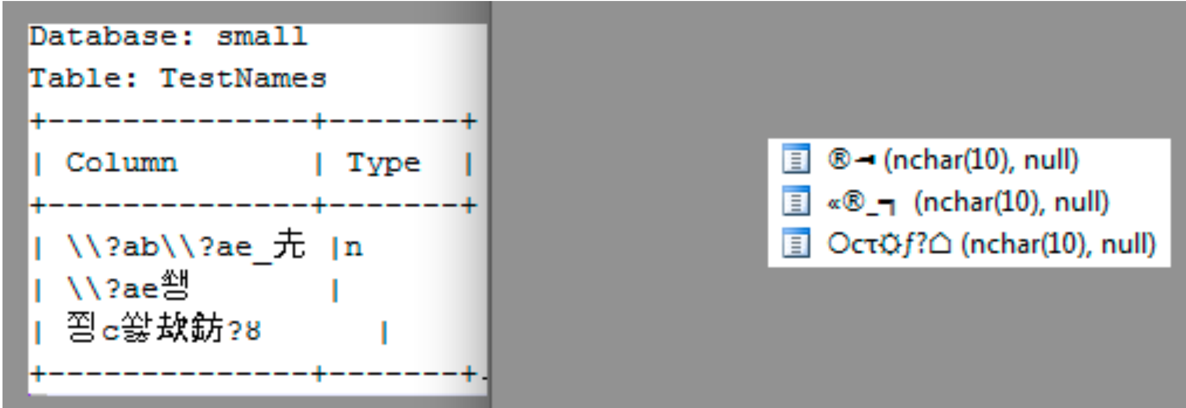


Figure 6: Obfuscation Through Unicode. On the left we have the log results from SQLMAP, and on the right are the column names in SQL Management Studio.

5.2 Improvements

While we did see a strong performance increase using BCSQL, it performs best with the expectation of programmers using the same names in the web pages that are used in the database. Larger dictionary files may help, but are not necessarily better. We should monitor performance so that we are not making more than five auto complete attempts per object (our savings from the NULL check) if we want to stay on target with performance with other tools.

Using Markov Chains may assist on handling the auto completion, especially with developers and databases that use a lot of concatenated words using various types of casing routines (e.g. ShippingMethod, ShippingMethodID, ShippingWeight, etc.). This would also be helpful in the case of partial word completion, such as for an object like ShippingMethodDestination. Knowing the length of the last object and doing a test after getting “Shi” for “Shipping”, then finding the next few letters “Me” and test for “ShippingMethod”, and so on, would be possible using Markov Chains. This would require statistical discovery and analysis against the dictionary

file, requiring the web crawler finding useful words, but would be an interesting approach to finding exceptionally long object names.

Lastly, support for other DBMS systems would be ideal, along with some of the other features found in other tools, like letting the web crawler find the injection point, handle more types of injection, etc. BCSQL was developed as a proof of concept and works with the data at hand and will be open source for anyone to expand upon.

5.3 Conclusion

In conclusion, in practice BCSQL is ideal for websites with a Blind SQL Injection entry point. The worst case may run longer than worst case of other tools if the dictionary files are cluttered with bad autocomplete names, but if that is the case, clear the dictionary file and rerun the web crawler, or run without using a dictionary file. We should still see some savings alone from owner string completion, the improvement for NULL checks, and the bisection preferences. These small improvements give us the opportunity to do auto completion attempts without hurting overall performance, while giving us the opportunity to see a reduced set of requests against the server.

REFERENCES

[1] Secrets of the Little Blue Box, Esquire Magazine, October 1971

[2] The MITRE Corporation: CWE/SANS Top 25 Most Dangerous Software Errors. (2011) http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf

[3] Open Sourced Vulnerability Database: OSVDB: Open Sourced Vulnerability Database. (2014) http://osvdb.org/

[4] The OWASP Foundation: OWASP Top 10 - 2013. (2013) http://owasptop10.googlecode.com/files/OWASPTop10-2013.pdf

[5] Testing for SQL Injection, OWASP.ORG, https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)

[6] 2014 Website Security Statistics Report, WHITEHATSEC.COM, https://www.whitehatsec.com/resource/stats.html (URL includes previous years reports, but requires registration (free))

[7] SQLMAP – Automated SQL Injection Tool, http://sqlmap.org/

[8] William G.J. Halfond and Alessandro Orso, Detection and Prevention of SQL Injection Attacks. http://www-bcf.usc.edu/~halfond/papers/halfond07springer.pdf

[9] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL Injection Attacks and Countermeasures. http://www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf

[10] Atefeh Tajpour , Suhaimi Ibrahim, and Mohammad Sharifi. Web Application Security by SQL Injection Detection Tools. IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 2, No 3, March 2012 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.402.8046&rep=rep1&type=pdf

[11] D. Aich, Secure Query Processing By Blocking SQL Injection. http://ethesis.nitrkl.ac.in/1504/1/thesis_to_upload.pdf

[12] Ke Wei, M. Muthuprasanna, and Suraj Kothari. Preventing SQL Injection Attacks in Stored Procedures. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.9358&rep=rep1&type=pdf

[13] G. J. Smith. Analysis and Prevention of Code-Injection Attacks on Android OS. *ProQuest Dissertations and Theses* pp. 42. 2014. http://search.proquest.com/docview/1639154499?accountid=14745

[14] K. A. Roundy. Hybrid analysis and control of malicious code. *ProQuest Dissertations and Theses* pp. 163. 2012. Available: http://search.proquest.com/docview/1016080667?accountid=14745

[15] Gray Wolf – Reverse Engineering Tool http://www.digitalbodyguard.com/graywolf.html

[16] Donald Ray and Jay Ligatti. Defining Code-injection attacks. POPL'12, January 25–27, 2012, Philadelphia, PA, USA http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.405.8124

[17] D. Ray. Defining and preventing code-injection attacks. *ProQuest Dissertations and Theses* pp. 49. 2013. Available: http://search.proquest.com/docview/1346190700?accountid=14745

[18] Diallo Abdoulaye Kindy and Al-Sakib Khan Pathan. A Survey on SQL Injection: Vulnerabilities, Attacks, and Prevention Techniques. http://irep.iium.edu.my/769/1/ISCE2011_paper323.pdf

[19] San-Tsai Sun, Ting Han Wei, Stephen Liu, Sheung Lau. Classification of SQL Injection Attacks. https://courses.ece.ubc.ca/cpen442/term_project/reports/2007-fall/Classification_of_SQL_Injection_Attacks.pdf

[20] Nikita Patel, Fahim Mohammed, and Santosh Soni. SQL Injection Attacks: Techniques and Protection Mechanisms. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.301.5263&rep=rep1&type=pdf

[21] M. Junjin. An Approach for SQL Injection Vulnerability Detection Proc. of the 6th Int. Conf. on Information Technology: New Generations, Las Vegas, Nevada, pp. 1411-1414, April 2009.

[22] P. 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):1–39, 2010.

[23] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. ACM Symposium on Principles of Programming Languages (POPL'2006), January 2006.

[24] Mei Junjin. An Approach for SQL Injection Vulnerability Detection. Proc. of ITNG '09, pp.1411-1414, 27-29 April 2009.

[25] Donald Ray and Jay Ligatti. Defining Injection Attacks. http://www.cse.usf.edu/~ligatti/papers/bronies.pdf

[26] 10 Powerful SQL Injection Tools That Penetration Testers Can Use, http://www.efytimes.com/e1/fullnews.asp?edid=132535

[27] Angelo Ciampa, Corrado Aaron Visaggio, and Massimiliano Di Penta. A heuristic-based approach for detecting SQL-injection vulnerabilities in Web applications. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.212.2564&rep=rep1&type=pdf

[28] S.Mirdula1 , D.Manivannan. Security Vulnerabilities in Web Application - An Attack Perspective. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.411.9277&rep=rep1&type=pdf

[29] Abot C# Web Crawler, open source C# crawler https://code.google.com/p/abot/, https://www.nuget.org/packages/Abot/, https://code.google.com/p/abot/wiki/v1d5Dependencies

[30] Apache log4net, logging framework for .NET http://logging.apache.org/log4net/

[31] CsQuery – C# jQuery Port for .NET 4, https://github.com/jamietre/CsQuery

[32] Information Schema Views – Tables (Transact-SQL), https://msdn.microsoft.com/en-us/library/ms186224.aspx

[33] ASPDOTNETSTOREFRONT Manual, Demo Data http://manual.aspdotnetstorefront.com/p-1725-demo-data-for-9420.aspx

[34] OWASP SQL Injection Prevention Cheat Sheet https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

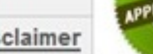
APPENDICES

Appendix A Copyright Permissions

Below is the permission for the use of Figure 1.

Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)

This is a human-readable summary of (and not a substitute for) the **license**.



Disclaimer

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give **appropriate credit**, provide a link to the license, and **indicate if changes were made**. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the **same license** as the original.

No additional restrictions — You may not apply legal terms or **technological measures** that legally restrict others from doing anything the license permits.

Figure 1 is under the Creative Commons Attribution-ShareAlike 3.0 License.

http://creativecommons.org/licenses/by-sa/3.0/

Per this license, credit was given with links to each figure, and no changes were made. Also per the CC BY-SA 3.0, I am providing a link to the owner's license. https://www.owasp.org/index.php/OWASP_Licenses#Licensing_of_OWASP_Website_Content

ABOUT THE AUTHOR

Ryan Wheeler graduated from the University of South Florida (USF) with a Bachelor of Science in Computer Science and a Minor in Mathematics. During his time at USF, he has been an active member of the USF Fencing Team, having served as Treasurer and Vice President over the years, as well as instituting fencing tournaments being held at USF since 2012. He is also a certified US Fencing Association referee, and has earned his C rating in epee and E rating in foil.

In addition to attending USF, Ryan works as an IT Consultant in a variety of forms, including software and database development, project management, and system support. He is an expert in tokenization of credit card payment systems. While working on his Bachelor's and Master's Degrees in Computer Science, he also received his Graduate Business Foundations certificate from USF, and is a certified PMP (Project Management Professional).

Outside of school and work, Ryan is an active member of the Tampa Hackerspace. He enjoys figuring out how things work, as well as teaching others, working on DIY (Do It Yourself) projects, and brewing beer at home. He is a member of the local DefCon813 security group, and would like to present a BCSQL talk at a future DefCon conference in Las Vegas, NV.