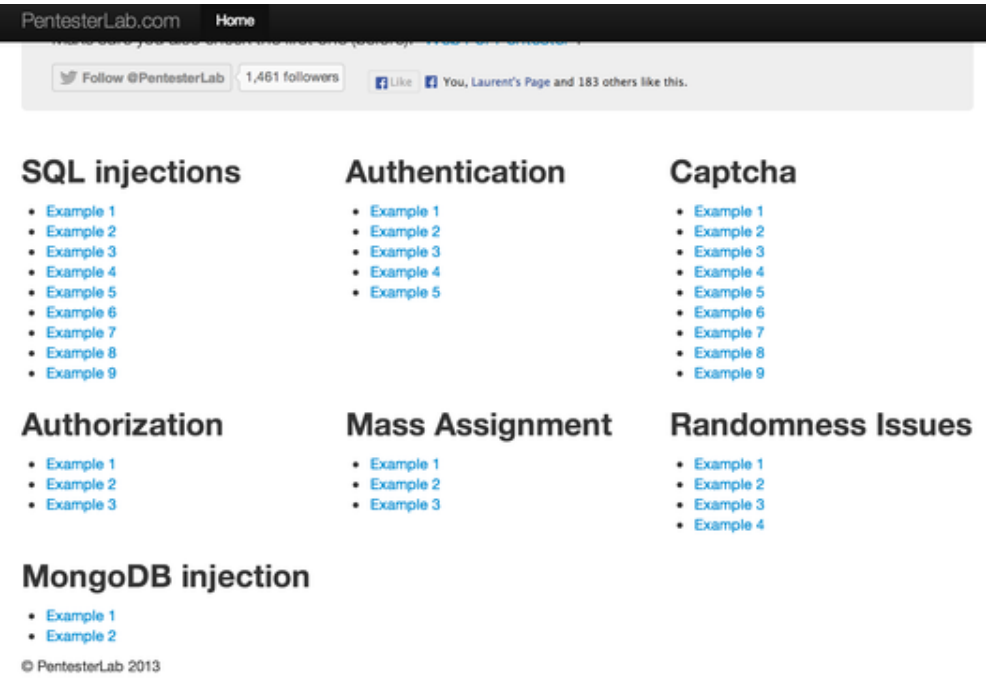


# Introduction

If you haven't done it already, make sure you check out our first exercise: [Web For Pentester](#). It's important that you start with it before starting this one.

If you feel really comfortable, you can try to exploit these vulnerabilities without following the course. You just need to be able to write small scripts in order to send HTTP requests so that you can finish all of these exercises.



# Introduction

## SQL injections

In this section, some common SQL injection examples are provided, the first examples are authentication bypasses where the other ones are more traditional SQL injections.

### Example 1

The first example is the most common SQL injection example that you can find. The goal here is to bypass the authentication page.

This example is the best known way to bypass an authentication page. It's even used in a lot of comics related to SQL injections. Let's see what happens... The initial query looks like:

```
SELECT * FROM users WHERE username='[USERNAME]' AND password='[PASSWORD]'
```

The `[USERNAME]` and `[PASSWORD]` are under your control. The application will check that the `[USERNAME]` and `[PASSWORD]` are correct by ensuring that at least one record is returned by the SQL query. Therefore, the SQL injection needs to ensure that at least one record is returned even if the `[USERNAME]` and `[PASSWORD]` are incorrect.

There are many ways to perform this task. Your best bet is to inject in the `[USERNAME]` since the `[PASSWORD]` may be hashed or encrypted (even if it's not in this example).

First you need to keep in mind the `OR` operator:

|                   |                |                |
|-------------------|----------------|----------------|
| <code>`OR`</code> | <code>0</code> | <code>1</code> |
| <code>0</code>    | <code>0</code> | <code>1</code> |

| `OR` | 0 | 1 |
|------|---|---|
| 1    | 1 | 1 |

We will use this to make sure the condition is always- **true** ( **1** ). Our goal is to use the **[USERNAME]** to inject our always- **true** condition but first we need to break out of the SQL syntax using a single quote **'** :

```
SELECT * FROM users WHERE username='' AND password='[PASSWORD]'
```

The query's syntax is now invalid (since there is an odd number of quotes), but we will come back to this later. So far our payload is just a single quote **'** , we now need to inject our always- **true** condition. The easiest way is to use **or 1=1** since **1=1** is **true** the condition will always be **true** . Our query now looks like:

```
SELECT * FROM users WHERE username='' or 1=1 ' AND password='[PASSWORD]'
```

The query's syntax is still incorrect. It's the last problem to solve in our injection; we need to get rid of the end of the query. We can use comments ( **--** or **#** ) to get rid of it:

```
SELECT * FROM users WHERE username='' or 1=1 -- ' AND password='[PASSWORD]'
```

This way MySQL will only see:

```
SELECT * FROM users WHERE username='' or 1=1 --
```

Our final payload is **' or 1=1 --** . This payload can be optimized to **'or 1#** to bypass some filtering, since MySQL will accept this syntax.

Commenting using **--** can often create problem if **--** is not followed by a space. That's why it's always a good idea to add a space at the end.

Once the payload is ready, you can just put it in the form and submit. If you directly inject the payload in the URL, you will need to encode some characters ( **=** , **#** and spaces). You can check [man ascii](#) or the first "Web For Pentester" for more details on URL-encoding.

## Example 2

This example is the same vulnerability with a twist. In the first example, the code only checked that something was returned. In this version, the developer decided to ensure that only one user exists.

To bypass this restriction, you can get all the rows with the trick seen above and then limits this number using the SQL keywords **LIMIT** .

## Example 3

In this example, aware of the risk of SQL injection, the developer decided to block single quotes **'** by removing any single quote **'** in the query. However, there is still a way to break out of the SQL syntax and inject arbitrary SQL.

To do so, you need to think of the query:

```
SELECT * FROM users WHERE username='[username]' and password='[password]'
```

The problem here is that you cannot, in theory, break out of the single quotes **'** , since you cannot inject any quote. However, if you inject a back-slash **\** , the second **'** in the query (the one that is supposed to finish the string **[username]** ) will be escaped and will be closed by the third one (the one that is supposed to start the string **[password]** ).

Using that, you can then use the parameter **password** to complete the query and return an always true statement. Don't forget to comment out the end of the query to avoid the remaining SQL code.

## Example 4

In this example, the developer puts part of the query directly in a parameter. It's really rare in traditional web applications but can sometimes be found in web services, especially for mobile applications. You are injecting directly in

the **WHERE** statement and can manipulate the request to retrieve anything you want.

## Example 5

In this example, you are injecting after the keyword **LIMIT**. On MySQL, this type of injection can only be exploited using **UNION SELECT...** if there is no **ORDER BY** keywords used in the query. Furthermore, the **ORDER BY** keywords need to be located before the **LIMIT** keyword in order for the query to be valid. So you cannot get rid of it using comments.

Some methods exist to exploit injections in **LIMIT** with **ORDER BY** using the **INTO OUTFILE** or **PROCEDURE ANALYSE()** but they are a bit too complex to be covered here.

If there is an **ORDER BY** keyword, you can try to remove the corresponding parameter from the HTTP request to see if it allows you to get rid of the statement in the query.

You can simply use union-based exploitation to retrieve arbitrary information in this example. A full example of this type of exploitation is available in the PentesterLab's exercise: "From SQL injection to Shell".

## Example 6

This is another example of SQL injection, but this time after the **GROUP BY** keywords, union-based exploitation can also be used to exploit this type of issue. The good thing is that **ORDER BY** will be located after **GROUP BY**. So even if **ORDER BY** is used, you can get rid of it using SQL comments.

## Example 7

In this example, two queries are performed. The first query retrieves the user details based on the parameter **id**; the second one uses the username from the previously retrieved record to retrieve the user.


To exploit this issue you will need to use blind SQL injections. However, since the error messages are displayed, we can use error-based exploitation to get information.

The idea behind error-based exploitation is to use error messages to gather information. By injecting error-prone statements, we can get information directly from the error messages instead of using a blind SQL injection.

For example, you can use the following statement:

```
extractvalue('%3Cxml%3E',concat(%22/%22,(select%20version())))
```

by accessing [http://vulnerable/sqlinjection/example7/?id=extractvalue\('%3Cxml%3E',concat\(%22/%22,\(select%20version\(\)\)\)](http://vulnerable/sqlinjection/example7/?id=extractvalue('%3Cxml%3E',concat(%22/%22,(select%20version()))) to get the following error message:

 **vulnerable/sqlinjection/example7/?id=extractvalue('<xml>',concat('/',(select%20version()))**

**PentesterLab.com**    **Home**

**Mysql2::Error: XPATH syntax error: '5.1.66-0+squeeze1': SELECT \* FROM users WHERE id=extractvalue('<xml>',concat('/',(select version()))**

| id | name |
|----|------|
|    |      |

**© PentesterLab 2013**

It's a really good way to demonstrate that a page is vulnerable to SQL injections and that you can gather information from the database.

## Example 8

This example is vulnerable to "second order SQL injection", instead of directly injecting your payload into the request, you will first **insert** it in the database using a first request and then trigger the payload in a second request. The first request is not vulnerable to SQL injection, only the second is. However, you do not directly control the value used; you need to inject it using the first request. This issue comes from the fact that the developer trusted the values coming from the database.

Each attempt will need two steps:

- Create a user with your payload.

- Access this user information to trigger your payload.

If you want to be efficient, you need to automate this process using a simple script. The payload can be as simple as a union-based exploitation.

## Example 9

This example was first published in 2006 on [Chris Shiflett's Blog](#) as a way to bypass `mysql-real-escape-string`. It relies on the way MySQL will perform escaping. It will depend on the charset used by the connection. If the database driver is not aware of the charset used it will not perform the right escaping and create an exploitable situation. This exploit relies on the usage of `GBK`. GBK is a character set for simplified Chinese. Using the fact that the database driver and the database don't "talk" the same charset, it's possible to generate a single quote and break out of the SQL syntax to inject a payload.

Using the string `\xBF` (URL-encoded as `%bf%27`), it's possible to get a single quote that will not get escaped properly. It's therefore possible to inject an always-true condition using `%bf%27 or 1=1 --` and bypass the authentication.

As a side note, this issue can be remediated by setting up the connection encoding to 'GBK' instead of using an SQL query (which is the source of this issue). Here the problem comes from the execution of the following query:

```
SET CHARACTER SET 'GBK';
```

It is a pretty unlikely issue for a web application but it's always good to know that it exists, especially if you play CTFs.

## Authentication issues

This section puts together example of issue in authentication pages: from trivial brute-force to more complex issues.

### Example 1

This example is really simple. Just carefully read the prompt and you should be in pretty quickly. Common passwords are probably the easiest and most common way to bypass authentications.

### Example 2

This example is an exaggerated version of a non-time-constant string comparison vulnerability. If you compare two strings and stop at the first invalid character, a string `A` with the first 6 characters in common with the string `B` will take more time to compare than a string `A'` with only the first 2 characters in common with the string `B`. You can use this information to brute force the password in this example.

Here the username is provided in the prompt, you just need to find the password. To do so you need to loop through all characters until you find the one that took the most time, since the application compares one more character.

First, you need to be able to send HTTP request with an `Authorization: Basic` header. The format used is:

```
GET /authentication/example2/ HTTP/1.1
Host: vulnerable
Authorization: Basic aGFja2VyOnBlbnRlc3RlcmxhYgo=
```

Where the string `aGFja2VyOnBlbnRlc3RlcmxhYgo=` is the base64 of `username:password`.

Once you have the code to send this request, you will need to loop on all the character set (here it's limited to lowercase characters and numbers) and check how much time each request takes.

For example, this is the output of my script:

```
$ ruby auth-example2.rb
hacker:a -> 1.4625000953674316
hacker:b -> 1.4070789813995361
hacker:c -> 1.407270908355713
[...]
hacker:l -> 1.4061241149902344
hacker:m -> 1.4065420627593994
hacker:o -> 1.4070839881896973
hacker:p -> 1.6072182655334473
```

```
[...]  
hacker:4 -> 1.4077167510986328  
hacker:5 -> 1.4075558185577393  
hacker:6 -> 1.40665602684021  
hacker:7 -> 1.4062080383300781  
hacker:8 -> 1.4082770347595215  
hacker:9 -> 1.407080888748169
```

You can see that the letter **p** took more time than the others.

It's also confirmed when you try to guess the next letter:

```
[..]  
hacker:pa -> 1.6075811386108398  
hacker:pb -> 1.608860969543457
```

Finally, the script needs to exit once you get a HTTP 200 response. This means that the correct credentials have been found.

This technique is likely to fail depending on how stable your setup is. A good idea is too keep monitoring the script and manually review the result. You should be able to easily restart the script from where it stopped (for example after a connection issue) by adding the value you already guessed to the username **hacker:**.

This example was one of the easy web challenges during Ruxcon 2012 CTF.

## Example 3

In this exercise, you can log in as **user1**. Your goal is to get logged in as **admin**. To do so, you need to carefully look at the response sent back by the server.

You can see that when you log in as **user1**, you get a cookie named **user1**. From that you can easily modify this value (using a proxy or a browser's extension) to get logged in as **admin**.

## Example 4

This example is similar to the previous example. As soon as you receive a cookie from an application, it is always good to see what it looks like. Try to crack it using a password cracker or try to just Google it. From that you should be able to generate a valid cookie for the user **admin**.

If you get the same session ID many times when logging in: there is a problem! If you log in from a clean browser, you should **never** get the same cookies twice.

## Example 5

This example shows the consequence of different method of string comparison. When you create a user, the application will check programmatically that the user does not exist by comparing the username provided with the existing user. When you log in, the application will check that your username and password are correct, and then it will save your username in your session. Finally, every time you will access the application, the application will retrieve your user's details based on the username provided in the session.

The trick here comes from the fact that the comparison when you create a user is done programmatically (i.e.: in Ruby) but when the user's details get retrieved, the comparison is done by the database. By default, MySQL (with the type **VARCHAR**) will perform a case insensitive comparison: "admin" and "Admin" are the same value.

Using that information, you should be able to create a user that will be identified as **admin**.

## Example 6

To remediate the previous issue, the developer decided to use a case sensitive comparison during users' creation. This check can also be bypassed based on the way MySQL performs string comparison: MySQL ignores trailing spaces (i.e.: **pentesterlab** and **pentesterlab** are equals). Using the same method as above, you should be able to pretend to be logged in as the user **admin**.

A good way to prevent this issue is to tell the database that the username is a PRIMARY KEY. This method is, for example, used in Tomcat documentation to use a SQL backend as a Realm.

# Captcha

When attacking Captcha, and before starting some hardcore coding, make sure there is no logic flaws or some kind of predictability. If you can bypass a captcha without breaking it... Don't break it!

Attention to details is key to find logic flaws in captcha. Make sure you check every details of the response when you need to crack a captcha.

The first examples are badly developed captchas with common logic flaws, later examples are easier to break and can be broken.

In all the following examples, the goal is to build automation around the captcha with a high success rate (100% for most of them).

I have no real knowledge of image processing and related research in the domain of captcha. These methods are mostly hacks and don't represent the state of the art in this domain.

## Example 1

This script is a common issue with badly implemented captcha. To avoid error message the developer checks that a captcha parameter exists before ensuring that its value is correct:

```
if params[:captcha] and params[:captcha] != session[:captcha]
  # ERROR: CAPTCHA is invalid redirect
  [...]
end
# CAPTCHA is valid
[...]
```

However, this example presents a vulnerability: if no captcha is provided, the script does not fail safely.

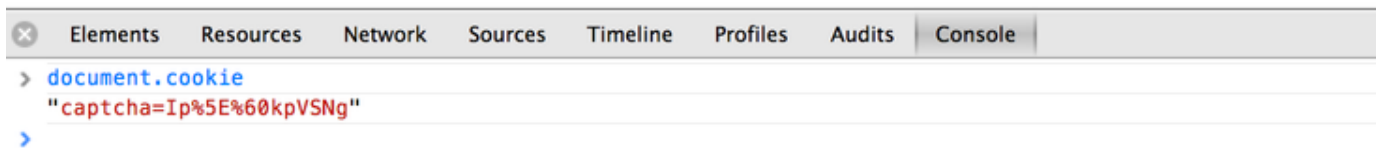
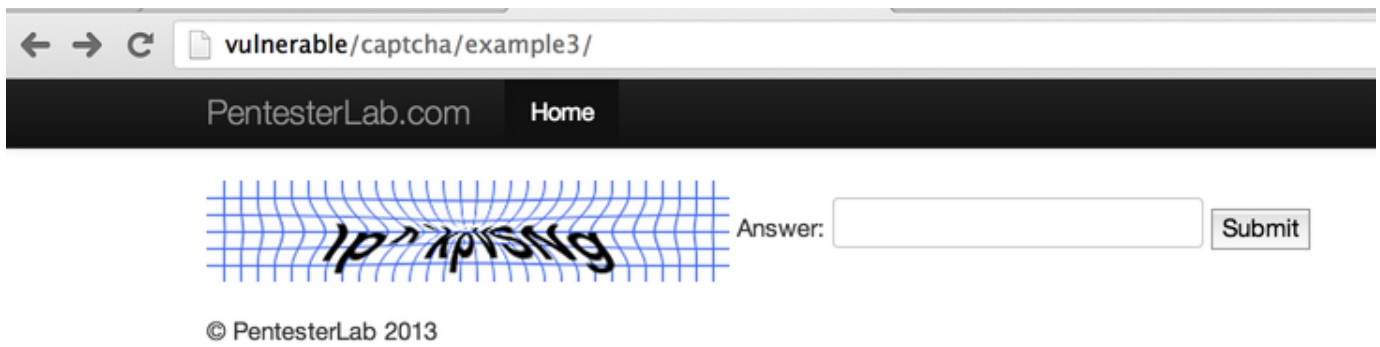
## Example 2

In this example, the answer is leaked by the application. By inspecting the source of HTML page returned, you should be able to write a script that can automatically break this captcha.

## Example 3

In this example, the answer is leaked by the application. By inspecting the response sent back by the server, you should be able to write a script that can automatically break this captcha.

You can also retrieve the captcha from the cookie using the JavaScript console and call `document.cookie` :



## Example 4

This is quite a funny example, since it's a mistake I made during the development of this set of exercises.

Here, you don't have to really crack the captcha. You just need to crack it once and you can reuse the same value and session ID to perform the same request again and again. When you try to crack a captcha, make sure that an answer can only be used once. You can easily script this exploitation by writing a script that takes a session ID and a value for parameters, and submit them again and again.

## Example 5

This example is the last example of weakness, here the weakness comes from the dictionary used to create the captcha; there are only a limited number of words used. You can easily write a cracker by generating a list of all words and the MD5 of the image. Then, when you want to submit the form, you just need to retrieve the image, compute its MD5 and submit the matching word.

## Example 6

In this example, we are going to use the OCR tool ([Tesseract](#)) to crack this easy captcha. The goal here is to build a script that will get a high success rate.

Just with a basic script using tesseract, you can expect a success rate of more than 95%. You will need to use the following algorithm:

- Go to the main page <http://vulnerable/captcha/example6/> to get a new captcha and the cookie ( `rack.session` ).
- Retrieve the image.
- Run tesseract on the image and retrieve the result.
- Submit the result with the correct cookie.

The following things can improve your success rate:

- Only submit a value if it's a word.
- Only submit a value if it only contains lower case characters.

Depending on the application workflow, you may want to have a really high success rate. For example, if you spend 10 minutes filling forms, you want to make sure that the captcha cracker has a high success rate. Where if it's only to exploit a SQL injection, you can just retry until you find the right value and you don't need to be really accurate.

## Example 7

If we use the same technique as the one above, we can see that the success rate is significantly lower. To improve the recognition, we are going to try to remove the blue "HatchFill".



A really simple way to do that, is to use a threshold to modify the image before running `tesseract` on it. This can be done by the following code:

```
require 'RMagick'
image = Magick::Image.read("current7.png").first
image = image.threshold(THRESHOLD)
image.write("current7.png")
```

You can play with the `THRESHOLD` value to improve the detection rate. Just by using this simple trick, the success rate increases. Also, the time required to get this rate is lower since the information detected by `tesseract` matches the conditions (only one word composed of lowercase characters).

## Example 8

Here we can see that the image is imploded. We are going to open the image, un-implode it and use the previous technique to increase the success rate.

The following code can be used:

```
require 'RMagick'
image = Magick::Image.read("current8.png").first
image = image.implode(IMPLODE)
image = image.threshold(THRESHOLD)
image.write("current8.png")
```

You can play with the `THRESHOLD` and `IMPLODE` (`IMPLODE` can be negative) values to improve the detection rate.

As with the previous example, the detection rate will be improved and the time necessary to get this rate will be decreased.

## Example 9

This captcha relies on asking the client to perform a simple arithmetic operation, it's really simple to crack. You can just `eval` the string provided (probably not a good idea) or you can write a simple parser to do the arithmetic operation for you. This kind of protection will protect from really dumb bots but is really easy to bypass for anyone who can do a bit of scripting.

# Authorization

Authorization issues are very common in web applications. This section puts together some common examples of vulnerability. Modern web development frameworks often protect from all kind of injections but cannot take care automatically of this kinds of issues automatically, since they cannot understand the business logic behind them. Authorization issues cannot really be tested by automatic web scanners for the same reason; that's why it's often a good source of vulnerabilities and it's important to know how to test them. The following section "Mass Assignment" is also a problem of authorization but I prefer to separate the exercises in two sections for clarity.

## Example 1

In this example, you can log in with the following user: `user1`, with the password "pentesterlab". Once you've logged in and play around, log out and try to access the same information.

This is a pretty common issue with poorly designed web applications, even if you cannot access the post-authentication page, you are still able to access other pages if you know their URLs: <http://vulnerable/authorization/example1/infos/1> for example.

## Example 2

In this example, you can log in with the following user: `user1` with the password `pentesterlab`. Once you are logged in, you can start accessing information and see the pattern used: `/infos/1`, `/infos/2`. If you keep incrementing the number in the URL, you can access information from the other users.

## Example 3



In this example, you can access the information using a method similar to the one seen previously. You cannot just directly access the information, however you can see that you are now able to edit information, and you can use this feature to access information from other users just by incrementing the number in the URL.

# Mass-Assignment attacks

When people started building website with database to store information, they had to do a lot of SQL manually. Few people realized that it was not the best solution and started working on smarter alternatives and started building Object-relational mapping (ORM) to easily query the database without any SQL knowledge. For example, in Ruby (using ActiveRecord), you can do things like:

```
@user = User.find_by_name('pentesterlab')
```

This will automatically generate and execute the query and retrieve the result in a `User` object.

Another really handy usage was to automatically create and update an object from a hash:

```
@user = User.create(myhash)
[...]
```

```
@user.update_attributes(anotherhash)
```

Unfortunately, this useful feature comes with a security price, if a developer did not correctly ensure that attributes of the object `@user` were protected, an attacker could arbitrary overwrite any of these attributes. In this section, we will see some common examples of these types of issues: Mass-Assignment.

## Example 1

In this example, you can register a user. The application has two levels of privileges:

- User.
- Admin.

The admin privilege is set using the attribute `admin` on the object `user`. If you look closely at the format used by the web application: `user[username]` and `user[password]`, you should be able to find a way to get `admin` access. Three methods can be used:

- Modify the page directly using a browser extension.
- Save the page and modify offline to create a page that will send the right payload to the right URL.
- Use a proxy to intercept the legitimate request and add your parameter (fastest option).

## Example 2

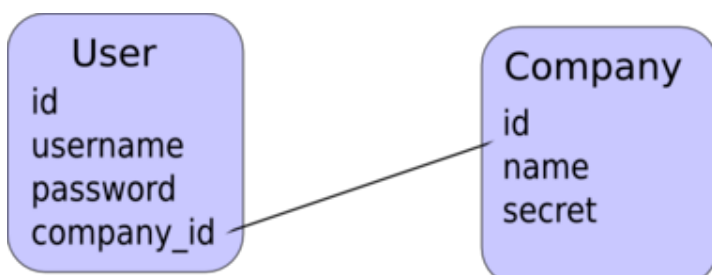
In this exercise, the developer fixed the previous bug. You cannot create a user with `admin` privileges... or at least no directly. Try to find a way to do the same thing.

## Example 3

In this exercise, you can log in with the following user: `user1` with the password `pentesterlab`. Once you logged in, try to access the information from the company "Company 2".

To do so you will need to modify your company using mass-assignment.

By convention (can be changed programmatically) when a developer uses ActiveRecord (Ruby-on-Rails' most common data mapper), and a class `Company` has multiple `User`, the relation is managed using a field `company_id` inside the `User` class:



The following code is used in Ruby:

```
class User < ActiveRecord::Base
  belongs_to :company
end

class Company < ActiveRecord::Base
  has_many :users
end
```

Ruby-on-Rails enforces "convention" over "configuration" which really helps to guess class names and attributes' name...

Using this information, you should be able to modify your current company to get access to the "secret" of the other company. Once you get this "secret", you can reset your `company_id` to get back to your company's details.

## Randomness Issues

Depending on how a developer generates random numbers or strings, the values created can be more and less random. The most, and biggest, mistake is to manually seed the random generator using a constant or the current time; this will allow an attacker to be able to predict what values have been and are going to be generated.

In this section, we will see some examples that will convince you that random is not always random.

### Example 1

This first example is just here to show how random is not really random. The problem comes from the use of a seeded random generator. The developer used the value `0` to seed the random generator.

If you just replay the script, you should be able to find the password generated for the administrator `admin`.

### Example 2

This example shows another example of bad seeding of a random generator. The random generator is seeded with the current time.

To work your way to the `admin` password, you need to brute force the seed. To do so you can start from the current time and decrease it while replaying the algorithm used to generate the values until you get your password.

Once you get your password, you know what seed was used (or more precisely at what time the random generator was initialized). You can then get the `admin` password.

### Example 3

This example is really simple and similar to the first one. You just need to replay the code to get the `admin` password. The fact that the password's length is random has no impact on the fact that you can guess it.

### Example 4

In this example, you don't know how many times the random generator was used before your password was generated (since it's a call to `rand(1000)` as opposed to `s.rand(1000)`). You can still get the previous password generated. To get it, you just need to brute force this value until you get your password.

## MongoDB injections

Even if MongoDB is a NoSQL database, it's still possible to write vulnerable code and therefore this exercise features two NoSQL-injections.

If you want to try them on your own, try to follow these steps:

- Learn and understand what the MongoDB syntax looks like (find the project's website and read the documentation).
- Find what can be used to get rid of the code after your injection point (comments, null byte).
- Find how you can generate always `true` conditions (for example 1).

- Find how you can retrieve information (for example 2).

As opposed to SQL databases who almost support the same syntax, NoSQL databases have different syntax.

## Example 1

This example is the MongoDB version of the (in)famous `' or 1=1 --`. If you remember what you saw previously, you know that you will need two things to bypass this login for:

- An always true condition.
- A way to correctly terminate the NoSQL query.

First, by reading MongoDB documentation you can find that the SQL `or 1=1` translates to `|| 1==1` (note the double `=`). Then by poking around, you can see that a NULL BYTE will prevent MongoDB from using the rest of the query. You can also use the comments `//` or `<!--` to comment out the end of the query.

With this information, you should be able to bypass the authentication form.

## Example 2

In this example, we will try to retrieve more information from the NoSQL database.

Using a bit of guess work (or previous knowledge of the application), we can deduce that there is probably a `password` field.

We can play around to confirm that guess:

- if we access `[http://vulnerable/mongodb/example2/?search=admin'%20%26%26%20this.password.match(/./)/+%00](http://vulnerable/mongodb/example2/?search=admin'%20%26%26%20this.password.match(/./)/+%00)`: we can see a result.
- if we access `[http://vulnerable/mongodb/example2/?search=admin'%20%26%26%20this.password.match(/zzzzz/)/+%00](http://vulnerable/mongodb/example2/?search=admin'%20%26%26%20this.password.match(/zzzzz/)/+%00)`: we cannot see a result.
- if we access `[http://vulnerable/mongodb/example2/?search=admin'%20%26%26%20this.passwordzz.match(/./)/+%00](http://vulnerable/mongodb/example2/?search=admin'%20%26%26%20this.passwordzz.match(/zzzzz/)/+%00)`: we get an error message (since the field `passwordzz` does not exist).

Now, we have a way to perform a blind injection since we have two states:

- No result when the regular expression does not match something: `false` state.
- One result when the regular expression matches something: `true` state.

Using this knowledge, we can script the exploitation to guess `admin` password. We will first ensure that the matching is done correctly by using: `^` and `$` to make sure we do not match characters in the middle of the string (otherwise iterating will be far harder).

The algorithm looks like:

- test if password match `/^a.$/` if it matches test without the wildcard ```. Then move to the next letter if it does not match.
- test if password match `/^b.$/` if it matches test without the wildcard ```. Then move to the next letter if it does not match.

For example, if the password is `aab`, the following test will be performed:

- `/^a.*$/` that will return true.
- `/^a$/` that will return false.
- `/^aa.*$/` that will return true.
- `/^aa$/` that will return false.
- `/^aaa.*$/` that will return false.
- `/^aab.*$/` that will return true.
- `/^aab$/` that will return true. The password has been found.

With these details, you should be able to retrieve the password for the user `admin`.

In case the password field does not exist for some records (since it's a NoSQL database), it's always a good idea to ensure its presence by using `... && this.password && this.password.match(...)` instead of just using `... && this.password.match(...)`.

# Conclusion

This exercise follows our first exercise "Web For Pentester" and provides a really good course for people who want to progress in doing web application penetration testing. If you are interested by this subject, you should check out our other exercises available at the following address: <https://www.pentesterlab.com/>. Other exercises are more scenario-based and more realistic of typical web engagements. I hope you enjoyed learning with PentesterLab.