**Web Security and Program Security**

**Part I – Vulnerability Analysis**

**Task 1:**Assume that you are the system administrator. Your task for this project would be to go through the code (you will have to go through the file logCleaner in the cronjob directory and also all the supporting files) and understand what it is trying to do and how it works. Your task is greatly simplified since you are just going to go through 165 lines of code. Most administrators have to go through thousands of lines of code! Write a paragraph in your project report on your analysis of the logClean code and its flow of control. Be very precise and to the point here. We are not expecting a line–by–line explanation of the code. An overview covering all the important points would suffice.

**Sol.**

Basic flow control is as given below:

* Initialization of variables –   
   my $BaseDir = "..";  
   my $TempDir = "/tmp";

my $LogDir = "../logfiles";

my $PreProcess = "../services/preprocess";

my $Version = '1.0';

my $VDate = '08/31/04';

* Add “/” to BaseDir  
  Value of **BaseDir** is “**..**”. We append a backslash to the value BaseDir. Therefore -   
   unless ($BaseDir =~ m=/$=) {

$BaseDir = $BaseDir . "/";

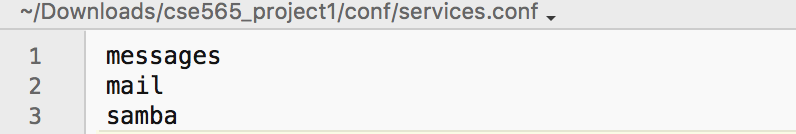
}  
Here value of BaseDir becomes - “../”

* Make a call to subroutine – ReadConfigFile   
  ReadConfigFile ($BaseDir . "conf/services.conf");

After that read each line of “services.conf” under “conf” folder and store in variable - “ThisValue”.   
 while (defined($ThisLine = <READCONFFILE>)) {

$ThisLine =~ s/#.\*$//;

chomp ($ThisLine);

$ThisValue = lc($ThisLine);  
  
Then we further ‘push’ this value to array – ReadConfigValues  
push @ReadConfigValues, $ThisValue;  
Since, services.conf contains – “messages”, “mail” and “samba” as visible in the screenshot below -  
    
Therefore, **ReadConfigValues = [‘messages’, ‘mail’, ‘samba’]**

* Push these values in - @ServiceList and @LogFileList too:  
   for ($i = 0; $i <= $#ReadConfigValues; $i++) {

push @ServiceList, $ReadConfigValues[$i];

push @LogFileList, $ReadConfigValues[$i];  
Therefore,   
**ServiceList = [‘messages’, ‘mail’, ‘samba’]**

**LogFileList = [‘messages’, ‘mail’, ‘samba’]**

* Create temporary arrays -   
   @TempLogFileList = ();

@TempServiceList = ();  
  
Since both the arrays are empty, therefore -   
**$#TempLogFileList; == -1 and $#TempServiceList; == -1**  
Had the either of the two arrays contained any values, then these values would be stored in $LogFileList for TempLogFileList and @ServiceList for TempServiceList.   
 if ($#TempLogFileList > -1) {

…

}

if ($#TempServiceList > -1) {

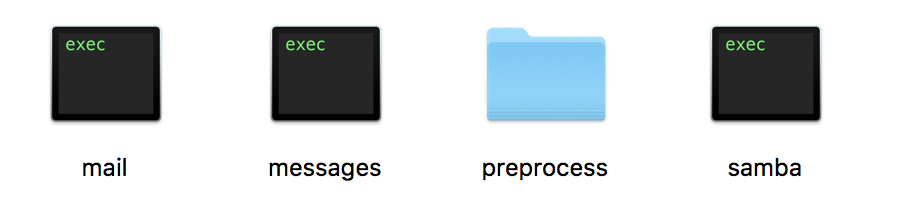
…

}  
But since both *TempLogFileList* and *TempServiceList* are empty, therefore both the “if” conditions are false.

* If there is nothing in “services.conf”, then both *ServiceList* and *LogFileList* would be empty. In this case, it is assumed that all the services are to be performed. Therefore, store ‘all’ in ServiceList and LogFileList -   
   if ( ($#ServiceList == -1) and ($#LogFileList == -1) ) {

push @ServiceList,"all";

}

* Read all services present on the system – That is, all the files present in “services” folder   
  and store the file names list in – “AllServices”  
    
   opendir(SERVICESDIR,$BaseDir . "services") or die $BaseDir . "services/, no such directory.";

while (defined($ThisFile = readdir(SERVICESDIR))) {

unless (-d $BaseDir . "services/" . $ThisFile) {

$ThisService = $ThisFile;

$ThisService = lc($ThisService);

**push @AllServices, $ThisService;**

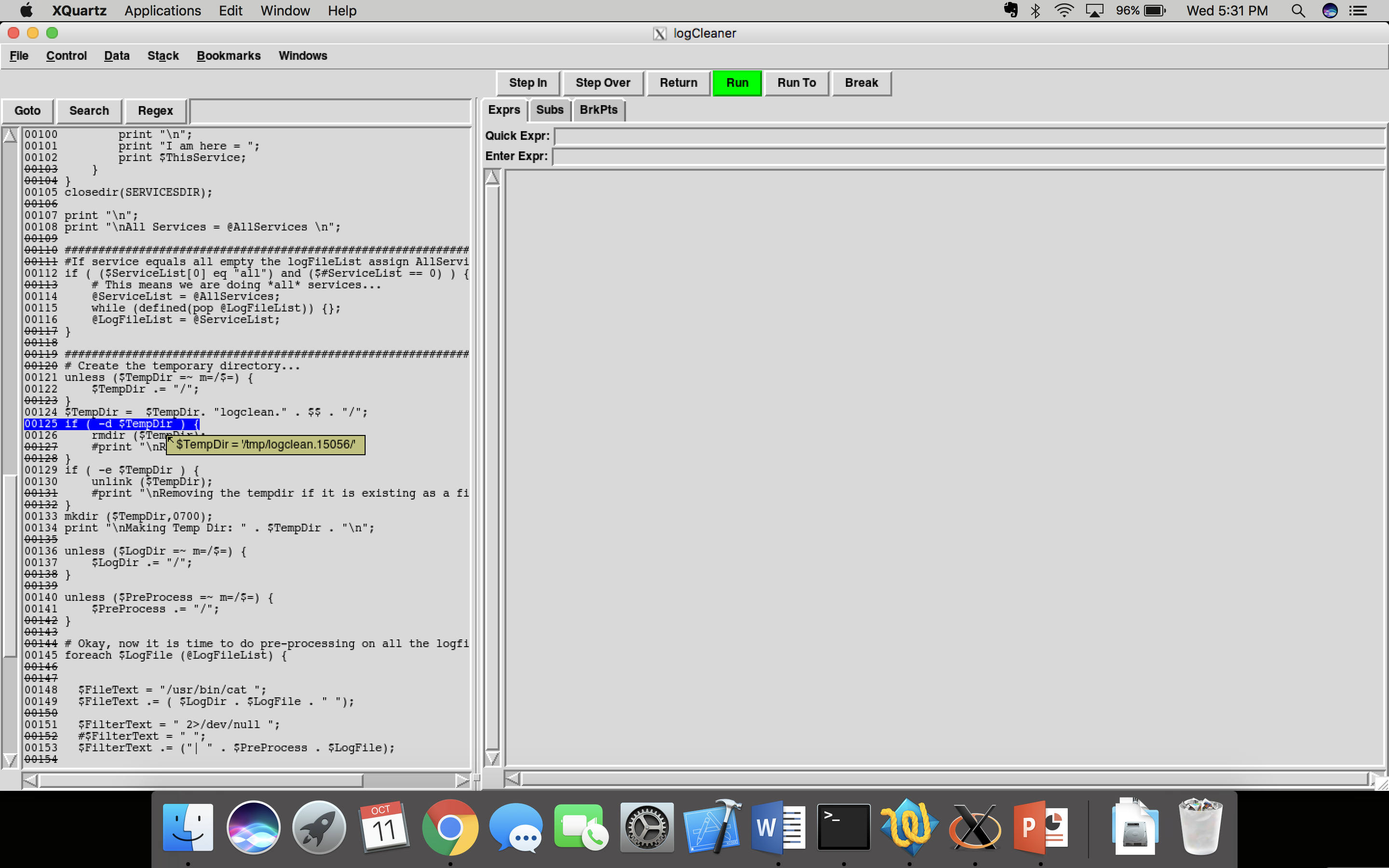
* If the value of the first element of “*ServiceList*”, that is – “*ServiceList*[0]” equal to ‘all’ and there is only one element in *ServiceList*, that is – “*$#ServiceList == 0*” (as -1 is for empty), then store all values of *AllServices* into *ServiceList*   
   if ( ($ServiceList[0] eq "all") and ($#ServiceList == 0) ) {

@ServiceList = @AllServices;

Then further store all values of *ServiceList* to *LogFileList*, for all the values for which the values are different -   
 while (defined(pop @LogFileList)) {};

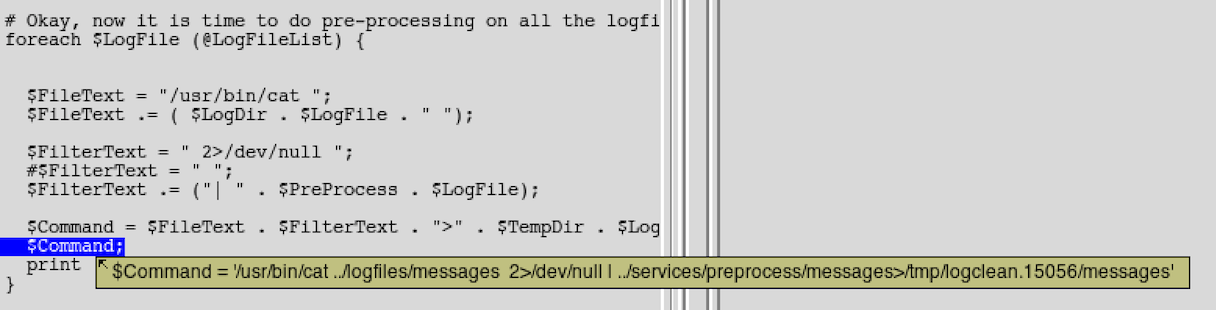
@LogFileList = @ServiceList;

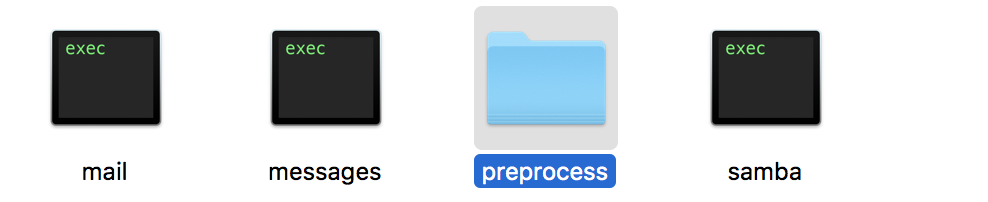
* Creating a temporary Directory –
  + TempDir = /tmp/logclean.15056
  + Check if the temporary dir already exists – if( -d $TempDir)
  + Remove TempDir – if it is existing as a file – if( -e $TempDir)

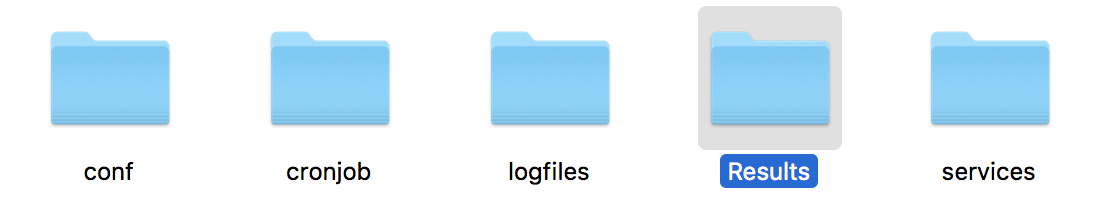


* Add / to LogDir i.e. = $LogDir = “../logfile/”  
   unless ($LogDir =~ m=/$=) {  
   $LogDir .= "/";
* Add / to PreProcess i.e. = $Preprocess = “../services/preprocess/”  
   unless ($PreProcess =~ m=/$=) {

$PreProcess .= "/";

* Do pre-processing on all files in $LogFileList  
  
  + For each loop, we preprocess the files in folder “preprocess” in the “services” folder and the filtered text is stored in temporary directory which was created:   
    – “/tmp/logclean.15056/messages”  
    – “/tmp/logclean.15056/mail”

– “/tmp/logclean.15056/samba”  


* Do Actually Processing –
  + For each loop, we use the temporary files stored at the temporary location which is created and final results are stored in the “Results” folder.  
      
    – “../Results/messages”

– “../Results /mail”

– “../Results /samba”

**Task 2:**

After developing a complete understanding of what the code does and with the details provided above, you are to track the vulnerability in the code. You have to figure out what might be the possible flaws in the code that an attacker might take advantage of (you have to start thinking like a hacker!). Write a paragraph on the vulnerability explaining how you thought about it and how can a malicious user take advantage of it. Be very concise here as well. Be sure to mention your chain of thoughts while analyzing the code which led to the specific conclusion by you about the vulnerability.

**Sol.**

While analyzing the code we observe that this code first reads a file and then based on the input from that file, performs further operations and generates temporary logfiles which are then used to generate the final results. There are many different ways in which the code can be easily compromised. If we assume that the hacker does not have access to the initial “services.conf”, then in the code where the file is read, and then the contents of the file are added to the array “ReadConfigValues” – this is where any further information could be injected. Since this result is used by “ServiceList” and “LogFileList”, changing the values to these arrays will allow the hacker to change the input. This can be easily accomplished by putting values for “TempLogFileList” and “TempServiceList” – which, unless empty, are copied to “ServiceList” and “LogFileList”. As a result, the data sent to get the final result will be compromised, and the final result will too be invalid.

**Task 3:**

After figuring out the weakness in the code, try to form a pseudo code (or a code in any language of your choice, this is just because a lot of people prefer writing code in a language they are comfortable with like C, C++ or Java rather than writing a pseudo code) which exploits the vulnerability you detected in the tool logClean.

**Sol.**

**Pseudo Code:**

We will write code to compromise the data that is being stored in “ServiceList” and “LogFileList”. This data is being stored in these two arrays after the call to the subroutine “ReadConfigFile”, where “ReadConfigValues” is input with the values from “services.conf”.

We have created a subroutine – “UpdateTempLists”, which will update the values for “TempLogFileList” and “TempServiceList”. Since if “TempLogFileList” and “TempServiceList” are not empty, they will replace the values for “LogFileList” and “ServiceList”, this is an easy way to update the lists. The rest of the code will remain the same –

#Code before this remains same – Added code has been written in bold

. . .

**sub UpdateTempLists {**

**my $valuesToHackServiceList() = $\_[0]; #First input array to this subroutine**

**my $valuesToHackLogFileList() = &\_[1]; #Second input array to this subroutine**

**for ($i = 0; $i <= $# valuesToHackServiceList; $i++) {**

**push @ TempServiceList, $ valuesToHackServiceList[$i];**

**}**

**for ($i = 0; $i <= $# valuesToHackLogFileList; $i++) {**

**push @ TempLogFileList, $ valuesToHackLogFileList [$i];**

**}**

**}**

@TempLogFileList = ();

@TempServiceList = ();

**UpdateTempLists($<invalid or mis-directing input to replace the original values of ServiceList>, $<invalid or mis-directing input to replace the original values of LogFileList> );**

if ($#TempLogFileList > -1) {

@LogFileList = @TempLogFileList;

for ($i = 0; $i <= $#LogFileList; $i++) {

$LogFileList[$i] = lc($LogFileList[$i]);

print $LogFileList[$i];

}

@ServiceList = ();

}

if ($#TempServiceList > -1) {

@ServiceList = @TempServiceList;

for ($i = 0; $i <= $#ServiceList; $i++) {

$ServiceList[$i] = lc($ServiceList[$i]);

print $ServiceList[$i];

}

}

. . .

#Rest of the code remains the same

**Task 4:**

Your next task would be to patch the vulnerability. Here your task would be to suggest a way to patch the vulnerability. You are not expected to code the patch, but you have to provide details on fixing the vulnerability. Include a paragraph in your project report giving details on how one can patch (fix) the vulnerability in the tool logClean without rewriting a lot of code or major modifications in the program structure.

**Sol.**

We can patch the vulnerability by checking the values for “LogFileList” and “ServiceList” and comparing it with the original value of “ReadConfigValues” which was obtained in the call to the subroutine “ReadConfigFile”. This contains the values which were directly read from “services.conf” (We are assuming that “services.conf” has not been modified or hacked).

This is like a checksum, where the final input, before processing is compared by the original input.

Also, the user can be notified whenever the value of “TempLogFileList” and “TemoServiceList” is not empty, to check if the value was correctly updated. This can be easily done without any major change to the original code.

**Part II – Web Security**

**Q1** List any four malicious objectives that an attacker can achieve using SQL injection attack?

**Sol:** An SQL injection can allow an attacker to do the following:

1. The attacker can use SQL Injection attack to gain unauthorized access to an application without providing a valid username or password. The attacker may also impersonate another authorized user to bypass other security checks.
2. The attacker can directly or indirectly gain access to sensitive information from the database. Since SQL injection is based on SQL syntax and commands, the attacker can easily use SQL to retrieve data from the database.
3. If the attacker can gain authoritative access to the server, he can alter sensitive data by inserting malicious values, updating record or deleting even the entire table. Altering sensitive information can affect data integrity and financial operations.
4. Some databases allow the execution of Operating Systems commands on the DB server. The attacker can easily use SQL injection as an initial step for an attack to an internal network that is protected by a firewall.

References: 1. [https://www.cisco.com/c/en/us/about/security-center/sql-injection.html 2](https://www.cisco.com/c/en/us/about/security-center/sql-injection.html%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%202). <https://www.acunetix.com/websitesecurity/sql-injection/>

**Q2** Can SQL attack enable an attacker to issue commands to a database? Can SQL attack enable an attacker to issue commands to an operating system?

**Sol:**  SQL is a technical language that is used for managing data stored in databases, therefore SQL can be used to access, modify and delete data of a database. When the right conditions are created, the attacker can modify the SQL injection query to issue commands to the operating system. Furthermore, if the attacker can impersonate an authorized user, he can easily bypass the firewall network that protects the various elements of the system. Keeping the above in mind, when considering the above, it’s easier to understand how lucrative a successful SQL Injection attack can be for an attacker.

References: <https://www.acunetix.com/websitesecurity/sql-injection/>

**Q3 a)** Can you add a new employee to the database? Explain your answer and write down the SQL queries involved, if any.

**Sol:** Yes we can see a full list of employees in the database by generating a query that always returns TRUE in the end. SQL injections is based on ‘1’=’1’ is always true. For example, consider the following SQL injection query that always returns ‘TRUE’. For example:

SELECT \* FROM Users WHERE LASTNAME = ‘JONES’ OR 1=1;

References: <https://www.w3schools.com/sql/sql_injection.asp>

**Q3 b)** Can you add a new employee to the database? Explain your answer and write down the SQL queries involved, if any.

**Sol:** Yes we can create a new employee using SQL injection. The attacker can inject arbitrary data if the inputs are not sanitized. We can insert a new user with an SQL query having a single quote (‘) and a comment string (-- -): For example, if we give the value ‘$NAME,’PASS’,’SIGN’)—for the username field web form, we get the confirmation ‘account created for $NAME’,’PASS’,’SIGN’)-- -. 1 row inserted’. For example:

SELECT USERNAME, PASSWORD, LOGINID

FROM members

Where LASTNAME=’X’;

INSERT INTO members (‘USERNAME’,’PASSWORD’,’LOGINID’)

VALUES (‘$FIRST\_NAME’,’$1234\_PASS’,’\_FIRSTN’);-- ‘;

**Q3 c)** Can you update employee information? Explain your answer and write down the SQL queries involved, if any.

**Sol:**  Yes, we can update employee information using SQL query. Let’s assume that we know that a user with the email ID ‘$EMAIL\_ID@XYZ.com’ in the database. We can use our SQL query to update his email id in the system, with any email id we want.

Let’s further assume that the parameter $FIRSTNAME in the database is injectable. Also, let the application uses the $FIRSTNAME field to create the following query:

SELECT \* FROM Employees WHERE $FIRSTNAME=’MARK’ and LOGIN ID=’MARK23’.

Since the parameter $FIRSTNAME is injectable, instead of ‘MARK’ we can insert

MARK; UPDATE Employees SET filed=123456 –

This turns the first query into :

SELECT \* FROM Employees WHERE $FIRSTNAME=MARK; UPDATE Employees SET filed = 123456 - - and $AGE=1, where ; closes the first query and - - comments out anything following it.

**Q3 d)** Can you delete employees with a certain last name? Explain your answer and write down the SQL queries involved, if any.

**Sol:**  Yes, we can delete employee information using SQL Query. Consider the query SELECT \* FROM Employees WHERE FirstName =’$FHS’. Now, If an attacker enters the string "$FIRSTNAME'); DELETE FROM Employees; --" for FirstName, then the query can take the form:

SELECT \* FROM Employees

WHERE FirstName = '$FIRST\_NAME';DELETE FROM Employees;--'

The trailing hyphens (--), specify that the rest of the command is a comment and should not be executed. In this case the comment character serves to remove the trailing single-quote left over from the modified query.

**Q4** Briefly describe and demonstrate the prevention measures that can be taken against SQL injection attacks.

**Sol:** The following preventive measures can be taken to prevent SQL attacks:

1. Stored Procedures: Application should not issue it’s own SQL statements. We should use Stored Procedures to access the database. Stored procedures accept parameters and based on these parameters, handle different requests made. Using stored procedures can ensure that the application cannot access the database tables directly. The application can only give permissions to access the database, thus in this case, even if a malicious attacker tries to access the database tables via SQL injection, the SQL statement would not be able to give the necessary permissions to access the tables.
2. Avoid Dynamic SQL: In cases such as LOGIN, we cannot avoid Dynamic SQL. But we can use functions to escape all the characters which might try to access the database. This method is called White List validation. It validates values of all input fields provided by the user. We should also perform sanity tests on all inputs entered by the user.
3. Strategy of Lease Privilege: Using stored procedures can mitigate the risk of SQL injection but we should also ensure that every database entity has an access grant. System administration accounts should also be avoided as they are most vulnerable to attacks. Having access privileges can reduce the risk of SQL injection by having another layer of check before being able to access the database.
4. Suppress all error messages that can be used by malicious attackers. An attacker might exploit the error messages to get access to the system, thus these should not give more detail that can be a threat to the software.
5. Web Application Firewalls: Firewalls help to filter out malicious data. A firewall also allows us to add new security features apart from the custom ones.
6. Encryption of data and other confidential information. In this case, even if a malicious attacker gets access to the data, information will still be secure.

**Q5** Because WebGoat is a vulnerable web application, it will make your system insecure while you work on this project. How can an attacker attack your system if you are using WebGoat without taking any precautions?

**Sol:** Webgoat can bypass the systems Firewall and access the system’s internal network. We should turn off the internet if Webgoat is running in the system.

**Q6**. How do you fix the above-mentioned problem? (Hint: There are multiple ways to fix this problem. You get points for mentioning the solution that does not hamper other system functionalities.)

**Sol:** Using a proxy setting on the system can solve this problem as a proxy acts an as intermediate for communications between computer and the outside world, thus making sure that our system’s identity is protected. Thus, attacker will not be able to get access to the system.

**Q7.** What is the principle of least privilege? How can you manage access privileges for using WebGoat?

**Sol:** Principle of least privilege is an idea which allows bare minimum privileges to a user, program or process to perform its functions. For example, in an online create-user webform should only use a SQL account that has only create-user privileges so that no malicious attacker can access the whole database. Managing access privileges can be done by limiting user from connecting to the external network, and restricting access instead of admin access (principle of POLP).

References: <https://digitalguardian.com/blog/what-principle-least-privilege-polp-best-practice-information-security-and-compliance>

**Q8** Pick out 6 terms that belong to attack techniques and explain each of them.

**Sol:** The following are 6 terms that belong to the attack techniques:

1. Brute-Force attack:

It is a trial and error method used to decode encrypted data that involves trying every possible combination of characters or data in order to find DES keys or passwords.

1. Password Sniffing:

This method is used by hackers to scan messages which traverse through the network to find out passwords. If we login across a network, and we come across computers on which hackers have added programs, our credentials can be accessed by the hackers.

1. Man-in-middle

Man-in-middle is a form of eavesdropping in which the attacker intercepts the communication and then the entire conversation is controlled by the hacker. If the attacker impersonates himself, then this kind of attack can be successful.

1. Phishing

Phishing is a way in which the attacker tries to replicate websites, emails or texts and when the victim clicks on these links, his/her security can be breached. This way the hacker can get access to the victim’s personal information.

1. Denial of Service (DoS)

A denial-of- service (DoS) is any type of attack where the attackers (hackers) try to make a machine or network resource unavailable by flooding a network with information.

1. Privilege Escalation

It is a type of attack in which the attacker takes advantage of programming errors or design flaws to get access to the network and its associated data and application.

**Q9** Pick out 6 terms that belong to defense techniques and explain each of them.

**Sol:**  The following are 6 terms that belong to the defense techniques:

1. Firewall:

Firewall is a network security system that monitors the traffic (i.e incoming and outgoing data) by checking if the specific traffic or link should be blocked or allowed based on some predefined security rules.

1. Antivirus

It is a software to protect the computer from viruses, worms and Trojan horses. It is used to prevent, detect and remove malicious software.

1. Sandboxing

It is a mechanism to execute untested or untrusted programs of code, like programs from unverified parties. We use this method to reduce failures or vulnerabilities from spreading.

1. Intrusion Detection System

<https://www.lifewire.com/introduction-to-intrusion-detection-systems-ids-2486799>

An intrusion detection system monitors the network traffic for suspicious activities that may indicate a network or system attack and alerts the network or system administrator.

1. Input Validation

It is a method in which all user inputs are validated before storing into the database or rather before issuing SQL queries to perform some action on the database.

1. Checksum

Checksum value indicates if the file or data has been changes or corrupted. This can be used to detect if the data has been modified and thus might pose a threat to the receiver.

**Q10** For each lesson, summarize your understanding including:

1. SQL statements used.
2. Modified SQL statement used for the SQL injection. How did it work?
3. How did WebGoat fix the vulnerability (Hint: See ‘Show Java’ tab)?
4. Screenshots

**Sol:**

1. **Blind SQL Injection:**

**SQL Query**: "SELECT \* FROM user\_data WHERE userid = " + accountNumber

**SQL Injection Query:** Consider the query -

101 AND (ascii( substr((SELECT first\_name FROM user\_data WHERE userid=15613) , 1 , 1) ) = 74 );

The answer to this query is ‘valid’ if the first character of the first\_name of userid 15613 is equal to the ASCII value of the character 'M'. In a similar manner, we can execute other queries to obtain a value for the account number field:

101 AND (ascii( substr((SELECT first\_name FROM user\_data WHERE userid=15613) , 2 , 1) ) = 111 ); [Ascii(111)=o]

For the third character: 101 AND (ascii( substr((SELECT first\_name FROM user\_data WHERE userid=15613) , 3 , 1) ) = 101 ); [Ascii(101) = e]

For the fourth character: 101 AND (ascii( substr((SELECT first\_name FROM user\_data WHERE userid=15613) , 4 , 1) ) = 115 ); [Ascii(115) = s]

For the fifth character: 101 AND (ascii( substr((SELECT first\_name FROM user\_data WHERE userid=15613) , 5 , 1) ) = 112); [Ascii(112) = p]

For the sixth character: 101 AND (ascii( substr((SELECT first\_name FROM user\_data WHERE userid=15613) , 6 , 1) ) = 104); [Ascii(104) = h]

From the above queries, we get that the name to be entered in the account number field is ‘Joesph’. When we enter this value, we get:

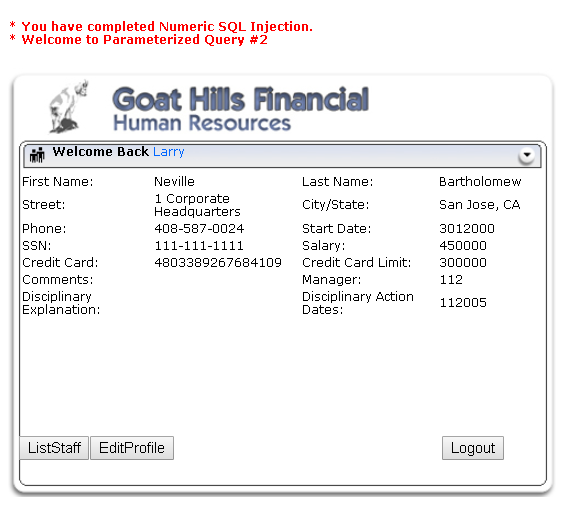


1. **Numeric Sql Injection:**

**SQL Query:** "SELECT \* FROM employee WHERE userid = " + userId + " and password = '" + password + "'";

**SQL Injection Query:** We replaced the ID 101 with '101 OR 1=1' to generate a query that always returns TRUE.

**Correctness:** "SELECT \* FROM employee WHERE userid = ? and password = ?";

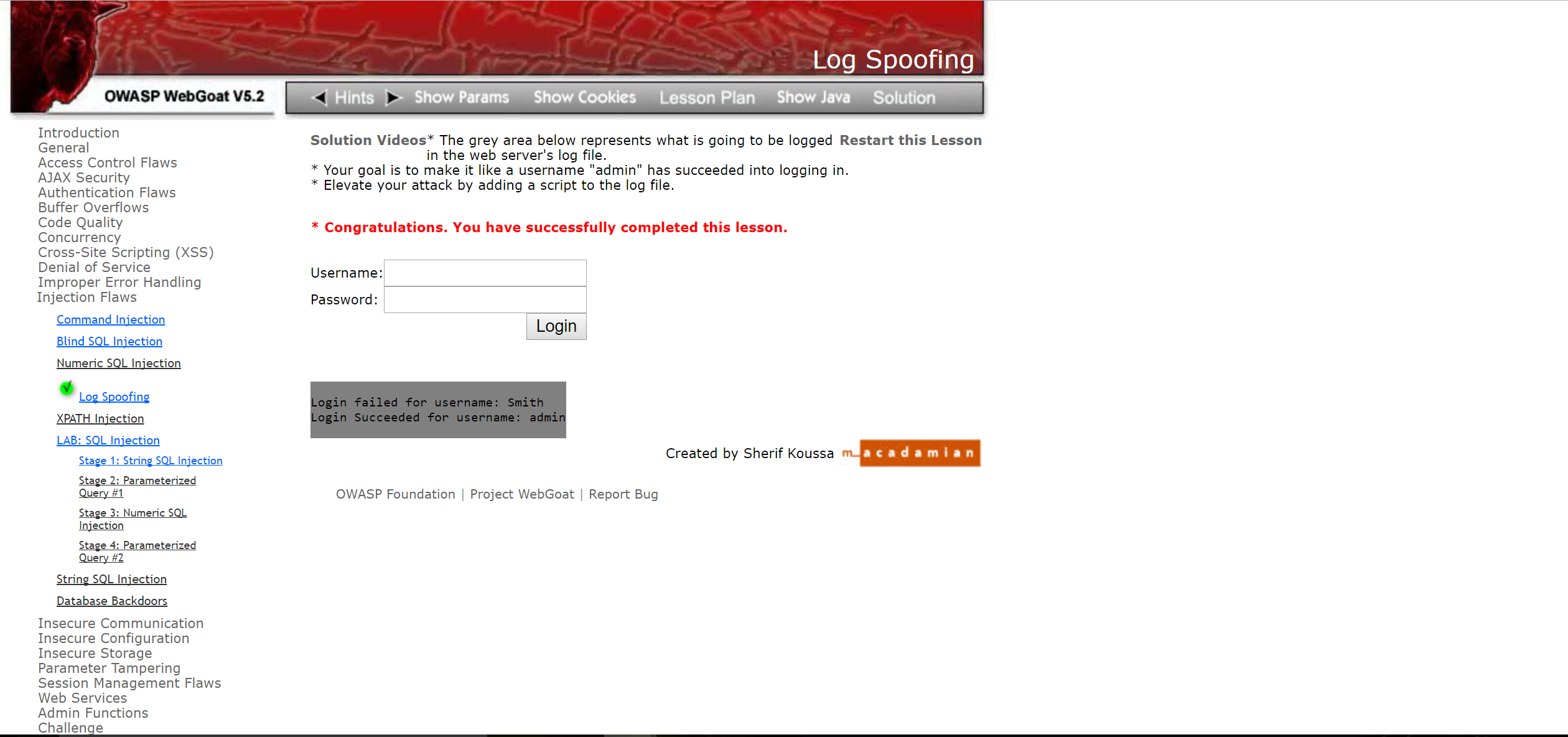


1. **Log Spoofing**

**SQL query:**

inputUsername = new String(s.getParser().getRawParameter(USERNAME, ""));  
              if (inputUsername.length() != 0)  
 { inputUsername = URLDecoder.decode(inputUsername, "UTF-8");  
               }  
                ec.addElement(new PRE(" "));  
                Table t2 = new Table(0).setCellSpacing(0).setCellPadding(0).setBorder(0);  
              TR row4 = new TR();  
              row4.addElement(new TD(new PRE("Login failed for username: " + inputUsername))).setBgColor(HtmlColor.GRAY);

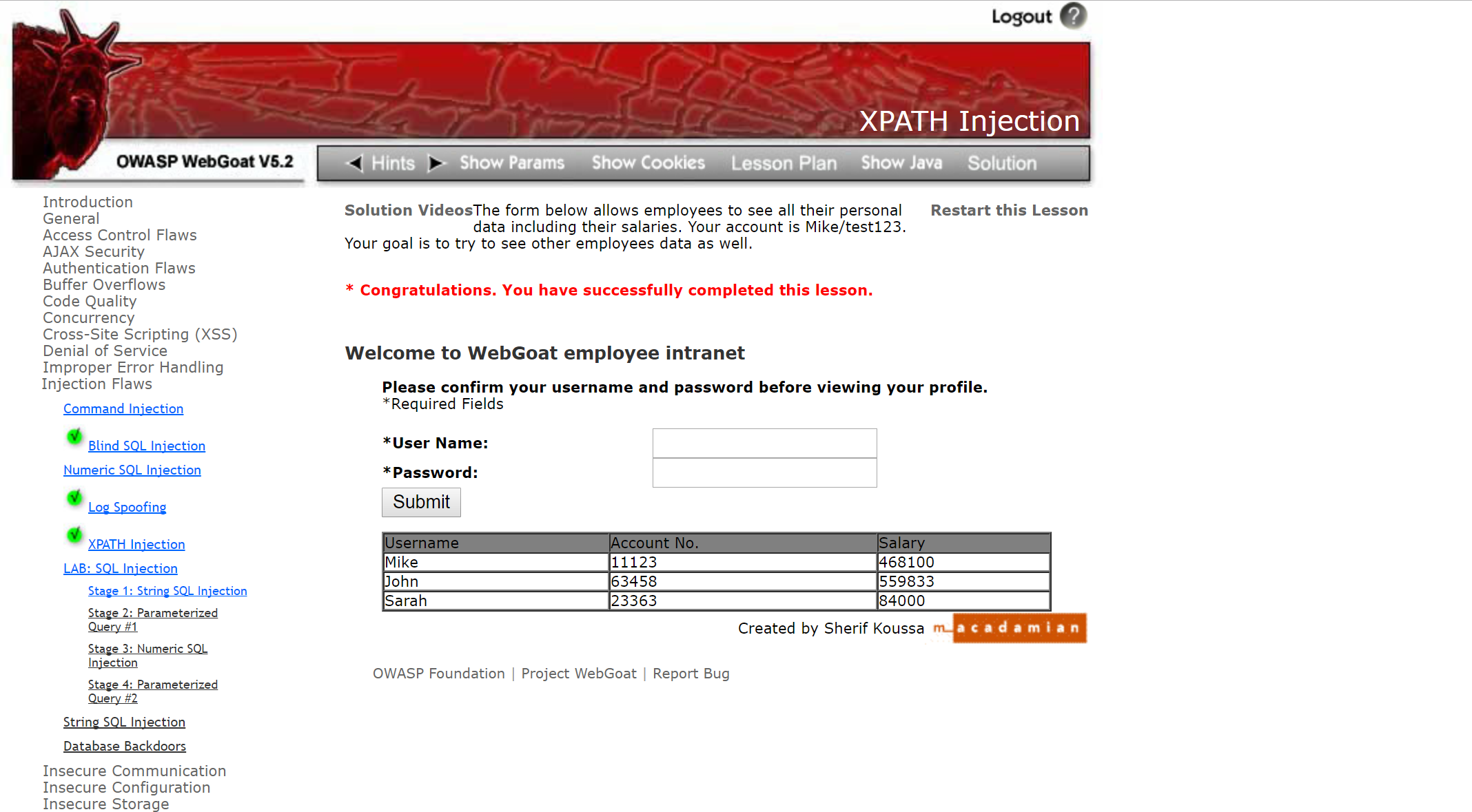
**SQL Injection Query:** Smith%0d%0aLogin Succeeded for username: admin Password: NONE



1. **Xpath Injection**

**SQL Query**: String dir = s.getContext().getRealPath("/lessons/XPATHInjection/EmployeesData.xml");  
File d = new File(dir);  
XPathFactory factory = XPathFactory.newInstance();  
XPath xPath = factory.newXPath();  
InputSource inputSource = new InputSource(new FileInputStream(d));  
String expression = "/employees/employee[loginID/text()='" + username + "' and passwd/text()='" + password + "']";  
nodes = (NodeList) xPath.evaluate(expression, inputSource, XPathConstants.NODESET);

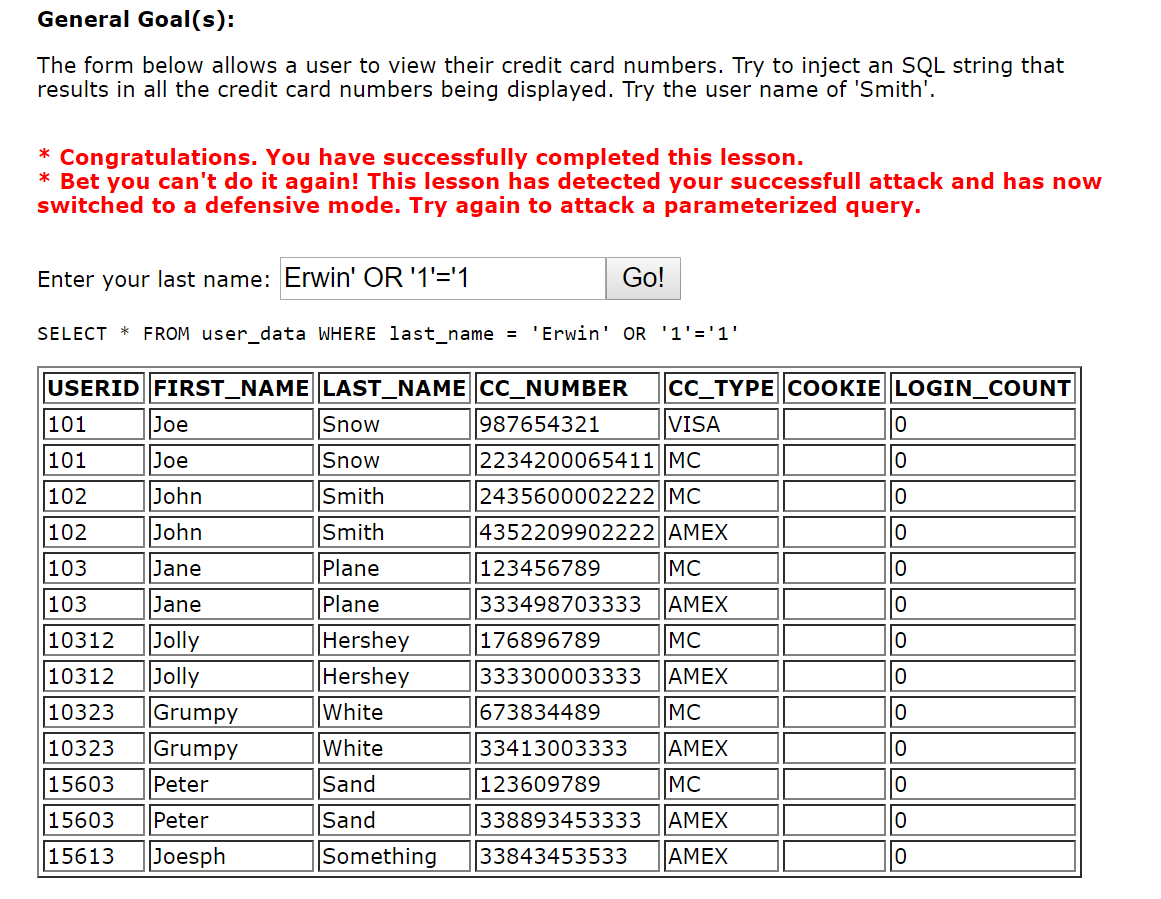
**SQL Injection Query:-** We enter Smith' or 1=1 or 'a'='a in the username field and any password.



1. **String Sql**

**Sql Query:** SELECT \* FROM user\_data WHERE last\_name = 'Your Name'

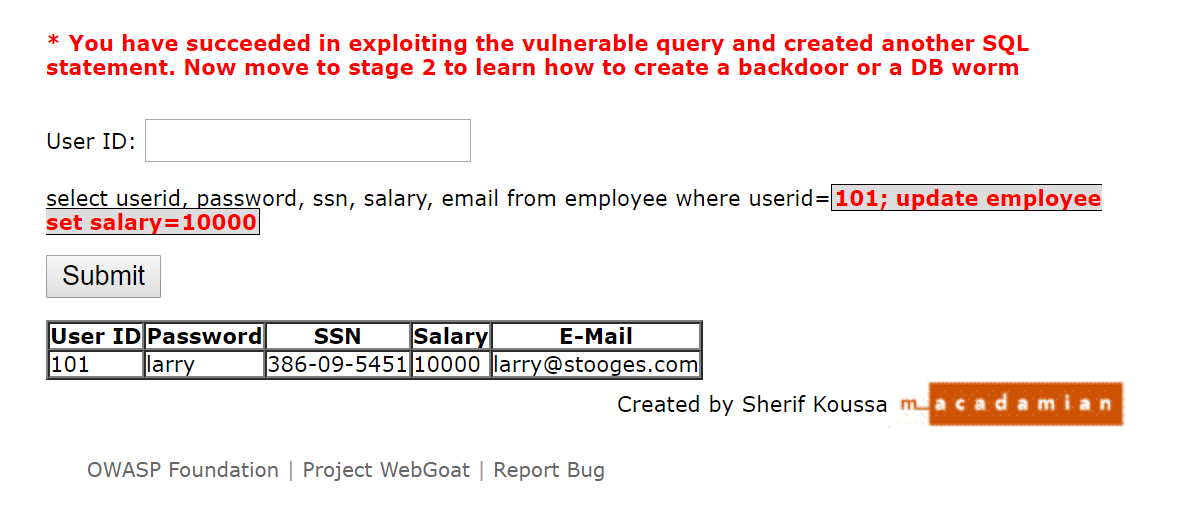
**SQL Injection Query**: We enter the last name as Erwin' OR '1'='1



1. **Backdoor Injection**

**SQL Query:** update Employees set salary=10000

**SQL Injection Query**: We enter the User ID as : CREATE TRIGGER myBackDoor BEFORE INSERT ON employee FOR EACH ROW BEGIN UPDATE employee SET email='john@hackme.com'WHERE userid = NEW.userid



SQL uses parameterized queries to fix the vulnerabilities caused by these injection attacks.