Database Security – master, 2nd year
Laboratory 5

Database applications and data security

Keywords:	
• Application's context • Dynamic SQL	• Creator's rights • Invoker's rights • SQL Injection

1. Application's context

- The **application's context** is similar to a global variable of record type in a PL/SQL package, except that its value cannot be changed by the user by simple assignment.
- Once set at login, the value of the application's context can be read throughout the user's session. The value of this variable can only be changed by a procedure call.
- At the same time, this variable is maintained by Oracle in the PGA (*Program Global Area*), thus it is private to each user session.
- As we saw in the last lab, there are ways to grant (or not) the right to execute a stored procedure to regular users.
- By default a user has the (default) USERENV context, which contains a lot of context attributes such as:
  - *IP_ADDRESS* – the IP address. In the case of the server, it will be *blank* when queries;
  - *AUTHENTICATION_TYPE* – we used it in lab 3 to find the type of authentication for the created users;
  - *CURRENT_SQL* – we used it in lab 2 to audit the SQL statements executed by the user;
  - *LANGUAGE* – the language of the user's session; ş.a.
- The database administrator can similarly define attributes for the context he creates, by running the context-associated procedure

- The steps to configure an application context by *SYS AS SYSDBA*:
  1. **Creating the context:**

     ```
     CREATE CONTEXT app_ctx USING proced_app_ctx;
     ```

  2. **Creating the procedure associated to the context** (*proced_app_ctx*)**.**

     ```
     CREATE OR REPLACE PROCEDURE proced_app_ctx IS
     …
     BEGIN
     …
     DBMS_SESSION.set_context ('APP_CTX', 'NEW_ATTRIBUTE', 'VALUE');
     …
     END;
     /
     ```

Note that the modification of contextual attributes can only be done by the procedure attached to the context. In the case of the above example, the modification of the context attributes can be done by executing the procedure proced_app_ctx. For this, SYS AS SYSDBA executes the command:

```
EXEC proced_aplicatie_ctx();
```

3. SYS AS SYSDBA creates a **logon trigger** which causes the application context for the user session to be automatically set when connecting to the database:

```
CREATE OR REPLACE TRIGGER TR_AFTER_LOGON
AFTER LOGON
ON DATABASE
BEGIN
    proced_app_ctx();
END;
/
```

4. Subsequently, it will be possible to test the value returned by the call *SYS_CONTEXT ('APP_CTX', 'NEW_ATRIBUTE')* in BEFORE INSERT / BEFORE UPDATE triggers and, depending on this value, the respective LMD operation will be allowed or not.

- **Example:** Exercise 1 proposed (and solved) at the end of the lab.

2. Dynamic SQL and its security risks

- Dynamic SQL represents those statements that allow the execution of any SQL code at *runtime*. (The concept was presented in the DBMS course.)

2.1 Dynamic Cursors

- What does a dynamic cursor look like?
- **Example:** The user ELEARN_App_Admin creates a procedure with a dynamic cursor whose aim is to retrieve the list of the students in the system.

```
--SYS AS SYSDBA executes the statements:
create user elearn_app_admin identified by 12345;
grant connect to elearn_app_admin;
alter user elearn_app_admin quota 2m on users;
grant create table to elearn_app_admin;
grant create procedure to elearn_app_admin;

--elearn_app_admin:
create table student(id number primary key,
        last_name varchar2(30), first_name varchar2(30),
        current_year number, speciality varchar2(3), group
        number);
```

```
create table subject (id number primary key,
                      title varchar2(20));
create table exam (id number primary key, subject_id number,
             exam_date date,
             constraint fk_ex2 foreign key (subject_id)
             references subject (id));
create table assessment (student_id number not null,
        exam_id number not null, grade number(4,2) default -1,
        constraint pk_ev1 primary key (student_id, exam_id),
        constraint fk_ev1 foreign key (student_id) references
        student(id), constraint fk_ex1 foreign key (exam_id)
        references exam(id));

insert into student values (1,'A','Abc',2,'Inf',231);
insert into student values(2,'B','Bbc',2,'Inf',231);
insert into subject values(1,'Algebra');
insert into exam values(1,1,sysdate-700);
insert into exam values(2,1,sysdate-300);
insert into assessment values(1,1,3);
insert into assessment values(2,1,10);
insert into assessment values(1,2,9);

--the user elearn_assistant3 tries to select data directly from
the elearn_app_admin’s tables:
SELECT EV.cod_student||EV.grade||EX.EXAM_DATE||S.TITLE
          as Info
FROM elearn_app_admin.assessment ev,
     elearn_app_admin.exam ex, elearn_app_admin.subject s
WHERE ev.exam_id=ex.id
AND ex.subject_id=s.id;
```

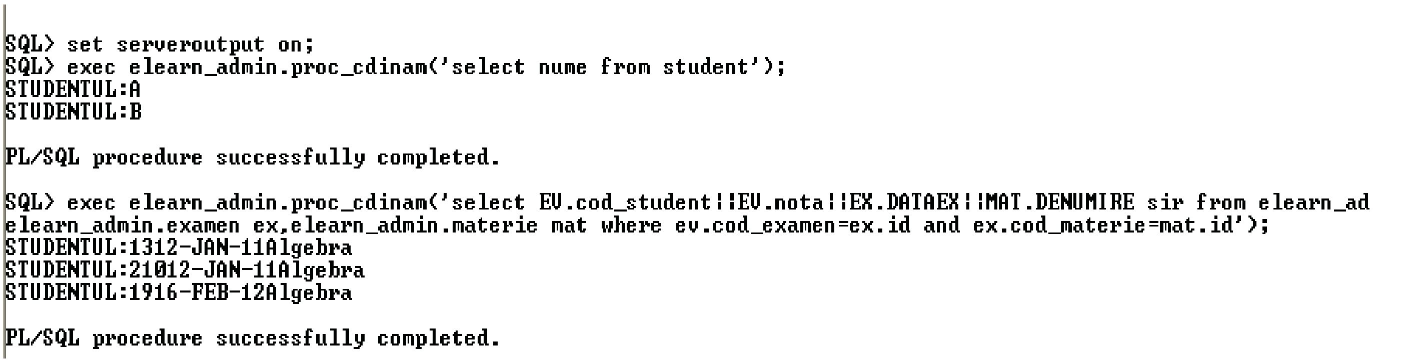
==> **Error:** table or view does not exist (actually, in this case it is due to insufficient privileges)

```
--now, elearn_app_admin creates a procedure which contains a
dynamic cursor:
CREATE OR REPLACE PROCEDURE PROC_CDYNAM(sql_query VARCHAR2) AS
  TYPE type_ref_c IS REF CURSOR;
  ref_c type_ref_c;
  v_string VARCHAR2(200);
BEGIN
  OPEN ref_c FOR sql_query;
  LOOP
     FETCH ref_c INTO v_string;
     EXIT WHEN ref_c%NOTFOUND;
```

```
        DBMS_OUTPUT.PUT_LINE('STUDENT: '||v_string);
     END LOOP;
     CLOSE ref_c;
   END;
   /

-- and grants the procedure execution privilege to the assistant
elearn_assistant3:
grant execute on proc_cdynam to elearn_assistant3;

--elearn_assistant3 retries:
SELECT EV.student_id||EV.grade||EX.EXAM_DATE||S.TITLE Info
FROM elearn_app_admin.assessment ev,
     elearn_app_admin.exam ex, elearn_app_admin.subject s
WHERE ev.exam_id=ex.id
AND ex.subject_id=s.id;
```

==> **Error:** table or view does not exist (actually, in this case it is due to insufficient privileges)

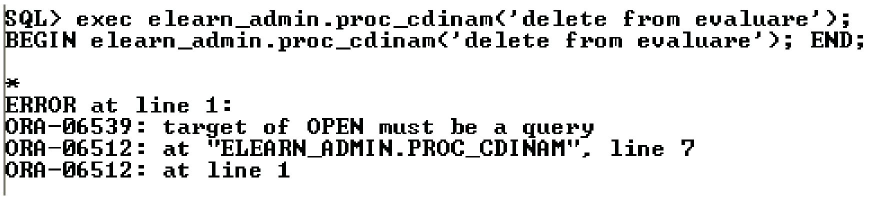
```
--He tries to use the procedure containing the dynamic cursor:
set serveroutput on;
exec elearn_app_admin.proc_cdynam('select last_name from
student');

exec elearn_app_admin.proc_cdynam('SELECT EV.student_id||
EV.grade||EX.EXAM_DATE||S.TITLE Info FROM elearn_app_admin.
assessment ev, elearn_app_admin.exam ex, elearn_app_admin.subject
s WHERE ev.exam_id=ex.id AND ex.subject_id=s.id');
```



```
SQL> set serveroutput on;
SQL> exec elearn_admin.proc_cdinam('select nume from student');
STUDENTUL:A
STUDENTUL:B

PL/SQL procedure successfully completed.

SQL> exec elearn_admin.proc_cdinam('select EV.cod_student||EV.nota||EX.DATAEX||MAT.DENUMIRE sir from elearn_ad
elearn_admin.examen ex,elearn_admin.materie mat where ev.cod_examen=ex.id and ex.cod_materie=mat.id');
STUDENTUL:1312-JAN-11Algebra
STUDENTUL:21012-JAN-11Algebra
STUDENTUL:1916-FEB-12Algebra

PL/SQL procedure successfully completed.
```

- This example shows how dynamic cursors provide read access to confidential information if they are not properly managed.
- **Note**, however, that the risk is only to disclose information, not to make changes. This is because only dynamic queries can be queried.
- If an LMD or LDD command's execution is tried, it will fail. For example:

```
exec elearn_app_admin.proc_cdynam('delete from assessment');
```



```
SQL> exec elearn_admin.proc_cdinam('delete from evaluare');
BEGIN elearn_admin.proc_cdinam('delete from evaluare'); END;

*
ERROR at line 1:
ORA-06539: target of OPEN must be a query
ORA-06512: at "ELEARN_ADMIN.PROC_CDINAM", line 7
ORA-06512: at line 1
```

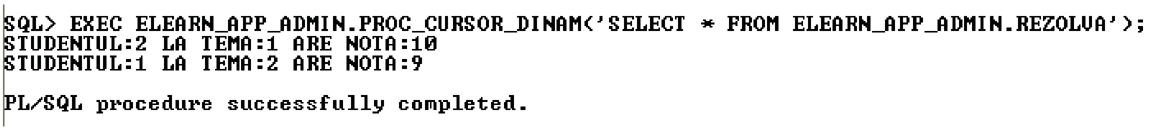
2.2 *EXECUTE IMMEDIATE*

- Another form of dynamic SQL that we might encounter is the following, based on the *EXECUTE IMMEDIATE* statement. This is one of the most vulnerable ways to execute dynamic code, as it can also allow data modifying actions.
- We use the same example as above and recreate the procedure with a new body:

```
CREATE OR REPLACE PROCEDURE
  PROC_DYNAM(sql_query VARCHAR2)
AS
  TYPE solutions_table IS TABLE OF
                        ELEARN_APP_ADMIN.SOLVES%ROWTYPE;
  v_table solutions_table;
BEGIN
  EXECUTE IMMEDIATE(sql_query) BULK COLLECT INTO v_table;
  FOR i IN 1..v_table.COUNT LOOP
     DBMS_OUTPUT.PUT_LINE('STUDENT:'||v_table(i).STUDENT_ID
     ||' HAS THE GRADE:' || NVL(v_table(i).GRADE,0) || ' AT THE
     HOMEWORK:' || v_table(i).HOMEWORK_ID);
  END LOOP;
END;
/

GRANT EXECUTE ON PROC_DYNAM TO ELEARN_professor1;
```

- First, the professor *ELEARN_professor1* executes the procedure correctly:

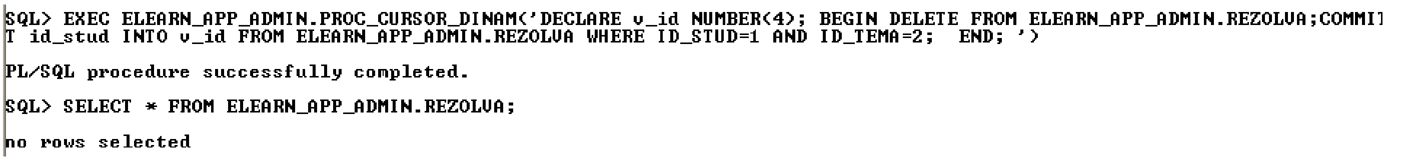
```
EXEC ELEARN_APP_ADMIN.PROC_DYNAM('SELECT * FROM
ELEARN_APP_ADMIN.ASSESSMENT);
```



```
SQL> EXEC ELEARN_APP_ADMIN.PROC_CURSOR_DINAM('SELECT * FROM ELEARN_APP_ADMIN.REZOLVA');
STUDENTUL:2 LA TEMA:1 ARE NOTA:10
STUDENTUL:1 LA TEMA:2 ARE NOTA:9

PL/SQL procedure successfully completed.
```

- Then, the professor *ELEARN_professor1* executes the modified procedure the second time and manages to delete all the records in the table *SOLVES*:

```
EXEC ELEARN_APP_ADMIN.PROC_DYNAM('DECLARE v_id NUMBER(4); BEGIN
DELETE FROM ELEARN_APP_ADMIN.SOLVES;COMMIT; SELECT stud_id INTO
v_id FROM ELEARN_APP_ADMIN.SOLVES WHERE STUD_ID=1 AND
HOMEWORK_ID=2; END; ');
```

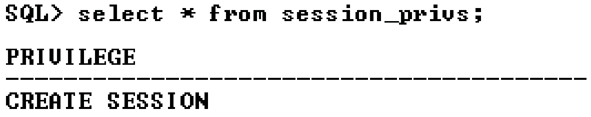


```
SQL> EXEC ELEARN_APP_ADMIN.PROC_CURSOR_DINAM('DECLARE v_id NUMBER(4); BEGIN DELETE FROM ELEARN_APP_ADMIN.REZOLVA;COMMI
T id_stud INTO v_id FROM ELEARN_APP_ADMIN.REZOLVA WHERE ID_STUD=1 AND ID_TEMA=2;  END; ')

PL/SQL procedure successfully completed.

SQL> SELECT * FROM ELEARN_APP_ADMIN.REZOLVA;

no rows selected
```

- **What did he actually do?** He included an entire PL/SQL block as the argument for the command *EXECUTE IMMEDIATE*.
- **How did this succeed** in terms of privileges?

- We verify what privileges user *ELEARN_professor1* has in the current session:

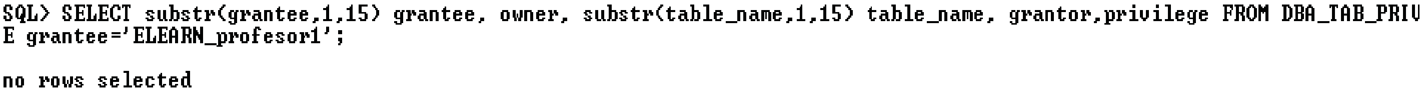
```
select * from session_privs;
```



```
SQL> select * from session_privs;

PRIVILEGE
----------------------------------------
CREATE SESSION
```

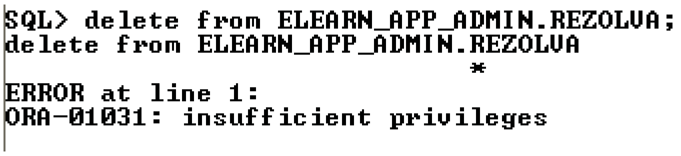
- We can also query as *SYS AS SYSDBA* to check. Indeed, we will notice that the professor does not have delete privileges on the table:

```
SELECT substr(grantee,1,15) grantee, owner,
       substr(table_name,1,15) table_name, grantor,privilege
FROM DBA_TAB_PRIVS
WHERE grantee='ELEARN_professor1';
```



```
SQL> SELECT substr(grantee,1,15) grantee, owner, substr(table_name,1,15) table_name, grantor,privilege FROM DBA_TAB_PRIV
E grantee='ELEARN_profesor1';

no rows selected
```

- The third way to convince ourselves is for the teacher to try to execute the delete command directly from the prompt:



```
SQL> delete from ELEARN_APP_ADMIN.REZOLVA;
delete from ELEARN_APP_ADMIN.REZOLVA
                            *
ERROR at line 1:
ORA-01031: insufficient privileges
```

- **Note** that the invoker *ELEARN_professor1* would not have been allowed to delete from the table *ELEARN_APP_ADMIN.SOLVES*. **However**, he succeeded it because he executed the procedure **in the context of privileges of the procedure's creator**, i.e. *ELEARN_APP_ADMIN*.

- We remember the following table from the last lab.

User X creates a view object (trigger, procedure -)

	In X's own schema			In another user (Y)'s schema		
	Accesses objects in X's own schema	Accesses objects in the Y's schema (select Y.D, insert Y.D)		Accesses objects in X's own schema	Accesses objects in the Y's schema (select Y.D, insert Y.D)	

What privileges are needed by X?	CREATE VIEW	CREATE VIEW
		SELECT ON Y.D
		INSERT ON Y.D
			SELECT ON Y.D WITH GRANT OPTION
			INSERT ON Y.D WITH GRANT OPTION
				CREATE ANY VIEW	CREATE ANY VIEW
					SELECT ON Y.D
					INSERT ON Y.D
						SELECT ON Y.D WITH GRANT OPTION
						INSERT ON Y.D WITH GRANT OPTION

What privileges are needed by a caller Z?	SELECT ON view
	INSERT ON view
		SELECT ON view
		INSERT ON view
		SELECT ON Y.D
		INSERT ON Y.D
			SELECT ON view
			INSERT ON view
				SELECT ON view
				INSERT ON view
					SELECT ON view
					INSERT ON view
					SELECT ON Y.D
					INSERT ON Y.D
						SELECT ON view
						INSERT ON view

- We are in the case where user *X* (*ELEARN_APP_ADMIN*) has created a procedure in his own schema, and this one accesses objects from his own schema (the table *ELEARN_APP_ADMIN.SOLVES*).
- Therefore, the caller *Z* (*ELEARN_professor1*) only needs privileges on the procedure (*ELEARN_APP_ADMIN.PROC_DYNAM*) to be able to execute with it whatever is allowed to the creator of the procedure.

How do we protect the code from such attacks?

- **The first way** is to **add the *AUTHID CURRENT_USER* clause** to the procedure header. Thus, only the caller's context of privileges will be used at runtime.
  This technique is called the "Invoker Rights' Model".

```
CREATE OR REPLACE PROCEDURE
  PROC_DYNAM(sql_query VARCHAR2) AUTHID CURRENT_USER
AS
  TYPE solutions_table IS TABLE OF
                               ELEARN_APP_ADMIN.SOLVES%ROWTYPE;
  v_table solutions_table;
BEGIN
  EXECUTE IMMEDIATE(sql_query) BULK COLLECT INTO v_table;
  FOR i IN 1..v_table.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE('STUDENT:'||v_table(i).STUDENT_ID
        ||' HAS THE GRADE:' || NVL(v_table(i).GRADE,0) || ' AT
        THE HOMEWORK:' || v_table(i).HOMEWORK_ID);
  END LOOP;
END;
/

GRANT EXECUTE ON PROC_DYNAM TO ELEARN_professor1;
```

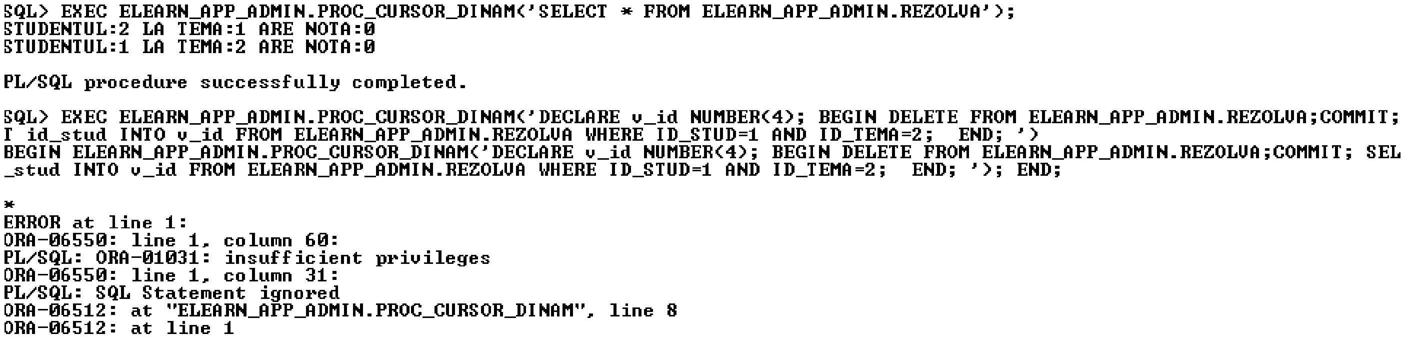
- We test the effect. First *ELEARN_APP_ADMIN* restores the data of the *SOLVES* table with the following *insert* commands:

```
INSERT INTO SOLVES (HOMEWORK_ID,STUDENT_ID,UPLOAD_DATE)
VALUES(1,2,SYSDATE-3);
INSERT INTO SOLVES (HOMEWORK_ID,STUDENT_ID,UPLOAD_DATE)
VALUES(2,1,SYSDATE-7);
COMMIT;
```

- The user *ELEARN_professor1* tries again to execute the two calls, one with a *SELECT* statement and one with an included PL/SQL block:

```
EXEC ELEARN_APP_ADMIN.PROC_DYNAM('SELECT * FROM
ELEARN_APP_ADMIN.SOLVES');
```

```
EXEC ELEARN_APP_ADMIN.PROC_DYNAM('DECLARE v_id NUMBER(4); BEGIN
DELETE FROM ELEARN_APP_ADMIN.SOLVES;COMMIT; SELECT student_id
INTO v_id FROM ELEARN_APP_ADMIN.SOLVES WHERE STUDENT_ID=1 AND
HOMEWORK_ID =2; END; ');
```



```
SQL> EXEC ELEARN_APP_ADMIN.PROC_CURSOR_DINAM('SELECT * FROM ELEARN_APP_ADMIN.REZOLVA');
STUDENTUL:2 LA TEMA:1 ARE NOTA:0
STUDENTUL:1 LA TEMA:2 ARE NOTA:0

PL/SQL procedure successfully completed.

SQL> EXEC ELEARN_APP_ADMIN.PROC_CURSOR_DINAM('DECLARE v_id NUMBER(4); BEGIN DELETE FROM ELEARN_APP_ADMIN.REZOLVA;COMMIT;
I id_stud INTO v_id FROM ELEARN_APP_ADMIN.REZOLVA WHERE ID_STUD=1 AND ID_TEMA=2;  END; ')
BEGIN ELEARN_APP_ADMIN.PROC_CURSOR_DINAM('DECLARE v_id NUMBER(4); BEGIN DELETE FROM ELEARN_APP_ADMIN.REZOLVA;COMMIT; SEL
_stud INTO v_id FROM ELEARN_APP_ADMIN.REZOLVA WHERE ID_STUD=1 AND ID_TEMA=2;  END; '); END;

*
ERROR at line 1:
ORA-06550: line 1, column 60:
PL/SQL: ORA-01031: insufficient privileges
ORA-06550: line 1, column 31:
PL/SQL: SQL Statement ignored
ORA-06512: at "ELEARN_APP_ADMIN.PROC_CURSOR_DINAM", line 8
ORA-06512: at line 1
```

- We notice that the second call (the one with the malicious PL/SQL code) failed, because the procedure was executed with the invoker's rights, who did not have the privilege to delete on the table *ELEARN_APP_ADMIN.SOLVES.*

- **The second way** to protect from vulnerabilities in dynamic SQL code is by using regular expressions. Thus, we will validate the query entered by the user before executing it.
- For our previous example, the procedure is rewritten as follows, to test that even if he had DML privileges on the table, the user will use the procedure only for queries:

```
CREATE OR REPLACE PROCEDURE
  PROC_DYNAM(sql_query VARCHAR2) AUTHID CURRENT_USER
AS
  TYPE solutions_table IS TABLE OF
                       ELEARN_APP_ADMIN.SOLVES%ROWTYPE;
  v_table solutions_table;
  is_ok NUMBER(1) :=0;
BEGIN
  IF REGEXP_LIKE(sql_query,'SELECT [A-Za-z0-9*]+ [^;]') THEN
       is_ok:=1;
  END IF;
  IF is_ok = 1 THEN BEGIN
    EXECUTE IMMEDIATE(sql_query) BULK COLLECT INTO v_table;
    FOR i IN 1..v_table.COUNT LOOP
       DBMS_OUTPUT.PUT_LINE('STUDENT:'||v_table(i).STUDENT_ID
       ||' HAS THE GRADE:' || NVL(v_table(i).GRADE,0) || ' AT
       THE HOMEWORK:' || v_table(i).HOMEWORK_ID);
    END LOOP;
  END;
  ELSE
    DBMS_OUTPUT.PUT_LINE('The command contains suspicious
    malicious code. Only queries are allowed');
  END IF;
END;
/
```

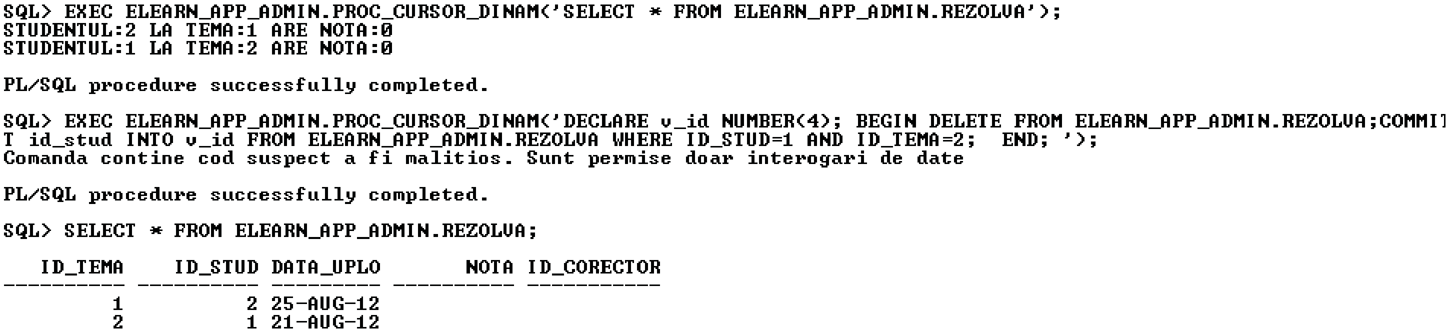
```
GRANT EXECUTE ON PROC_DYNAM TO ELEARN_professor1;
```

- Suppose that strictly for this example the teacher is also granted the *DELETE* privilege on the *SOLVES* table.

```
GRANT DELETE ON ELEARN_APP_ADMIN.SOLVES TO ELEARN_professor1;
```

- The user *ELEARN_professor1* tries again to execute the two calls, one with a *SELECT* statement and one with an included PL/SQL block:

```
EXEC ELEARN_APP_ADMIN.PROC_DYNAM('SELECT * FROM
ELEARN_APP_ADMIN.SOLVES');

EXEC ELEARN_APP_ADMIN.PROC_DYNAM('DECLARE v_id NUMBER(4); BEGIN
DELETE FROM ELEARN_APP_ADMIN.SOLVES;COMMIT; SELECT student_id
INTO v_id FROM ELEARN_APP_ADMIN.SOLVES WHERE STUDENT_ID=1 AND
HOMEWORK_ID =2; END; ');
```



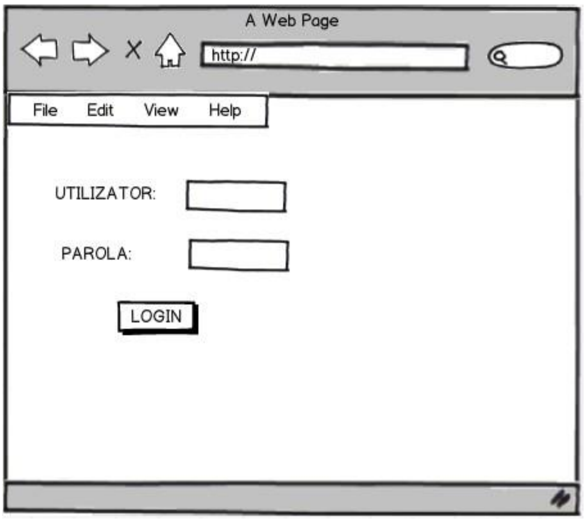
```
SQL> EXEC ELEARN_APP_ADMIN.PROC_CURSOR_DINAM('SELECT * FROM ELEARN_APP_ADMIN.REZOLVA');
STUDENTUL:2 LA TEMA:1 ARE NOTA:0
STUDENTUL:1 LA TEMA:2 ARE NOTA:0

PL/SQL procedure successfully completed.

SQL> EXEC ELEARN_APP_ADMIN.PROC_CURSOR_DINAM('DECLARE v_id NUMBER(4); BEGIN DELETE FROM ELEARN_APP_ADMIN.REZOLVA;COMMIT
T id_stud INTO v_id FROM ELEARN_APP_ADMIN.REZOLVA WHERE ID_STUD=1 AND ID_TEMA=2;  END; ');
Comanda contine cod suspect a fi malitios. Sunt permise doar interogari de date

PL/SQL procedure successfully completed.

SQL> SELECT * FROM ELEARN_APP_ADMIN.REZOLVA;

   ID_TEMA    ID_STUD DATA_UPLO       NOTA ID_CORECTOR
---------- ---------- --------- ---------- -----------
         1          2 25-AUG-12
         2          1 21-AUG-12
```

- We notice from the screenshot the effect of checking with a regular expression. The user’s attempt to execute malicious code failed.
- In order to restore the initial privileges, we revoke the professor's right to delete records on the SOLVES table, granted strictly for this didactic example:

```
REVOKE DELETE ON ELEARN_APP_ADMIN.SOLVES FROM ELEARN_professor1;
```

3. SQL Injection

- SQL Injection is a type of attack that involves inserting code snippets with valid SQL syntax into an existing SQL query, with the potential to produce destructive effects.
- A typical example is a query that uses values received at runtime in the *WHERE* clause. Often, the target of SQL injection attacks is the login forms that are unprotected face to such attacks.

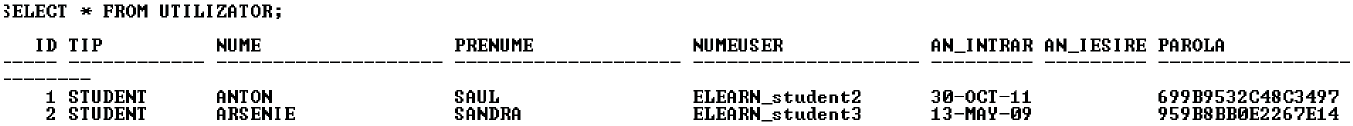


- In order to exemplify, the user *ELEARN_APP_ADMIN adds a column to the USER_ table that will store passwords in encrypted form.*

```
ALTER TABLE USER_ ADD PASSWORD VARCHAR2(32);
```

- For testing, we populate this field with *dummy data*, reusing the knowledge from lab 1 (about encryption):

```
--SYS AS SYSDBA must give the privilege to make full use of the
encryption package for the user ELEARN_APP_ADMIN:
GRANT ALL ON DBMS_CRYPTO TO ELEARN_APP_ADMIN;

--The function ENCRYPTION1 encrypts a string received as a
parameter using the algorithm DES, the key ’12345678’, padding
with zeroes and the ECB chaining method.
CREATE OR REPLACE FUNCTION encryption1(plain_text IN VARCHAR2)
       RETURN VARCHAR2 AS
  raw_string RAW(20);
  raw_password RAW(20);
  result RAW(20);
  password VARCHAR2(20) := '12345678';
  operating_mode NUMBER;
  encrypted_text VARCHAR2(32);
BEGIN
  raw_string:=utl_i18n.string_to_raw(plain_text,'AL32UTF8');
  raw_password :=utl_i18n.string_to_raw(password,'AL32UTF8');
  operating_mode := DBMS_CRYPTO.ENCRYPT_DES +
               DBMS_CRYPTO.PAD_ZERO + DBMS_CRYPTO.CHAIN_ECB;
  result := DBMS_CRYPTO.ENCRYPT(raw_string,
             operating_mode,raw_password);
  --dbms_output.put_line(result);
  encrypted_text := RAWTOHEX(result);
  RETURN encrypted_text;

END;
/
--Next, the application's administrator (ELEARN_APP_ADMIN)
updates the passwords in the USER_ table:
UPDATE USER_
SET PASSWORD=encryption1('Password1')
WHERE ID=1;
UPDATE USER_
SET PASSWORD=encryption1('Password2')
WHERE ID=2;

SET LINESIZE 200
SELECT * FROM UTILIZATOR;
```



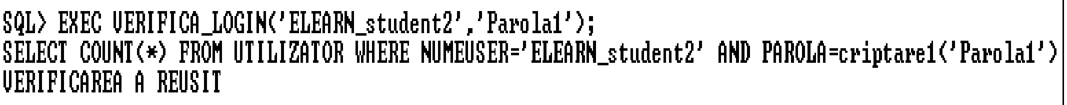
```
3ELECT * FROM UTILIZATOR;

   ID TIP          NUME                 PRENUME              NUMEUSER             AN_INTRAR AN_IESIRE PAROLA
----- ------------ -------------------- -------------------- -------------------- --------- --------- ----------------
--------
    1 STUDENT      ANTON                SAUL                 ELEARN_student2      30-OCT-11           699B9532C48C3497
    2 STUDENT      ARSENIE              SANDRA               ELEARN_student3      13-MAY-09           959B8BB0E2267E14
```

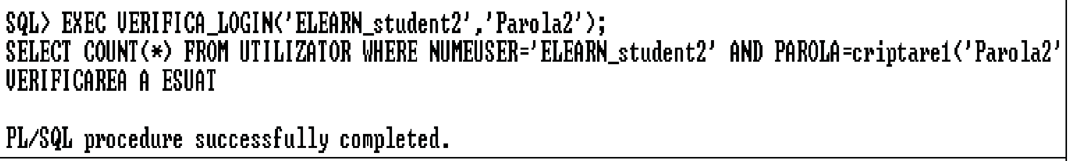
- We can continue the discussion about SQL Injection. Suppose that the "LOGIN" button on the form corresponds to a stored procedure created by the e-learning application administrator, which aims to verify the match between the username and password entered through the form

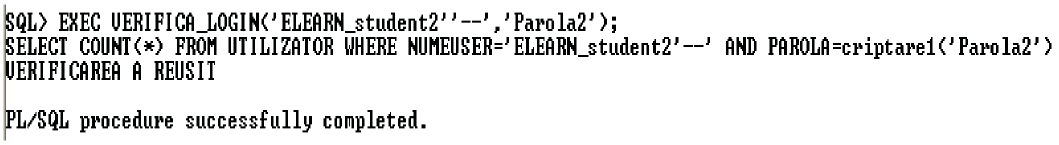
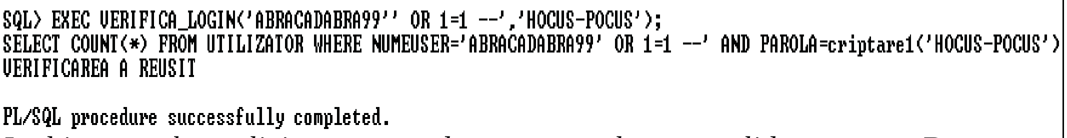
```
CREATE OR REPLACE PROCEDURE
       VERIFY_LOGIN (P_USERNAME VARCHAR2,P_PASSWORD VARCHAR2) AS
  v_ok NUMBER(2) :=-1;
BEGIN
  EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM USER_ WHERE
     USERNAME='''||P_USERNAME||''' AND
     PASSWORD=encryption1('''||P_PASSWORD||''')' INTO v_ok;
  DBMS_OUTPUT.PUT_LINE('SELECT COUNT(*) FROM USER_ WHERE
     USERNAME='''||P_USERNAME||''' AND
     PAROLA=encryption1('''||P_PASSWORD||''')');
  IF v_ok=0 THEN
       DBMS_OUTPUT.PUT_LINE('VERIFICATION FAILED');
  ELSE
       DBMS_OUTPUT.PUT_LINE('VERIFICATION SUCCESSFUL');
  END IF;
END;
/
```

- **What kind of attacks** could happen by calling this procedure with malicious parameters?

Value of parameter **P_USERNAME**	Value of parameter **P_PASSWORD**	**Result**

'ELEARN_student2'	'Password1'	**EXEC VERIFY_LOGIN('ELEARN_student2','Parola1');**
		
		SQL> EXEC VERIFICA_LOGIN('ELEARN_student2','Parola1');
		SELECT COUNT(*) FROM UTILIZATOR WHERE NUMEUSER='ELEARN_student2' AND PAROLA=criptare1('Parola1')
		VERIFICAREA A REUSIT

'ELEARN_student2'	'Password2'	**EXEC VERIFY_LOGIN('ELEARN_student2','Parola2');**
		
		SQL> EXEC VERIFICA_LOGIN('ELEARN_student2','Parola2');
		SELECT COUNT(*) FROM UTILIZATOR WHERE NUMEUSER='ELEARN_student2' AND PAROLA=criptare1('Parola2'
		VERIFICAREA A ESUAT
		PL/SQL procedure successfully completed.

'ELEARN_student2''--'	'Password2'	**EXEC VERIFY_LOGIN('ELEARN_student2''--','Parola2');**  SQL> EXEC VERIFICA_LOGIN('ELEARN_student2''--','Parola2'); SELECT COUNT(*) FROM UTILIZATOR WHERE NUMEUSER='ELEARN_student2'--' AND PAROLA=criptare1('Parola2') VERIFICAREA A REUSIT PL/SQL procedure successfully completed. In this case, *the malicious person knows a valid username, but does not know the password*. By adding apostrophes and the 2 dashes he inhibits (as a comment) the password part of the SELECT query.
'ABRACADABRA99'' OR 1=1 --'	'HOCUS-POCUS'	**EXEC VERIFY_LOGIN('ABRACADABRA99'' OR 1=1 --','HOCUS-POCUS');**  SQL> EXEC VERIFICA_LOGIN('ABRACADABRA99'' OR 1=1 --','HOCUS-POCUS'); SELECT COUNT(*) FROM UTILIZATOR WHERE NUMEUSER='ABRACADABRA99' OR 1=1 --' AND PAROLA=criptare1('HOCUS-POCUS') VERIFICAREA A REUSIT PL/SQL procedure successfully completed. In this case, *the malicious person does not even know a valid username*. By adding apostrophes and the 2 dashes he inhibits (as a comment) the password part of the SELECT query. Moreover, the OR clause also allows in this case the cancellation of the test of the username's existence in the table.

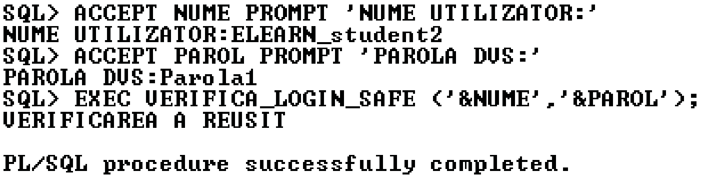
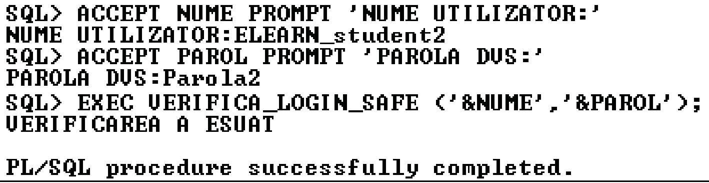
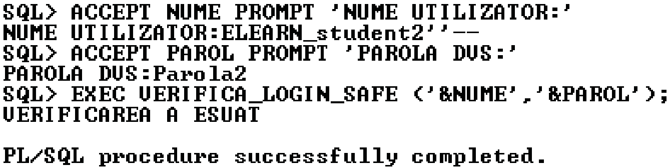
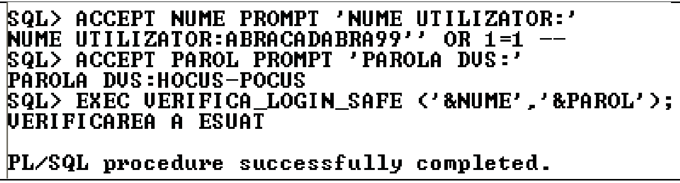
How do we protect from such attacks?

- By replacing the concatenation in the string that represents the SQL command with bind variables
- We present the protection options in the case of the procedure which verifies the correspondence username - password in the form.
- **Option 1** of rewrite for protection:

```
CREATE OR REPLACE PROCEDURE
   VERIFY_LOGIN_SAFE (P_USERNAME VARCHAR2,P_PASSWORD VARCHAR2) AS
  v_ok NUMBER(2) :=-1;
BEGIN
  SELECT COUNT(*) INTO v_ok FROM USER_ WHERE USERNAME=P_USERNAME
```

```
AND PASSWORD=encryption1(P_PASSWORD);
   IF v_ok=0 THEN
          DBMS_OUTPUT.PUT_LINE('VERIFICATION FAILED');
   ELSE
          DBMS_OUTPUT.PUT_LINE('VERIFICATION SUCCESSFUL');
   END IF;
END;
/

--On call, the parameters are provided as follows:
ACCEPT USERNAME PROMPT 'USER NAME:'
ACCEPT PASSWORD PROMPT 'PASSWORD:'
EXEC VERIFY_LOGIN_SAFE ('&USERNAME','&PASSWORD');
```

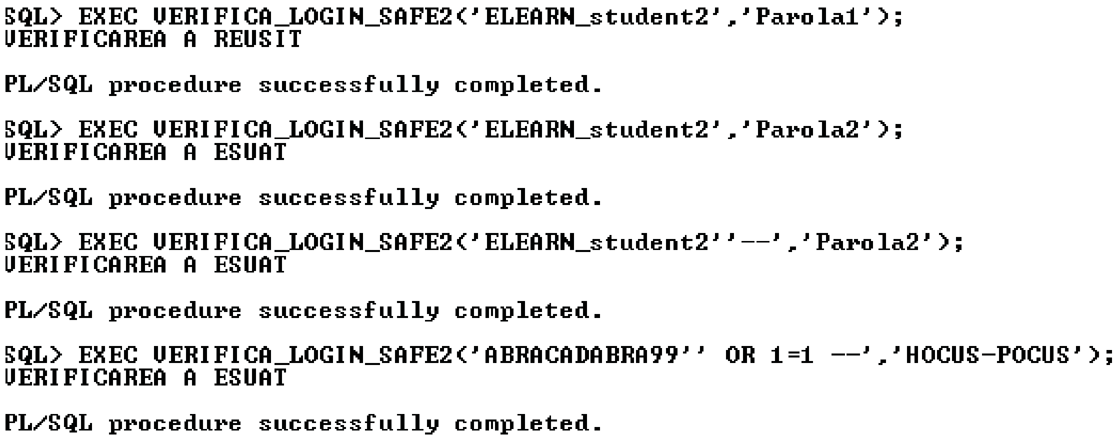
- We re-test to make sure that malicious attacks are no longer successful:

Value of parameter **P_USERNAME**	Value of parameter **P_PASSWORD**	**Result**
'ELEARN_student2'	'Password1'	 SQL> ACCEPT NUME PROMPT 'NUME UTILIZATOR:' NUME UTILIZATOR:ELEARN_student2 SQL> ACCEPT PAROL PROMPT 'PAROLA DVS:' PAROLA DVS:Parola1 SQL> EXEC VERIFICA_LOGIN_SAFE ('&NUME','&PAROL'); VERIFICAREA A REUSIT PL/SQL procedure successfully completed.
'ELEARN_student2'	'Password2'	 SQL> ACCEPT NUME PROMPT 'NUME UTILIZATOR:' NUME UTILIZATOR:ELEARN_student2 SQL> ACCEPT PAROL PROMPT 'PAROLA DVS:' PAROLA DVS:Parola2 SQL> EXEC VERIFICA_LOGIN_SAFE ('&NUME','&PAROL'); VERIFICAREA A ESUAT PL/SQL procedure successfully completed.
'ELEARN_student2''--'	'Password2'	 SQL> ACCEPT NUME PROMPT 'NUME UTILIZATOR:' NUME UTILIZATOR:ELEARN_student2''-- SQL> ACCEPT PAROL PROMPT 'PAROLA DVS:' PAROLA DVS:Parola2 SQL> EXEC VERIFICA_LOGIN_SAFE ('&NUME','&PAROL'); VERIFICAREA A ESUAT PL/SQL procedure successfully completed.
'ABRACADABRA99'' **OR 1=1 --**'	'HOCUS-POCUS'	 SQL> ACCEPT NUME PROMPT 'NUME UTILIZATOR:' NUME UTILIZATOR:ABRACADABRA99'' OR 1=1 -- SQL> ACCEPT PAROL PROMPT 'PAROLA DVS:' PAROLA DVS:HOCUS-POCUS SQL> EXEC VERIFICA_LOGIN_SAFE ('&NUME','&PAROL'); VERIFICAREA A ESUAT PL/SQL procedure successfully completed.

- **Option 2** of rewrite for protection:

```
CREATE OR REPLACE PROCEDURE
     VERIFY_LOGIN_SAFE2 (P_USERNAME VARCHAR2,P_PASSWORD VARCHAR2)
  AS
  v_ok NUMBER(2) :=-1;
BEGIN
  EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM USER_ WHERE
    USERNAME=:name AND PASSWORD=encryption1(:passwd)' INTO v_ok
    USING P_USERNAME,P_PASSWORD;
  IF v_ok=0 THEN
       DBMS_OUTPUT.PUT_LINE('VERIFICATION FAILED');
  ELSE
       DBMS_OUTPUT.PUT_LINE('VERIFICATION SUCCESSFUL');
  END IF;
END;
/

-- On call, the parameters are provided as follows:
EXEC VERIFY_LOGIN_SAFE2('ELEARN_student2','Password1');
EXEC VERIFY_LOGIN_SAFE2('ELEARN_student2','Password2');
EXEC VERIFY_LOGIN_SAFE2('ELEARN_student2''--','Password2');
EXEC VERIFY_LOGIN_SAFE2('ABRACADABRA99'' OR 1=1 --','HOCUS-
POCUS');
```

- We re-test to make sure that malicious attacks are no longer successful:



```
SQL> EXEC VERIFICA_LOGIN_SAFE2(’ELEARN_student2’,’Parola1’);
VERIFICAREA A REUSIT

PL/SQL procedure successfully completed.

SQL> EXEC VERIFICA_LOGIN_SAFE2(’ELEARN_student2’,’Parola2’);
VERIFICAREA A ESUAT

PL/SQL procedure successfully completed.

SQL> EXEC VERIFICA_LOGIN_SAFE2(’ELEARN_student2’’--’,’Parola2’);
VERIFICAREA A ESUAT

PL/SQL procedure successfully completed.

SQL> EXEC VERIFICA_LOGIN_SAFE2(’ABRACADABRA99’’ OR 1=1 --’,’HOCUS-POCUS’);
VERIFICAREA A ESUAT

PL/SQL procedure successfully completed.
```

- At the end, we delete the *password* column, in order not to influence the subsequent solutions:

```
ALTER TABLE USER_ DROP COLUMN PASSWORD;
```

4. Exercises

1. **Create an application context that establishes, as a security measure, the possibility that teachers evaluate the homework submitted by students only in the working hours from the faculty, 8.00-20.00.**

2. **Find all the security breaches in the following procedure, which was intended to display the homework of all students submitted in a year or month or on an exact date provided as an input parameter:**

```
CREATE OR REPLACE PROCEDURE FIND_DANGERS(
                                 P_UPLOAD_DATE VARCHAR2) AS
   TYPE t_table IS TABLE OF ELEARN_APP_ADMIN.SOLVES%ROWTYPE;
   v_table t_table;
BEGIN
   EXECUTE IMMEDIATE 'SELECT * FROM SOLVES WHERE
      TO_CHAR(UPLOAD_DATE,''DD-MM-YYYY HH24:MI:SS'')
      LIKE''%'||P_UPLOAD_DATE ||'%'''
   BULK COLLECT INTO v_table;
   FOR i IN 1..v_table.COUNT LOOP
     DBMS_OUTPUT.PUT_LINE('STUDENT:' || v_table(i).STUDENT_ID
     || ' HAS THE GRADE:' ||NVL(v_table(i).GRADE,-1)|| 'AT THE
     HOMEWORK:' || v_table(i).HOMEWORK_ID || 'UPLOADED ON: '
     || v_table(i).UPLOAD_DATE);
   END LOOP;
END;
/
```

Suggestions for possible malicious attacks:
1. Get also all the information about the users of the application, since they are in the system.
2. Obtain also additional information about the status of students (whom of them are in the process of resuming their studies).