AUTOMATIC DETECTION OF VULNERABILITY IN WRAPPED PACKAGES IN ORACLE

Presenters:

Yaron Gur-Arieh
Nikita Zubrilov
Ilya Kolchinsky
The Problem
Our Project Deals With

☐ “One of the key threats to the security of Oracle database servers are bugs in the default PL/SQL packages, triggers, and types that are shipped with the database.” taken from ‘The Oracle Hackers Handbook’

☐ “If public gets DBA Role then it will be a disaster”
  - Shay Gur-Arye, DBA of over 25 years

☐ Vulnerabilities are “built in” by default

☐ Even when vulnerabilities are known, they cannot be easily detected and neutralized properly
It Gets Worse

- Code can be wrapped
- An application that automates the tax reform - made by Matrix co.
  - We want to use this application
  - Matrix co. wants to sell this application to us
- A product we want to use, but we cannot verify if it is safe for us to use it
What Do We Want

- To be certain that our databases are secure, even when using wrapped code
- To have a tool that can help us be sure that our code is safe to be used
- To stay safe even when new exploits are discovered
- Provide the industry with an automated way to test their code, granting them a tool to help safeguard themselves and others
What Has Been Done
In The Past

- Two main unwrapping techniques
  - Pete Finnigan
    - Deduce the source code from DIANA code
      (an intermediate language for ADA)
    - Oracle version does not matter
  - David Litchfield
    - Oracle 10g version needs a special treatment
    - Build a un-wrapper for the wrapped Oracle packages – it is easy, and then deal with plain PL/SQL code
Detection Approaches

- **Potential vulnerabilities** (SQL injections in Oracle)
  - SQL manipulation (adding OR, AND, MINUS, UNION, …)
  - Code injection (adding INSERTs, DELETEs, UPDATEs)
  - Vulnerable code defined with AUTHID CURRENT_USER
  - Using known SQL injections to escalate privileges
  - Buffer overflow flaws
  - Definition of anonymous PL/SQL blocks for execution multiple SQL statements
  - Definition of single PL/SQL statement
Detection Approaches (CONT.)

- **Solution approach** (Against SQL injections)
  - Use bind variables
  - Avoid dynamic SQL statements
  - Define functions using AUTHID CURRENT_USER
  - Avoid concatenation of SQL statements
  - Validate input strings
  - Check for special symbols as quotes, semicolons, ‘—’, stored words like ‘union’, words equal to fields of tables that are currently in use
Available Solutions

- **Commercial Solutions**
  - No real commercial solutions that specifically detect SQL injection attempts against an Oracle database
  - Most of the commercial tools to be used properly need rules and signatures to be defined for the specific Oracle cases
  - **Available tools:**
    - Acunetix – Web Vulnerability Scanner,
    - SecureSphere of Imperva,
    - Elanize KG - Maui Security Scanner,
    - appDetective,
    - N-Stalker - Web Application Security Scanner
Available Solutions (CONT.)

☐ Free Solutions
  ☐ Even less reliable, no tool focuses specifically on Oracle
  ☐ Available tools: sqlmap, SQLiX, OScanner, BobCat, SQLBrute, NGSS SQL Injector, SQL Injection Brute-forcer and others

☐ Academic Efforts
  ☐ They just don’t exist
    (only one project: ShieldGen tool- Microsoft and Columbia University)
Solution Implementations
Comparison (Byte-Code)

Byte-code approach (by Pete Finnigan)

- Positive aspects
  - Helps understand underlying Oracle flaws
  - Byte-code is claimed to be independent of Oracle version
  - Possible to find buffer overflows

- Negative aspects
  - Code is hard to read
  - There exist differences between different Oracle versions
  - Is not easily extendable, hard to test
Solution Implementations
Comparison (Plain Text)

Plain text approach (PL/SQL analysis)

Positive aspects

- Work can be divided into unwrapping and PL/SQL analysis
- Un-wrapper independent
- Easy to extend, easy to test
- Programmer-friendly, work with PL/SQL, not byte-code

Negative aspects

- Un-wrapping is not trivial mission
- Un-wrapping is Oracle version dependent
- May cause “false alarms” in complicated cases
Our Work

- Why did we choose plain text approach
  - Better documentation available
  - More interesting work (analyze PL/SQL, not byte-code)
  - Easier to see the results and check correctness
  - More scalable implementation, easily extendible
Our Work (cont.)

- **Resources needed for our work**
  - The Oracle un-wrapper
  - PL/SQL language specification
  - Known vulnerabilities types, samples of vulnerabilities
  - Known attack types

- **Problems during implementation**
  - Building un-wrapper is not so easy as David Litchfield claims
How Our Way Works

Wrapped Package ➔ Un-Wrapper ➔ Plain text PL/SQL

Vulnerability Report ➔ Automated Detector (shown in next slide)
How Our Way Works (CONT.)
Automated Detection

Plain text
PL/SQL → LEX → List of tokens

LEX → Bison Parser

Vulnerability Report

Known Vulnerabilities

Vulnerability Report

Vulnerability Report → Vulnerability?

Vulnerability Report → Vulnerability?

Vulnerability Report → Vulnerability?
Automatic detector and its features

- Given a PL\SQL source file as an input, tries to locate suspicious commands
  - Those having a non-zero probability of being vulnerable to SQL injection attack as described earlier

- Simple implementation
  - Perform only those operations/calculations which are really needed

- Easily extendable
  - Only basic knowledge of Lex/Bison is required in order to add new rules for vulnerability detection

- Fault tolerant
  - Silently ignores unidentified code
Maximize the number of real vulnerabilities detected

- Recognizes all types of mechanisms for dynamic SQL execution
  - EXECUTE IMMEDIATE, dynamic cursors, DBMS_SQL package…

- Treats each input variable as a potential source of danger
  - Makes no assumption about the content of function parameters

- For each suspicious input variable, calculates the set of internal variables whose values are influenced by this one
  - This way enabling detection of malicious input propagation

- Treats AUTHID CURRENT_USER as carefully as AUTHID DEFINER
  - Though a vulnerability in a function defined as AUTHID CURRENT_USER cannot be exploited directly, researches show that it can be used for attack on another function
Minimize the number of “false alarms”

- Recognizes the usage of bind variables
  - Bind variables are considered to be the best currently known solution to the problem of SQL injections

- Recognizes the usage of DBMS_ASSERT functions
  - DBMS_ASSERT provides an interface for validating input strings

- Performs basic data flow analysis to determine whether a given variable is dangerous in a given program point
  - A formerly malicious string whose value was substituted by harmless content is safe

- Checks only input variables of vulnerable types
  - It is OK to concatenate an input numeric value to a dynamic SQL statement even without knowing its exact value
A simple example

□ Taken from “SQL injection and Oracle” by Pete Finnigan

```sql
procedure get_cust (lv.surname in varchar2)
is
    type cv_typ is ref cursor;
    cv cv_typ;
    lv_phone customers.customer_phone%type;
    lv_stmt varchar2(32767):='select customer_phone '||
    'from customers '||
    'where customer.surname='''||
    lv.surname''''
begin
    dbms_output.put_line('debug:'||lv_stmt);
    open cv for lv_stmt;
    loop
        fetch cv into lv_phone;
        exit when cv%notfound;
        dbms_output.put_line(':::'||lv_phone);
    end loop;
    close cv;
end get_cust;
```

□ An output of our detector for this input is:

```
Line 12 (procedure get_cust): Vulnerability found - opening cursor with unsafe parameter lv_stmt
```
A more complicated example

- A package owm_ddl_pkg
- Size of an unwrapped package body is over 252 MB (6320 lines of code)
- A part of detector’s output for this package:

  Line 97 (procedure SKIPTOPINDEX): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 292 (procedure INSERTSADMETADATAFORSTTTAB): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 363 (procedure GENCODEFORSOMDATATSAATRSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQLSTR
  Line 425 (procedure GENCODEFORSOMDATATSAATRSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQLSTR
  Line 462 (procedure GENCODEFORSOMDATATSAATRSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQLSTR
  Line 534 (procedure COPYDDLCODETOREPLICATIONSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 586 (procedure COPYDDLCODETOREPLICATIONSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 623 (procedure COPYDDLCODETOREPLICATIONSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 667 (procedure GETCOLSTR): Vulnerability found - opening cursor with unsafe parameter SQL_STRING
  Line 772 (procedure COMMITTDDLATREPLICATIONSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQLSTRING
  Line 1060 (procedure JOININDEXCHECK): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter TABLE_NAME_VAR
  Line 1274 (procedure RECREATEVIEW): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1292 (procedure RECREATEVIEW): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1314 (procedure RECREATEVIEW): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1334 (procedure RECREATEVIEW): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1357 (procedure RECREATEVIEW): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1381 (procedure RECREATEVIEW): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1399 (procedure RECREATEVIEW): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1421 (procedure RECREATEVIEW): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1434 (procedure RECREATEVIEW): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1599 (procedure RECREATEINDEXOFTRIGS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1757 (procedure CREATESKELTONTABLE): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1771 (procedure CREATESKELTONTABLE): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 1888 (procedure TRANSFERINDEXES): Vulnerability found - opening cursor with unsafe parameter INDEX_CUR_SQL
  Line 2097 (procedure TRANSFERINDEXES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 2163 (procedure TRANSFERTRIGGERS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 2200 (procedure TRANSFERCHECKCONSTRAINTS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQLSTRING
  Line 2261 (procedure TRANSFERUNIQUECONSTRAINTS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 2326 (procedure TRANSFERCONSTRAINTS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 2414 (procedure ADDSQLSTR): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
  Line 2425 (procedure ADDSQLSTR): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
FUNCTION SKIPTOPOINDEX( IND_OWNER_VAR VARCHAR2, IND_NAME_VAR VARCHAR2 )
RETURN BOOLEAN
IS
  CNT INTEGER;
  SQL_STRING VARCHAR2(32000);
BEGIN
  IF ( SYS.LT_CTX_PKG.ALLOWDDLONTOPOINDEX ) THEN
    RETURN FALSE;
  ELSE
    BEGIN
      SQL_STRING := 'select count(*)
                   from all_sdo_index_metadata asi
                   where asi.sdo_index_owner = || IND_OWNER_VAR || ||
                   and asi.sdo_index_name = || IND_NAME_VAR || ||
                   and asi.sdo_index_type = 'TOPO';'
      EXECUTE IMMEDIATE SQL_STRING INTO CNT;
      EXCEPTION
        WHEN BADTAB_EXCEPTION THEN
          CNT := 0;
    END;
  END IF;
  RETURN CNT > 0;
END;
Future work

Still much work left to be done

- More advanced data flow analysis to detect rare complicated cases
- Defining several classes of risks
- Providing more information about a found vulnerability (for instance, which input variable was unchecked, what is an exact path from function entry point to the vulnerable statement (may be unreachable) etc.)
- Providing quick tips for a user describing what changes should be applied to the code in order to eliminate the vulnerability
- At a later stage it is possible to automatically replace vulnerable code with safe one (only in limited cases!)
Our Work References And Sources

- List of sources / resources
  - unwrap10.exe: un-wraper utility by mysterious Russian hacker
  - David Litchfield: Wiley’s “The Oracle Hacker’s Handbook” and “The Database Hacker’s Handbook”
  - Pete Finnigan: article “How to unwrap PL/SQL”
  - Esteban Martinez Fayo: “Advanced SQL injection in Oracle databases”
  - SQL Injection detection
    - [http://www.securityfocus.com/infocus/1714](http://www.securityfocus.com/infocus/1714)
  - Various PL/SQL internet books, tutorials, etc..