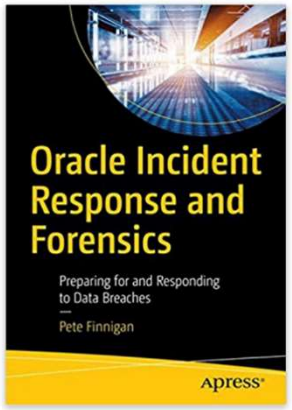

# Secure Coding in PL/SQL


# Legal Notice

## Secure Coding in PL/SQL

Published by
PeteFinnigan.com Limited
Tower Court
3 Oakdale Road
York
England, YO30 4XL

PFCL
PETEFINNIGAN.COM LIMITED




# Pete Finnigan – Background, Who Am I?

- Oracle Security specialist and researcher
- CEO and founder of PeteFinnigan.com Limited in February 2003
- Writer of the longest running Oracle security blog
- Author of the Oracle Security step-by-step guide and “Oracle Expert Practices”, “Oracle Incident Response and Forensics” books
- Oracle ACE for security
- Member of the OakTable
- Speaker at various conferences
  - UKOUG, PSOUG, BlackHat, more..
- Published many times, see
  - http://www.petefinnigan.com for links
- Influenced industry standards
  - And governments


# Agenda

- Introduction
- Common attacks on PL/SQL Code
- Types of PL/SQL code security issues
- Develop a secure coding policy, training and standards
- Detect PL/SQL issues
- Fixing PL/SQL code issues


# Section

## Introduction

PFCL
PETEFINNIGAN.COM LIMITED

# What If Your Database Is absolutely locked Down?

- We can completely lock down Oracle to prevent breaches or data loss THEN
- The only attacks would be
  - Corrupt employees can use apps to access data
  - Someone can gain access to the building and access unattended IT
  - **An attacker can exploit the applications PL/SQL business or security code**


http://www.oracle.com/us/support/assurance/coding/index.html

http://www.petefinnigan.com/weblog/archives/00001153.htm

# Oracles Efforts

- Since 2001 Oracle has had a lot of bugs reported in their PL/SQL
- Oracle has stated that they use Fortify SCA (probably now now)
- Oracle has stated and evidence shows that they use DBMS_ASSERT
- Oracle has stated in blogs and on their site that they train developers and have their own 300 page secure code standard
- We as security researchers can look at bugs fixed by oracle
- Most code in customer databases is not written by oracle and is not at same state as Oracle code. **Note EBS code is similar to customer code**.
- Oracle had a lot of pressure from the community to fix bugs over the years
- Customers **(your)** own code does not have the same pressures
- **Oracle tests its own PL/SQL codes not yours**
- **In 20 years of security audits I do not see secure code or evidence of any policy or process**


Simply following syntax tips and tricks would not be enough to create really secure PL/SQL

# Three Security Domains in Secure Code

- Secure code is much more than SQL Injection or other syntax based issues
- Secure code is also about good design

1. Design and privilege of the business code as its deployed
2. Access to dangerous logic or resources must be controlled
3. Syntax based issues / errors

- The first two need education and understanding and recognition
- The last can be by following checks/rules


# Section

## Attacks On / Via PL/SQL


# Using PL/SQL in SQL Injection

- SQL Injection attacks can be extended to not simply inject additional code into a where clause by adding a union
- By extending a SQL statement to also use a PL/SQL function its possible to do other things such as changing a password or removing audit or indeed anything else.
- The only remit is that the attackers database user must have the ability to create a PL/SQL procedure or function to enable him to inject his PL/SQL into SQL.
- Later we will see how this restriction can be lifted
- Problems:
  - SQL is vulnerable to concatenation
  - Privileges granted to the procedure
  - Owner of the procedure; better to share out responsibilities or invoker rights can be used
  - The attacker has the ability to create PL/SQL object

Demo: Run plinsql.sql


# Statement Injection Example

- A statement Injection is where PL/SQL can be injected into an existing function or procedure that executes dynamic PL/SQL
- This is worse than SQL injection as injecting PL/SQL means often that objects can be created, DDL executed and multiple statements executed at the same time making it easier to do things for the attacker that require the same session.
- Problems:
  - The procedure is vulnerable to PL/SQL statement injection
  - Privileges granted to the procedure
  - Owner of the procedure / invoker rights
  - The attacker can do anything in PL/SQL that the owner can do except use privileges granted via roles

Demo: Run statement.sql

# DDL Example

```
procedure expire_user(pv_user in varchar2) is
  ex_user_not_exist exception;
  pragma exception_init(ex_user_not_exist,-1918);
begin

  dbms_output.put_line(chr(10));

  if( upper(pv_user) in ('SYS','SYSTEM','ORABLOG','CCADMIN')) then
    dbms_output.put_line('Error: You are not allowed to expire certain users');
    return;
  end if;

  lv_rand:=dbms_random.value(lv_low,lv_high);
  lv_gen:=lv_passwd_stem||to_char(lv_rand);

  lv_cur:=dbms_sql.open_cursor;

  lv_stmt:='alter user '||pv_user||' identified by '||lv_gen
    ||' account unlock password expire';
  dbms_sql.parse(lv_cur,lv_stmt,DBMS_SQL.V7);

  dbms_sql.close_cursor(lv_cur);
  dbms_output.put_line('Expired password for '||upper(pv_user)||' is '||lv_gen);
```

- An attack would be to pass user “sys identified by a--”
- The attack is not just SQL injection but DDL injection and the SQL attack is first purpose is to bypass the security checks and then to DDL inject


Demo: Run danger.sql

# Dangerous Built In Packages

- Often it is advantageous to use packages shipped by Oracle as they provide functionality that is pre-written
- Often the functionality used in Oracle shipped packages can be considered “dangerous” as the packages access the network, file system, operating system, Oracle internals and more
- Any Oracle provided package used in your code should be considered as to “why it is used” and is it necessary or can a different approach be made
- Dangerous packages might include DBMS_SCHEDULER, UTL_TCP, DBMS_ADVISOR and more
- If dangerous packages are used then controls should be added to limit the use to that actually needed and also if necessary the choice of definer or invoker should also consider results of an attack


# Section

## Types of PL/SQL Code Issues


# Core Types of Code Problem

- Privilege escalation
- Injection – SQL and PL/SQL, DDL, XSS,
- Open Interfaces
- IPR and data loss
  - Critical Data, Business Data, Source code
- Runtime issues
  - Owner, access user, permissions (i.e. just run it and access data)
- Replacement
- Cursor snarfing / cursor injection
- Use of dangerous code - DDL
- Use of dangerous built-in code
  - File system access, Java, C, Network access
  - Vulnerable Oracle packages
- TOCTOU issues


There are many more areas that we can consider as weaknesses in PL/SQL. Some are highlighted here – some discussed in this lesson in more details, some not

# More Areas to Consider

- Comments that cause issue:
  - FUDGE, BODGE, FIXME, ****, ####, TODO, BUG, TICKET…
  - Comments that describe security functions and how they work – i.e. encryption key location or password mechanism
- Use of
  - Undocumented packages – i.e. %_FFI, KUPP$...
  - Deprecated methods and functions, i.e. DBMS_JOB
  - Vulnerable functions – i.e. CPU, PSU
  - Dangerous, i.e. UTL_MAIL
  - Open Interface, i.e. DBMS_DDL
- Magic values, i.e. passwords, keys, IP Address


# More Areas to Consider - 2

- Values passed to dangerous code, i.e. key passed to DBMS_CRYPTO
- Resources not managed – opened and not closed
- Dependencies – i.e. secure one layer and expose others
- Debug, Trace, Wrapping
- Schemas accessible so bypassing security logic
- Synonyms
- Non direct paths to code
- Multi layer code
- Exceptions, lack of, hiding, never raised, never caught


# More Areas to Consider - 3

- Standalone procedures
- Exceptions – Oracle codes, code and error numbers
- Unused variables, parameters, non-initialised
- Duplication
- Unused or unreachable code
- Ref cursors
- Non-standard concatenation or home grown SQL injection solutions
- Quoted identifiers
- Use of C or Java
- Code size


# Section

## Policy and Secure Coding


# Security Best Practice in PL/SQL

- Design code for security
- Separate logic / function (code) and data domains
- Design permissions (grants)
- Design for users who access the code
  - **i.e. we cannot as developers control users as once code is deployed we can have no further say in how its used**
  - But we can code in logic to prevent bad use
    - Use role detection
    - Use privilege detection
    - Use context based security in our code
- This cannot realistically be done for all code though


# Security API as a Design Pattern

- Identify the dangerous thing. i.e. use of ALTER USER, File, Link, Dangerous package…
- Create a separate schema to own that use
- Grant the privilege to the schema
- Create an API to limit how that privilege is used – i.e. enforce least use / rights
- Choose carefully definer / invoker / inherit
- Grant just the API to the other users who need access to the dangerous rights
- Add context based controls
  - the "**accessible by**" clause in 12c or "**who_called_me**" before that
- Add additional controls to the schema to prevent bypass
  - Password locked, schema only, limit with DDL triggers…
- Filter inputs, white list and black list


# Create a Policy

- The first step in establishing secure coding practices for PL/SQL is to create a secure coding standards for all PL/SQL development
- What does could it include?
  - Industry best practice
    - SANS
    - OWASP
    - PCI
- What format should it take?
  - It should be short
  - Simple to follow
  - Each “rule” should be distinct to avoid interpretation issues
- Define testing against the policy
  - This could be manual or automated
  - No testing should take place until a standard exists otherwise any testing would be completely random


# Section

## Detecting PL/SQL Code Issues


# Testing PL/SQL Code

Analysis involves two levels of issue; static "grep" like for dangerous code or IPR and flow analysis for dynamic SQL or injection

Flow analysis could be by tainting or by processing an AST

Three options exist to test your own code but it should be driven by a policy

1. Manually review code by eye
2. Use a commercial tool to scan PL/SQL code such as Fortify or PFCLCode from PeteFinnigan.com Limited
3. Use a free tool to scan PL/SQL code such as VCG or simple SQL and PL/SQL scripts

- Using any tool (free or commercial) out of the box has value
- But: ensure that you test to your own defined standards
- Also be aware that some tools are expensive
- Also be aware that some tools do not find what you need to test for


# Manual Review

- Develop a review cycle that includes security
- Cycle
  - Code is written
  - Scan / review for security issues
  - Function test and integration
  - Scan / review again for security issues
- This should fit into the normal development process as additional tasks that check for potential security issues

PFCL
PETEFINNIGAN.COM LIMITED

# Commercial Code Scanner Tools

- https://github.com/nccgroup/VCG
  - Nick Dunn and Jonathan Murray – latest is 2.1.0; I had pre-release and tested it
- Commercial tools
  - PFCLCode - http://www.petefinnigan.com/products/PFCLCode.htm
  - Checking
  - Fortify SCA
  - IBM Rational AppScan
  - Checkmarx
  - SonarQube – see paper in directory on this from quest toad
  - codeXpert also from toad
- Links
  - https://www.owasp.org/index.php/Source_Code_Analysis_Tools
  - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis


# VCG – Simple "Grep" Like Tool

- First let us state:
  - VCG is free
  - VCG is simple and regular expressions in a grep like way
  - VCG does not do flow analysis
  - Some tests are built in and some can be configured by the customer
  - It is good at matching text strings but bad at detecting SQL injection
- For instance running against Tanel's Moats script shows that v_sql is a critical SQL injection bug but the variable cannot be reached and also the value passed in is a number not a string

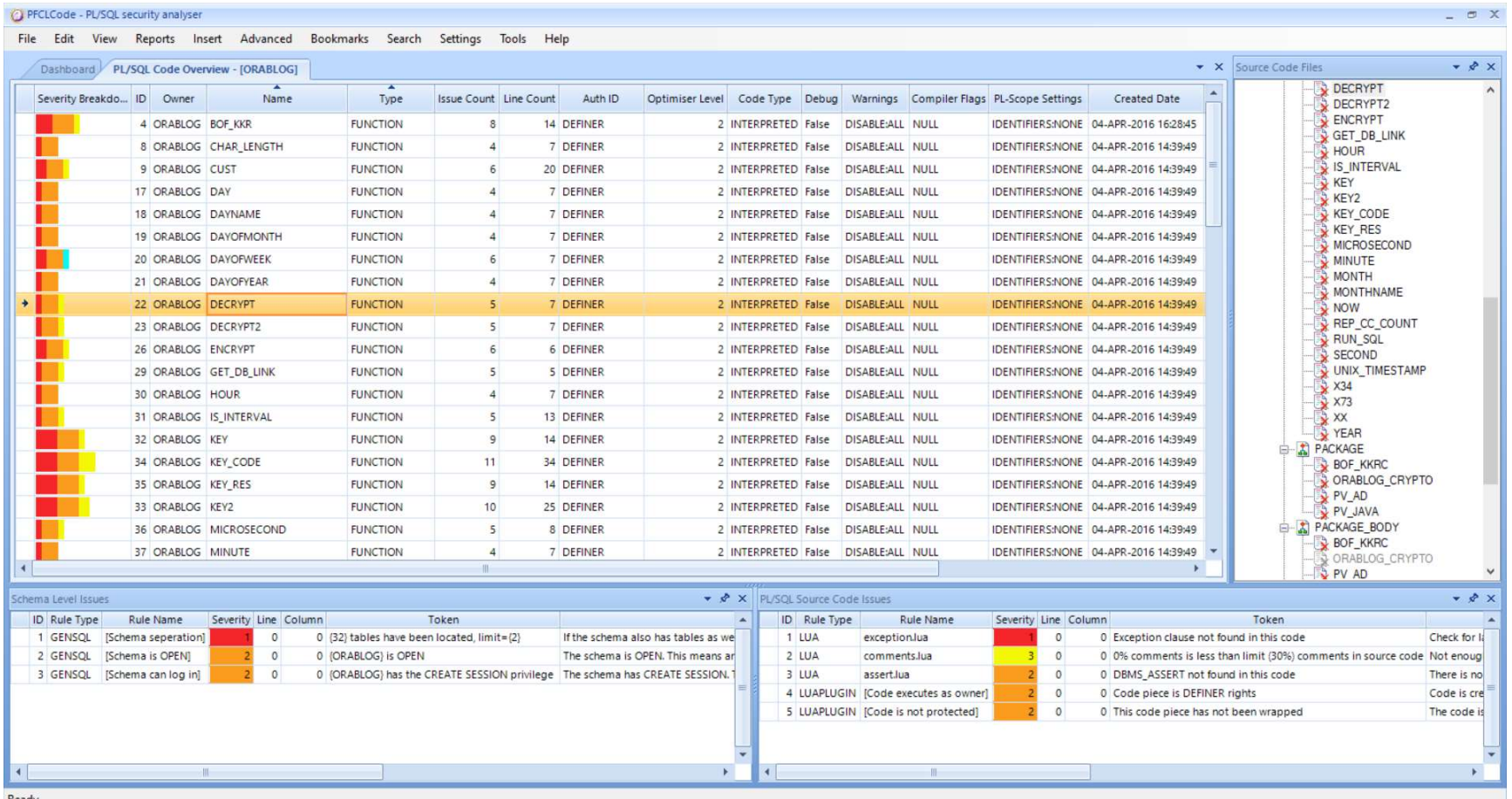

Note: It would be possible to reconstruct most of the PL/SQL source code from DBA_IDENTIFIERS so access to this view should be limited

# PL-Scope

- PL-Scope allows internals of PL/SQL to be “grabbed” by recompiling and viewable via USER|ALL|DBA_IDENTIFIERS
- 12.2 provides more depth
  - http://www.oracle.com/technetwork/testcontent/o50plsql-165471.html
  - http://www.oracle.com/technology/oramag/10-sep/o50plsql.zip
  - http://psoug.org/reference/plscope.html
  - http://www.thatjeffsmith.com/archive/2016/06/plscope-support/
  - http://www.toadworld.com/platforms/oracle/w/wiki/5770.plscope-plscp
  - http://tutorials.plsqlchannel.com/public/Presentations/MNG6/MNG6.pdf
  - https://technology.amis.nl/2007/11/14/oracle-11g-generating-plsql-compiler-warnings-java-style-using-plscope/
- Trivadis has combined a PL/SQL parser with PL-Scope with new dictionary views - https://www.salvis.com/blog/tvdca-trivadis-plsql-sql-codeanalyzer/ - This is in general not specifically security though

PFCL
PETEFINNIGAN.COM LIMITED

# PFCLCode




Available as a standalone tool or as part of PFCLScan http://www.petefinnigan.com/products/PFCLCode.htm


Demo: Run analyse.sql

# Simple Automated Code Analysis

- A simple code analysis can be done with queries against the database
- Analyse DBA_ARGUMENTS to locate public interfaces for PL/SQL that can be access by code
- Use script newcodea.sql to locate code that contains dangerous constructs such as DDL, dangerous packages such as DBMS_DDL or dynamic SQL such as EXECUTE IMMEDIATE, DBMS_SQL, DBMS_SYS_SQL and OPEN FOR
- These checks can identify code that could be analysed by hand to assess if issues exist
- The new PL-Scope interface also can provide value to analyse identifiers used in PL/SQL code.


# Section

## Fixing In-Secure Code Issues


# Example Fix: Basic Security Filtering

- White List inputs – fixed lists are best – YELLOW, GREEN, RED..
- Black List inputs
  - C style comments such as /* or
  - PL/SQL comments - -
  - Or unusual characters such as space
- Some techniques to use include:
  - Use DBMS_ASSERT.ENQUOTE_LITERAL – This will make safe any statement that could be exploited due to use of function names or unbalanced single quotes
  - Use DBMS_ASSERT.SIMPLE_SQL_NAME to validate objects passed in
  - Assert that any date or numeric format is specified against a hard coded format specifier
- Do not use quoted identifiers


# Example Fix: Static Concatenation

- Where dynamic SQL text must be used then the dynamic SQL should be built from static text and safe SQL statement text
- Static text is PL/SQL CONSTANT text
- Safe SQL statement text is text that consists of static text and also asserted text with dbms_assert.enquote_literal or dbms_assert.simple_sql_name
- The sample shows an example of these ideas

Demo: Run static.sql


# Example Fix: Bind Variables

- Where dynamic SQL or PL/SQL does not involve objects then bind variables should instead be used as this effectively prevents injection
- The sample shows a solution to simple.sql using bind variables.
- NOTE: The overall solutions are multi-faceted
  - We should use bind variables or explicit cursors
  - We should use static template text
  - We should assert input from attack
  - We should filter and test input to be from a narrow range

Demo: Run bind.sql


# Conclusions

- Focus first on architecture
- Separate dangerous code from normal logic
- Avoid loss of data via permissions issues
- Code securely (Injection, use, many more)
- Scan code for simple things
- Perform code review
- Create a policy


# Questions

?

If Anyone has questions, please ask now or catch me during the event!!


# Secure Coding in PL/SQL