SECURE SCHEMATA: WHAT NO ONE IS TELLING YOU

Barry Johnson - ~~World Bank~~
DataDemythed.com

Attend the typical presentation on Oracle security, and you will likely hear about some number of topics such as: password management, pre-DML triggers, password-protected roles, virtual private data bases, audit trails, DBMS_Obfuscation package, and Advanced Security. This is ***not*** one of those typical presentations!

Although I may touch on some of these towards the end, the fact is that they are all "after the fact" measures:

- Security must be part of the design, not some sort of add-on.
- Any afterthought is likely little more than a "speed bump".
- There have been dramatic changes to Oracle's architecture that support a better approach.
- There must be no shared authentications, *not even that of the application tables' owner*.

The conclusion is that fundamental issues must be addressed as part of setting up a data base schema. And that's the subject of this paper.

After wrestling with the "how" for some time, my epiphany came when I read the following in Oracle's documentation[1]:

> "A schema is a security domain that contain database objects ... unique schemas do not allow connections to the database ..."

So the essence of the idea is to have objects under the ownership of something *with* ***no*** CREATE SESSION privilege. We finish with clear segregation of *users* (who have connect privileges, but no objects) versus *schemata* (which have objects, but no connect privileges). This paper shows *how* to implement that strategy and, hopefully, explains *why* you will want to do so.

SCHEMA SETUP

There are a number of measures we take to lock down the schema. First of all, we create a special profile that is as restricted as we can make it:

```
CREATE PROFILE <SchemaProfile> LIMIT
    COMPOSITE_LIMIT              1
    CONNECT_TIME                 1 -- minutes
    CPU_PER_CALL                 1 -- 0.01 seconds
    CPU_PER_SESSION              1 -- 0.01 seconds
    FAILED_LOGIN_ATTEMPTS        1
    IDLE_TIME                    1 -- minutes
    LOGICAL_READS_PER_CALL       1
    LOGICAL_READS_PER_SESSION    1
    PASSWORD_GRACE_TIME          0
    PASSWORD_LIFETIME            0
    PASSWORD_LOCK_TIME      999999
    PASSWORD_REUSE_MAX      999999 --\ Set either to a large value, and
    PASSWORD_REUSE_TIME  UNLIMITED --/   the other to UNLIMITED.
    SESSIONS_PER_USER            1 ;
```

The use of resource limits assumes the Resource_Limit option is set for the data base. But you need to do that anyway, along with appropriately configuring all profiles *including* DEFAULT to reduce the "denial of service" risk from someone submitting a resource-intensive query.

With the profile done, now we can create the schema itself:

---

1. [1] **Oracle 9*i* Application Developer's Guide – Fundamentals Release 2 (9.2)**, Ch. 11 Database Security Overview for Application Developers, "*Unique Schemas*" although the equivalent can be found in older, pre 9i documentation.

```
CREATE USER <Schema>
    IDENTIFIED BY VALUES 'Locked Schema'
    PROFILE   <SchemaProfile>
    ACCOUNT   LOCK
    PASSWORD  EXPIRE
    DEFAULT   TABLESPACE <SchemaTS>
      QUOTA UNLIMITED ON <SchemaTS>
    TEMPORARY TABLESPACE <TempTS> ;
```

Obviously we take advantage of options to lock the account and expire the password. And since Oracle's password hash is always uppercase, we set the hash to a mixed-case string knowing that there is *no* password that will match it. We also set the default tablespace for objects that will be created under this schema since that will be used, when necessary, for objects created in the schema via an "any" privilege.

For now, we assign ***no*** other roles or privileges to this account.

SCHEMA ADMINISTRATION

There are two techniques for administering a schema's objects.

"ANY" PRIVILEGES

Ideally, we would like to do this:

```
GRANT CREATE TABLE,
      ALTER  TABLE, ...
    ON <Schema>
    TO <SchemaAdmin> ;
```

But, of course, Oracle doesn't support that scope-limitting "ON <Schema>" option. Instead, we have to hand out "any" privileges – such as CREATE ANY TABLE, ALTER ANY TABLE – to those who will be responsible for the schema. Of course, we do it by setting up a **<SchemaAdmin>** role and attaching the privileges to that:

```
GRANT CREATE ANY TABLE,
      ALTER  ANY TABLE, ...
    TO <SchemaAdmin> ;
```

But that leaves the recipient with data base-wide "omnipotence". How can we limit the use of "any" privileges? What if, for example, we host more than one schema in an instance, and delegate the administration of each to different groups? We want to ensure one group can't interfere with the schema of the other.

You might be forgiven for thinking Oracle has a solution to this with their schema trigger feature. Alas, ***you'd be wrong because the implementation belies the name***: it is ***not*** a schema trigger, but a ***user*** trigger. Perhaps an example best illustrates it: if I log in as Barry and, via an "any" privilege, try to change a Scott object, it will be a "schema" trigger keyed to "Barry", ***not*** "Scott", that fires. In other words, instead of:

```
CREATE TRIGGER <TriggerName>
    BEFORE DDL ON <Schema>.SCHEMA ...
```

the syntax *should* be:

```
CREATE TRIGGER <TriggerName>
    BEFORE DDL BY <User>.USER ...
```

(The schema trigger examples I've seen to date have all revealed the author's ignorance of this. Please feel free to join me in educating people on the reality. And, for what it is worth, I spent *considerable* time trying to convince Oracle, via MetaLink, that this was a bug, but they were adamant that the current implementation is correct and it is just the documentation that needs clarification. Which is a shame, really, because I've yet to conceive a use for the current implementation whereas a *true* schema trigger could be a *very* useful feature. It also requires a rewrite of Oracle's definition of "schema" cited in the introduction which is a pity, because I thought it correct.)

To use this trigger in its current form would require us to create one for *each* data base user. The good news is that there is an easy way to do that: a data base trigger! Consider the following:

```
CREATE TRIGGER <Schema>.DDLRoleCheck
    BEFORE DDL ON DATABASE
    WHEN( Ora_Dict_Obj_Owner = '<Schema>' )
    BEGIN
    IF NOT DBMS_Session.Is_Role_Enabled( '<SchemaAdmin>' ) THEN
        RAISE_APPLICATION_ERROR( -20999,
            '<Less-than-candid error message!>' ) ;
        END IF ;
    END ;
```

Things worth noting in this trigger include:

- "DDL" in the BEFORE clause is shorthand for assorted statements which, as an alternative, could be individually listed. See the SQL Reference manual for more details.
- The WHEN clause essentially turns this into the schema trigger we wanted! Within non-DML triggers, a number of special Ora_~ variables are available[2].
- The check for the appropriate role being enabled has to be inside the PL/SQL block, rather than in the WHEN clause, because SQL does not understand the BOOLEAN it returns.
- The trigger must RAISE an exception to block the DDL attempt. But the error returned probably needs to be vague since it might be someone attempting something untoward.

This could be enhanced in many ways: we could check for the DBA role and also let it through; we might send out a pager "alert" before raising the exception; …

At this point, though, we have a technique that effectively implements the GRANT ... ON SCHEMA ... statement we wanted at the start of the section.

"SUID" PROCEDURES

The second technique is, perhaps, less obvious although the idea will be familiar to *nix programmers because it is Oracle's "suid" mechanism: a way to give otherwise "unsafe" privileges to someone in a controlled, safe way. In Oracle, this is by creating a procedure in the <Schema> marked AUTHID DEFINER and then letting others execute it. Such procedures run with the privileges of the *owner* rather than the privileges of the *executer*.

> ***Bug***: In many of the currently-circulating PI/SQL releases, SYS_CONTEXT('UserEnv','Current_User') returns the wrong value within an AUTHID DEFINER procedure. It should return the procedure owner but, instead, it returns the logged-in user. The workaround is to wrap it in a SELECT ... FROM Dual, because the correct value is returned by the SQL context.

(The other AUTHID option, Current_User, means the procedure simply inherits the Current_Schema and Current_User of the calling context.)

As it happens, there are at least two cases where this is the *only* option:

- A private data base link may only be created by the intended owner. It is not a question of there being no "any" privilege for it: the statement syntax simply doesn't support it being created from outside the schema.)
- An object can only be renamed within the Current_Schema. Again, it is a matter of the statement syntax *not* supporting the use of a qualifying schema name for the object being renamed.

Prior to 9*i*R2, there was also a third case: at least the initial object privilege could only be GRANTed by the object owner. If that GRANT encompassed administrative rights – i.e., used the WITH GRANT OPTION clause – then the recipient could do subsequent GRANTs but with a downside: dropping a user from the data base *also* REVOKEs all the privileges they have GRANTed! Therefore, we really need the privileges GRANTed by the object-owning <Schema> itself.

Interestingly, 9*i*R2 introduces a new *system* privilege: GRANT ANY OBJECT PRIVILEGE. Someone with this can GRANT privileges on anything to anyone as freely as they can create a table anywhere with the CREATE ANY TABLE privilege. (Note that, under this option, all privileges are GRANTed *as if* they were assigned by the owning

2. **Oracle 9*i* Application Developer's Guide – Fundamentals Release 2 (9.2)**, Ch. 16 Working with System Events, also documented in the same manual in older releases.

schema itself. There will be no cascading revokation of privileges when someone who used this privilege is dropped.)

So, how does this "suid" technique work? Let's look at a simple implementation of the GRANT case that was necessary before 9*i*R2:

```
CREATE PROCEDURE <Schema>.Give( How  IN VARCHAR2,
                                What IN VARCHAR2,
                                Whom IN VARCHAR2 ) AUTHID DEFINER IS
    BEGIN
    EXECUTE IMMEDIATE 'GRANT ' || How
                      || ' ON ' || What
                      || ' TO ' || Whom ;
    END Give ;
```

Ahhh, but how to give EXECUTE privileges on *this* procedure? For that you will need a little "bootstrap" procedure with the appropriate GRANT "wired" into it! After that, be sure to change the above procedure to CREATE **OR REPLACE** ... so that privileges on the older version carry over to the new.

Better, of course, is to simply create a "DDL" PACKAGE that contains all the entry points of this type that you will build.

The interesting thing with this approach is the opportunity to add "features" to the basic functionality. In my implementation of the above procedure, for example, I accept an object list, and loop through it doing a GRANT on each. I also omit the first parameter and, instead, look up the type of the object: if it's a table or view, the privilege is a SELECT whilst any procedure type gets an EXECUTE.

An even better example is my data base link procedure, which includes the following:

- After creating it, the link is tested by trying a SELECT against User_Users at the other end.
- It creates local views for many of the remote All_~ catalogue views.
- It also GRANTs SELECT on those local views to the <SchemaAdmin> role. This makes it easier to look at available remote objects from within the local instance.

For procedures that create objects, the owning <Schema> must be directly given the corresponding privilege; it cannot be given via a role because roles are disabled in AUTHID DEFINER procedures. For the procedure that creates the data base link to work, we also need the following:

```
GRANT CREATE DATABASE LINK TO <Schema> ;
```

Just remember: ***never*** give the CREATE SESSION privilege!

These procedures will, obviously, have the potential to "do damage", so the code needs to be carefully written:

- Consider how exceptions are raised by the procedure. Don't reveal information that might be useful to someone trying to infiltrate the instance.
- Always check the parameters to ensure the procedure can't be compromised by malformed data. If a parameter is concatenated into a SQL statement, ensure the parameter can't be used to maliciously subvert that statement.
- It'd probably be worthwhile duplicating the role check included in the earlier data base trigger. At best, the same check will occur twice as part of a DDL execution; at worst, this one will still catch it if, for any reason, the trigger is not in place and active.

In theory, *all* administrative functions could be handled via this mechanism. I've added things that could be done with "any" privileges just so I could include some of those extra conveniences. But I *don't* expect to ever replace the complexity of CREATE/ALTER TABLE statements!

TRIGGERS

Triggers execute as an AUTHID DEFINER procedure: within a trigger, both Current_Schema and Current_User are set to the trigger owner before entering the trigger code.

This is especially reasonable behaviour if, in the course of its execution, the trigger calls another procedure and/or executes dynamic SQL. Without this, the execution results would be influenced by the caller's session. Which could give the caller an opportunity to manipulate, if not subvert, the trigger's actions.

By executing in the context of the trigger's owning schema, we ensure behaviour clearly expected at the time the trigger was created.

Views

Views also execute as an AUTHID DEFINER procedure, but with a twist: the changes in the Current_Schema and Current_User arising from the call to the view are *not* reflected in a SYS_CONTEXT call within the view. A quick look at the All_~ and User_~ catalogue views shows why this is good. Consider User_Tables, which is, *essentially*, the following:

```
CREATE VIEW User_Tables
    AS SELECT *
         FROM DBA_Tables
        WHERE Owner = SYS_CONTEXT( 'UserEnv', 'Current_User' ) ;
```

But, of course, *both* DBA_Tables *and* User_Tables are actually *owned* by Sys. If the Current_User switch was evident within the view's definition, this would *always* return Sys's objects *only*! But since Current_User *appears* to retain the value from the calling context, then the view returns the tables owned by that context, and we get the results we have come to expect.

On the other hand, if the Current_User security domain did *not* switch "behind the scenes", then we would not be able to GRANT SELECT on a view without *also* GRANTing privileges on all the objects referenced within the view. And we know we don't have to do that!

There is a simple way to prove this behaviour. Consider the following:

```
CREATE FUNCTION <Schema>.MyCurrentUser RETURN VARCHAR2 AUTHID Current_User AS
    BEGIN
    RETURN( SYS_CONTEXT( 'UserEnv', 'Current_User' ) ) ;
    END MyCurrentUser ;
```

Because it executes with AUTHID Current_User, it inherits the caller's context. Now consider the following:

```
CREATE VIEW <Schema>.CurrentUserCheck
    AS SELECT SYS_CONTEXT( 'UserEnv', 'Current_User' ) CUDirect,
              MyCurrentUser()                          CUFunction
         FROM Dual ;
```

If I – connected as Barry – do a SELECT * FROM <Schema>.CurrentUserCheck, the *first* column returns "Barry" while the *second* returns "<Schema>", even though – ostensibly – the security context does *not* change when calling a procedure of this AUTHID type! Substitute Current_Schema for Current_User and you get the same result.

Such views can*not* use Session_User: reference to an User_~ within an AUTHID DEFINER procedure must return the objects owned (or accessible, in the case of All_~ views) by the *current* security domain (i.e., Current_User). To do otherwise would let an AUTHID DEFINER procedure "peek" at the objects owned by the Session_User *without* being given such privileges. And that would be a security violation.

> ***Bug***: A few of the User_~ views, including User_Jobs, filter with Session_User instead of Current_User. Therefore these views return the wrong result within an AUTHID DEFINER procedure. A fix is available from Oracle.

Data Base Links

With security, we expect accountability. We need to be able to track who does what. Data base links that include an username/password are a major test of how well we do our job.

In an ideal world, data base links will have *no* pre-wired authentication. Anyone that uses such a link, then, will have their local credentials passed to the remote data base, and the user will have the same privileges as they would have by connecting directly to that remote data base. Unfortunately, this requires the synchronisation of credentials between data bases, a task that increases in complexity as the number of instances increases. Well, unless you pay extra money for Oracle's Advanced Security option, which adds support for network authentication services such as RADIUS, Kerberos, and PKI. (I return to this later.)

Commonly, people set up a "generic" username in the remote data base, and wire that (with its password which, by the way, is stored in the local data base as clear text) into the local data base link. Now the person accessing remote objects will be *masked* from the remote instance; think "anonymous ftp". By implication, the responsibility for tracking accesses to objects in the remote data base has been transferred to the local one. And we need to do due diligence by being able to show, when asked, how that link has been used.

But consider the case when such a link is PUBLIC: *every* user in the instance has access to everything available at the remote data base: is that *really* what the owner of that remote data intended when they shared their custodianship responsibilities with you?

Oracle provides *no* mechanism for controlling the use of a data base link. That leaves only the option of *hiding* it, by making it a *private* link, under the ownership of a <Schema>. Then we create local views of remote objects. And finally we GRANT access to those views. The result is demonstrable *local* stewardship of the remote data, a much more responsible and defendable position when the auditors start asking questions!

So the rule with respect to data base links is:

- A PUBLIC DATABASE LINK must ***not*** include a wired authentication (username/password).
- Conversely, a DATABASE LINK that includes a wired authentication ***must*** be ***private***. Access to remote objects is via local views and synonyms based on the link, on which appropriate privileges are given.

It should now be apparent why I mentioned the procedure for creating private data base links earlier! For what it is worth, I also have a procedure for creating the local view against a remote object that includes ensuring the remote object exists, amongst other things.

> ***Bug***: There is a bug in Oracle's AUDIT mechanism for private data base links: the owning schema is not captured (it is left NULL) although the session user is captured. This may lead to the erroneous conclusion that the link is owned by the session user, or maybe even PUBLIC. The last I heard, the fix is scheduled for a release after 9iR2.
>
> ***Bug***: If you execute:
>
> ```
> ALTER SESSION SET Current_Schema = <Schema> ;
> ```
>
> and *then* SELECT against a local view of a remote object with the view <Schema> implied, the link under <Schema> will be ignored. Indeed, Oracle will ***unconditionally*** attempt to resolve to a PUBLIC data base link of the same name. Oracle have known about this problem for over a year and a half, but I've yet to be told that they intend to fix it.

SYNONYMS

A goal of a good security model is predictability.

If Oracle can't find an object in the active security domain, it will look in the PUBLIC domain. Think of it as Oracle's equivalent of the *nix "path".

The interesting thing is that if a local object of the same name is created in the future, Oracle will – **without warning** – *switch* to use the new object. And if that invalidates the objects that depend on it, well ...

The other problem with depending on PUBLIC synonyms is that, by definition, they serve "multiple masters" (namely, *every* user of the instance). If someone decides to change the synonym's definition, not everyone may be happy about it.

So much better, in an attempt to control the application's destiny, is to *never* rely on PUBLIC objects. Instead, the <Schema> should have local references to *everything* required by the application. At the time of writing, my strategy is:

- For tables and views in a remote instance, create a local view that is a simple SELECT * ... against the remote object (via that private data base link, as necessary). The advantage of this is that the local data base gets a copy of the column list for the object.
- For tables and views in other schemata in the same data base, simply create private synonyms to them. For these, of course, the column list is already local.
- For anything else (i.e., procedures), create a private synonym regardless of whether it is a local or remote object.

Note that you can't create a synonym to a TYPE in another schema in releases earlier than 9*i*R2; such references must ***explicitly*** include the owning schema.

There should be ***no*** dependence on anything declared PUBLIC. Is there a way to check this? Yes, with something like the following:

```
SELECT Referenced_Name
  FROM DBA_Dependencies
 WHERE Referenced_Owner = '<Schema>'
   AND Referenced_Type  = 'NON-EXISTENT' ;
```

Each entry in the result is a dependency on a PUBLIC synonym (for which there is also an entry to be found in DBA_Dependencies) and candidate for being switched, without warning, to a currently-absent local object if ever one is created with the indicated Referenced_Name. It will included dependencies on Sys-owner objects; these may be tolerated (i.e. ignored) by enhancing the query:

```
SELECT Referenced_Name
  FROM DBA_Dependencies D
 WHERE Referenced_Owner = '<Schema>'
   AND Referenced_Type  = 'NON-EXISTENT'
   AND NOT EXISTS( SELECT '*'
                     FROM DBA_Synonyms
                    WHERE Owner        = 'PUBLIC'
                      AND Synonym_Name = D.Referenced_Name
                      AND Table_Owner  = 'SYS'
                      AND DB_Link     IS NULL ) ;
```

"Where Am I? Who Am I?"

We have mentioned aspects of it, but it's probably worthwhile to summarise the Oracle characteristics that give it some of the flavour we are used to seeing in *nix[3]:

- **Session_User**: This is the login user, which also becomes the initial Current_Schema *and* the initial Current_User.
- **Current_Schema**: This is the default object owner to *assume* when one is not explicitly given. It can be changed via the ALTER SESSION SET Current_Schema = ... statement. Like Current_User, it also changes to the owning <Schema> of an AUTHID DEFINER procedure when that procedure is entered. Think of it as the equivalent of a *nix "cd" command.
- **Current_User**: This is the current security domain controlling the session's privileges. It changes to the <Schema> owner of an AUTHID DEFINER procedure when that procedure is entered, in similar fashion to what happens when calling an *nix "suid" program. (Of course, it reverts back upon exiting that procedure.) Note that a view is supposed to execute as a "limited" AUTHID DEFINER procedure (as described earlier, and notwithstanding the bug previously cited in the way a data base link is resolved).
- **Proxy_User**: This is a trusted agent that can externally authenticate an user, and pass it for Oracle to accept "on faith". Proxy authentication is an advanced mechanism that requires a program using the C APIs or, in 9*i*, the Java equivalent.

Explicit separation of the Current_Schema and Current_User facets of the session, along with the means to manipulate them, is a significant enabler of our strategy. It lets us apply traditional, tried, and proven *nix techniques in the data base itself.

> ***Bug***: loadjava supports a "schema" parameter. But in older Oracle versions, it is ignored by the "resolve" function – even when an explicit "resolver" is given – and unconditionally attempts to resolve against the Session_User. One workaround is to simply load the java, and use DDL (ALTER JAVA ...) to resolve it later. A fix should be available from Oracle.

3. Rely on the descriptions of these items included with the SYS_CONTEXT definition in **Oracle9*i* SQL Reference Release 2 (9.2)**. The similar-looking information under "Providing Access to Predefined Attributes Through the USERENV Namespace" in **Oracle 9*i* Application Developer's Guide – Fundamentals Release 2 (9.2)**, Ch. 12 ImplementingApplication Security Policies contains errors.

The value of each of these attributes may be fetched with the SYS_CONTEXT('UserEnv', '<attribute>') call (allowing for bug citations scattered throughout this paper).

Given the strategy in this paper, it would be nice to be able to set a default Current_Schema for a user – other than their own username, that is – but, alas, I've not yet found a way to do that. It can't be done in a logon trigger for a couple of reasons:

- ALTER SESSION ... is a DCL statement, and only DML statements are permitted in such triggers.
- A logon trigger executes as an independent transaction, so presumably such a change would not filter back to the session anyway.

DML Privileges and Roles

We still have one more requirement to satisfy to complete the picture: we have to ensure only logically consistent and complete updates are applied to the data base. Since Oracle treats each DDL statement as a transaction, we only need to focus on DMLs where several statements are typically required to complete a transaction. (We ignore the use of the CREATE SCHEMA ... statement since its conformance to the minimal ANSI specification means it precludes use of many Oracle features. And it's hard to use Oracle properly without using those features! We also note that, unlike Oracle, some relational implementations include DDLs as part of the transaction, requiring a COMMIT for them to take effect, so the strategy we are about to describe would need to be adjusted accordingly.)

We require proper updates to come from code that we know we can trust. But when we look at Oracle's network protocol – SQL*Net v2, Net8, or Net, depending on your release – we see that only one thing is validated: the username.

Since there is no mechanism for certifying the *program* running at the client end of the link, we have to assume it can***not*** be trusted. The name may be available to us (in V$Session.Program), but how do we know it's not just SQL*Plus renamed? We don't.

If we can't establish the trustworthiness of the client-end program, neither can we trust it to send updates to the data base. In other words, ***we cannot entrust* INSERT, UPDATE, *or* DELETE *privileges to any code executing outside the data base***.

That leaves the option that update routines must only exist at the server end, where they can be trusted by dint of getting onto the machine. But in the same way, we can't guarantee a *sequence* of procedure calls from the client to the server. So our routines must satisfy the following rules to meet our requirements:

- Each consistent update will require a *single* call to a server-side procedure. The procedure will need to have as many parameters as required to meet the needs of the transaction.
- That procedure must perform *all* INSERT, UPDATE, and DELETE actions required for a single, logically-consistent, update of the data base.
- Finally, the procedure must do a COMMIT – or ROLLBACK – *before* returning control to the caller.

To put it another way, the only DML privileges GRANTed to a data base user are SELECT and EXECUTE. (Note, on the other hand, that Oracle's implementation of DDLs – a single call does everything including a COMMIT before returning to the caller - satisfies the requirements to be directly GRANTable!)

There are good reasons for wanting to do this anyway. By having the update routine implemented once at the server end, it is easier to support the same transaction from assorted sources: via a web form, using a custom client, and even as part of a batch run. But it must also sufficiently validate its parameters so that even someone typing in a procedure call at a SQL*Plus prompt cannot "do damage". Because, as we noted earlier, there's no way to tell whether that's what is happening.

Of course, these privileges should be assigned via roles. I would recommend structuring roles so that setting a single role suffices to be able to perform an update, including any SELECTs necessary to gather data for the update call.

(And forget password-protected roles as a way to address the issues I raise. Any sense of security they bring is false and easily compromised[4], especially if you are not using the Advanced Security option we'll discuss shortly.)

TRADITIONAL SECURITY TOPICS

That's about it for the *basic* architecture of a secured design. And we've yet to do more than mention in passing some of the "traditional" topics I cited at the start. As it happens, I have opinions on some of these that I think it important to share before I finish.

ADVANCED SECURITY

This feature – ***which is, astonishingly, an extra-cost option*** – is my litmus test when judging a book's coverage of Oracle security. Since the books I've checked to date give this feature something between no mention and *maybe* a page or two, I'm afraid they all "fail".

You cannot secure an Oracle-based application without this option.

Why is Advanced Security important? It adds two features to Oracle:

- The ability to authenticate data base users via network services such as RADIUS, Kerberos, or PKI.
- Encryption of [SQL*]Net[8] traffic.

Since, out of the box, Oracle already includes *some* level of authentication (especially once password management appeared in Oracle8), it is the second feature that is most important. With*out* the option, *all traffic transits the network "in the clear"*. Would you run an "ecommerce" web site without using TLS/SSL? And [SQL*]Net[8] is the conduit *directly to the data itself*.

(Actually, there is a trivial exception to this rule: the logon password is initially sent encrypted. But if that fails, it will be retried in the clear *unless* an option is set *at the client end* to *not* do this. Since I, as an administrator, have no control of the client, this seems pretty pointless. What I need is a *server*-side option to tell the client that a clear-text password will be refused, so it shouldn't even try it.)

Lest you find comfort in the idea that [SQL*]Net[8] might be secure because of its proprietary nature, check out a GPLed network "sniffer" called "ethereal"[5]. Amongst the protocols it claims to support is TNS (Transparent Network Substrate), which is the technical name for the [SQL*]Net[8] protocol.

So my advice is, at a minimum, to buy the option and enable the encryption feature, and then sleep a little better at nights. (I will *not* tell you that implementing it is going to be easy.)

Note that this option can only be used with JDBC Type 2 "thick" clients. If you prefer JDBC Type 4 "thin" clients – and you should! - you are out of luck. At least until the new JDBC v1.4 specification is implemented, if I read that correctly.

ORACLE AUDIT

This is a "logical" log of actions that occur within the data base. Conventional wisdom says the overhead is probably too much to use it. Conventional wisdom is wrong.

At the risk of showing my age, I remember trying to find people who were using redo logs back when they were optional on Oracle v5, but most people avoided them because of the assumed overhead. Firstly, the overhead turns out to be manageable if not reasonable, and secondly the benefits are important. With very little prudence on your part, I'm sure you'll find the same to be true with AUDIT.

Simply put, your AUDIT strategy starts as:

- NO AUDIT for Sys.Dual, Sys.DBMS_STANDARD, Sys.STANDARD, and Sys.~$ objects except Sys.Aud$.

---

4. See my article, "*Password-Protected Roles: No Security Blanket Here!*", in the August 1999 (Vol. 6, No. 5 – "electronic only") edition of the **Select Journal** archives on the IOUG web site. In the introduction to the paper I opine "... an application architecture that relies on [Password-Protected Roles] is an abdication by the DBA of all security responsibilities. ..." Hopefully, I then go on to substantiate that claim.
5. www.Ethereal.Com

- All failures BY ACCESS.

  (*Heh*, the first time I tried this I discovered programs mis-handling "not found" conditions.)
- All DML successes BY SESSION.
- All other successes BY ACCESS.

  Since DDLs should not occur too often, and given their potential impact, logging each and every one of them is prudent!

Based on what you get, you can refine it from there.

LogMiner now offers an overlapping feature by letting you extract information from the redo logs. The difference is that Audit Trail entries reflect user actions and are available immediately the event happens, whereas redo logs reflect physical file changes and can't be seen by LogMiner until a redo log is released. But I would encourage you to check them both out when deciding your strategy, and you probably want to take advantage of both options.

Another thing to note is that Oracle offers *no* administrative help with the Audit Trail. You will need to implement your own archive-and-purge mechanism. And you will want to keep an eye on it: by default it goes in the SYSTEM tablespace, and if this tablespace fills up, **the data base simply stops**. (Too bad Oracle didn't learn a lesson from the *nix world here, where the system automatically does a log file switch when a configurable file size is hit. Partitioned table, anyone??)

Finally, there are scripts around – there was even one in MetaLink, last time I looked! – describing how to move the Audit Trail into a different tablespace, which is a very good idea since this table is *waayyy* more dynamic than any of the other catalogue tables. But you should also weigh the following comment in Oracle's documentation[6]:

> "**Note:**
> Moving the SYS.AUD$ table out of the SYSTEM tablespace is not supported because Oracle code makes implicit assumptions about the data dictionary tables, such as SYS.AUD$, which could cause problems with upgrades and backup/recovery scenarios."

I've not heard of an explicit reason for this warning, but it seems to me anyone suggesting the relocation of the table should also be telling you about this warning from Oracle against it.

CONCLUSION

In summary, we have proposed some fundamental changes to the way schemata are set up and used within an Oracle instance:

- The owner of the objects should be locked down so that no one can connect with it.
  - There should be *no* shared authentications, not even among the DBAs.
- Objects are administered via two mechanisms:
  1. Give "any" privileges to schema administrators, controlling the scope of their use via a data base trigger.
  2. Implement DDL functions in schema-owned procedures with execution privileges given to schema administrators.
- Data base links with embedded username/password authentications are *always* created as private links.
  - Use of such links is by creation of objects upon them, and giving of access to those objects.
- SELECT and EXECUTE are the *only* DML privileges GRANTed.
  - All updating is via server-side, "safe" procedures requiring a single call to effect a logically-consistent update.

We don't claim this is *all* you need to do. But we do claim you will have a much stronger foundation from which to build a safe and secure application environment.

The author may be contacted at DataDemythed.com k.Org.

6. **Oracle9i Database Administrator's Guide Release 2 (9.2)**, Ch. 26 Auditing Database Use, "*Controlling the Growth and Size of the Audit Trail*".