External Table

Luca's blog on databases, data platforms, performance.

Tuesday, March 29, 2016

SystemTap Guru Mode and Oracle SQL Parsing

Introduction and motivations

SystemTap and dynamic tracing tools in general give administrators great control on their systems with the relatively little additional effort to learn the new tools. In this post you will see of how SystemTap that can be used to modify data on the fly at runtime. The outcome is a form of "live patching". Examples are provided on how to apply these ideas to Oracle SQL parsing functionality. This type of "guru mode" use of SystemTap is a corner case, but I believe it is important to know that such techniques exist and how they can be deployed, also because they can be implemented with just a few lines of code.

SystemTap has been successfully used for **emergency security band aid** of Linux systems for many years, see this presentation by Frank Ch. Eigler for full details. See also an example of how these techniques have been used in practice, described in the CERN openlab 2013 summer student lecture "SystemTap: Patching the Linux kernel on the fly".

This post is about **applying** the techniques and ideas of "live patching on the fly by data modification" to closed source application, when debuginfo is not available and in particular to **Oracle**. The post is structured around three examples of increasing complexity on how to hook and change the behavior of Oracle SQL hard parsing. Some of the topics that you will see addressed in the examples are:

- how to find the relevant function(s) to hook SystemTap to
- how to write into userspace memory with SystemTap probes
- how to modify CPU registers with SystemTap probes

Disclaimer: The tools and techniques presented in this post are intended for **learning/reference** only and are best used on a sandbox as they are **unsupported** and can potentially put at risk systems **stability** and integrity. **Administrator** privileges are needed to run SystemTap probes.

Programmable SQL filter

In this example you will see a method for **selectively blocking execution of SQL** based on programmable filter rules implemented with SystemTap.

As a first step you need to identify a relevant function for SQL parsing in the **Oracle binary**. Functions in Oracle binary are not documented, but luckily the function **opiprs** has been discussed previously in this blog and turns out to be a good choice to use with SystemTap probes. There are also other options but the details are outside the scope of this post. What you need to know about opiprs for this blog post is summarized in the table below:

Function name	Purpose	Selected parameters
opiprs	Oracle Program Interface - Parse This function is called when Oracle performs hard parsing (i.e. when a SQL statement that is not in the library cache needs to be parsed).	Notable function arguments: register rdx -> sql statement length register rsi -> pointer to the SQL text string

Note two important arguments of opiprs that are passed using CPU registers rdx and rsi: respectively containing the length of the sql statement and the pointer to the SQL statement text. The SQL text string is stored in memory, more precisely in the stack of the Oracle process, this can be confirmed by comparing the SQL text address with the process memory map from **/proc/<pid>/maps.**

A simple mechanism to implement the original goal of selectively blocking SQL execution is the following: write a SystemTap probe on the Oracle function opiprs that examines the SQL and if it matches some programmable rules block further parsing by forcing the SQL parsing to exit with an error.

SQL parsing can be forced to exit with an error by writing a 0 (end of line) in the first memory location of the buffer that contains the SQL text, effectively signaling a zero-length string. The effect of such change is that Oracle will throw the **error: ORA-900**, invalid SQL statement.

SystemTap probes can write into userspace memory using embedded C functions. This requires running SystemTap in "guru mode" and requires some additional syntax as detailed in the SystemTap documentation.

The code to implement the ideas described so far is summarized in the example script **filterSQL_opiprs.stp**. It consists of two main parts: one is a **probe on the Oracle function opiprs,** the other is an auxiliary C function called block_parse, that performs the task of **writing into memory** and specifically to the memry location that contains the SQL test string. A copy of the main text of the script for convenience:

```
function block_parse(pointersql:long) %{
    char *sqltext;
    sqltext = (char *) STAP_ARG_pointersql;
     /* Modify the SQL text with an end of line: this will throw ORA-00900: invalid SQL statement
*/
```


About Me

Luca Canali
Geneva, Switzerland
@LucaCanaliDB

View my complete profile

```
    sqltext[0] = 0;
%}

probe process("oracle").function("opiprs") {
   sqltext = user_string2(register("rsi"),"error")
   # debug code
   # sqllength = register("rdx")
   # printf("opiParse: arg2=%s, arg3=%d\n",sqltext,sqllength)
   if (isinstr(sqltext, "UNWANTED SQL")) {
      printf("FOUND!\n")
      block_parse(register("rsi"))
   }
}
```

Test the example:

1. Run the SystemTap script as root (note the oracle executable needs to be in the path) with:

```
# stap -g -v filterSQL_opiprs.stp
```

2. On a different session using SQL*Plus:

```
SQL> select 'Hello world' from dual; -- this runs normally

'HELLOWORLD
-----------
Hello world

SQL> select /* UNWANTED SQL */ 'Hello world' from dual;
select /* UNWANTED SQL */ 'Hello world' from dual
 *
ERROR at line 1:
ORA-00900: invalid SQL statement
```

This illustrates how **filterSQL_opiprs.stp** blocks any SQL that contains the string "UNWANTED SQL". The example **can be generalized** to filter generic SQL statements based on keywords or other complex rules.

Modify SQL on the fly

The code below shows an example of altering the SQL statement on the fly. It is an artificial example for demo purposes. The C function "replace_SQL" (see code) is used to write into Oracle userspace the new SQL text, effectively **modifying the statement that is being parsed**. The SystemTap script livepatch_basic_opiprs.stp is available at this link. Here is a copy of the main text:

```
%{
/* SQL that will replace TARGET_SQL */
#define REPLACEMENT_SQL "select power(count(*),3) from dba_objects"
%}

global TARGET_SQL = "select count(*) from dba_objects, dba_objects, dba_objects"

function replace_SQL(pointersql:long) %{
    char *sqltext;

    sqltext = (char *) STAP_ARG_pointersql;
    /* This changes in memory (stack) the SQL text that will be parsed */
    strcpy(sqltext, "select power(count(*),3) from dba_objects");
%}

probe process("oracle").function("opiprs") {
    sqltext = user_string2(register("rsi"),"error")
    # debug code
    # sqllength = register("rdx")
    # printf("opiParse: arg2=%s, arg3=%d\n",sqltext,sqllength)
    if (sqltext == TARGET_SQL) {
        printf("FOUND!\n") # debug code
        replace_SQL(register("rsi"))
    }
}
```

Test the example:

1. Consider this SQL. It may take days of CPU time to execute, as it has been built on purpose with cartesian joins:

```
SQL> select count(*) from dba_objects, dba_objects, dba_objects;
```

2. Run the SystemTap script livepatch_basic_opiprs.stp as root and execute the SQL again:

```
# stap -g -v livepatch_basic_opiprs.stp
```

2. Run the SQL again (flushing the shared pool is used to cause hard parsing).

```
SQL> set timing on
SQL> alter system flush shared_pool;
SQL> select count(*) from dba_objects, dba_objects, dba_objects;
```

► 2015 (10)
► 2014 (11)
► 2013 (10)
► 2012 (15)

Follow by Email

Email address... Submit

OakTable

```
POWER(COUNT(*),3)
-----------------
       7.5931E+25

Elapsed: 00:00:19.26
```

The SQL now runs in a few seconds because the statement with cartesian joins has been replaced "on the fly" by the SystemTap probe with an equivalent statement that executes much faster and without joins.
There is an important limitation to this implementation: the length of the "new" SQL statement must not exceed the length of the original SQL. A more general case is addressed in the next example.

Modify SQL on the fly, a more complex experiment

This example addresses the case of replacing SQL statements on the fly removing the limitation of the example above on the length of the SQL statement.The main point is that you also have to **update the register rdx** with the length of the new SQL statement. If the new statement is shorter than the original one this step can be omitted (as it was the case of the previous example).
How to modify the content of a CPU register with SystemTap? This is done by updating the CPU register value in CONTEXT->uregs. SystemTap takes care of restoring the register values when returning to Oracle userspace execution.

Another important point is about where to write the new SQL text, as we need a longer buffer than with the original SQL. Where to allocate the extra memory?
The example code referenced below writes the new SQL text in the process stack using the value of the %rsp pointer and subtracting 0x2000 to it. This is an educated **guess** that the target memory location is allocated to the process (in the memory chunk allocated for the stack), however enough "down in the stack" that it is free and will not be used by subsequent branches or leaf functions called by opiprs. From a few basic tests this approach seems to work, however please note also that the use of this script is intended mainly for reference and education purposes and can be potentially dangerous for system stability.

The SystemTap script livepatch_opiprs.stp is available at this link.

The proposed example script livepatch_opiprs.stp replaces the SQL "select sysdate from dual" with "select sysdate -1 from dual". This is inspired by a hypothetical situation where you want to replay a workload with time-dependent SQL. Another example of SQL replacement you may want to test is adding SQL hints. **Customize** the SQL replacement as you wish by editing REPLACEMENT_SQL and TARGET_SQL in the script.

Test the example:

```
SQL> alter session set nls_date_format='YYYY-MM-DD HH24:MI';
SQL> select sysdate from dual; -- all normal up to this point

SYSDATE
--------------
2016-02-22 12:00

Run the SystemTap script as root
# stap -g -v livepatch_opiprs.stp
```

Now the same SQL will return a different result (that is sysdate -1 instead of sysdate):

```
SQL> alter system flush shared_pool;
SQL> select sysdate from dual;

SYSDATE
--------------
2016-02-21 12:00
```

Clean up after testing

Once a SQL statement is hard parsed, all subsequent executions will also run with the **modified** text. If you want to revert to normal Oracle behavior you need to **flush the statement out** the library cache an re-parse (after having stopped the SystemTap script). For flushing statements out of the shared pool you can use "alter system flush shared pool" or dbms_shared_pool.purge (see also this post on dbms_shared_pool).

Systemtap and gdb

In the examples discussed in this post SystemTap has been used almost as an automated **debugger**. Notably with the addition that SystemTap has a low-overhead compared to many debuggers and provides a powerful programmable interface for defining the SQL text search and replacement actions. Another advantage of **SystemTap** is that it can **attach to all running processes** (of the Oracle executable in this case) if desired.
For completeness and as a reference, this is a short list of relevant **gdb** commands that can be used to reproduce some of the actions described in the examples of this post:

gdb -p <pid> start gdb against an existing Oracle session
break opiprs -> define a breakpoint on opiprs
continue -> continue program execution till breakpoint
info reg -> show registers
bt -> backtrace
x/1s $rsi -> visualize the SQL statement string using $rsi as pointer

write the new SQL length into the CPU register rdx:
 set $rdx=24

copy a SQL string into memory and update $rsi (examples):
 set $rsi=$rsp-0x2000

```
p strcpy($rsi, "select sysdate from dual")
set $rdx=25
```

Conclusions

This post illustrates SystemTap techniques that can be used to **modify userspace data on the fly** at runtime. Examples are provided on how to apply these techniques to Oracle SQL parsing. The techniques discussed here can be **generalized** and used on other functions of the Oracle kernel as well as be extended to "live-patch" data at runtime for other applications in the **Linux** environment. In particular the provided example SystemTap probes show techniques for writing into userspace **memory and into CPU registers**, useful to address cases where debuginfo is not available.

Disclaimer: The tools and techniques presented in this post are intended for **learning/reference** only and are best used on a sandbox as they are **unsupported** and can potentially put at risk systems **stability** and integrity. **Administrator** privileges are needed to run SystemTap probes.

Acknowledgements and references

A shout-out to Frank Ch. Eigler, for his work on SystemTap and in particular for his presentation "Applying band-aids over security wounds with systemtap" and related blog post and also for the tip on how to write into CPU registers with SystemTap.
Many thanks to Frits Hoogland for comments and improvements to this post and for collaboration on the investigation of Oracle internals and the use of Linux dynamic tracing tools for Oracle troubleshooting.

Link to the example code in Github.
Links to previous work on this blog on using SystemTap for Oracle tracing: SystemTap into Oracle for Fun and Profit and Life of an Oracle I/O: Tracing Logical and Physical I/O with SystemTap, Diagnose High-Latency I/O Operations Using SystemTap.

Posted by Luca Canali at 9:41 PM

Labels: internals, Linux, SystemTap, tools

No comments:

Post a Comment

Links to this post

Create a Link

Home Older Post

Subscribe to: Post Comments (Atom)