



DTrace by Example: Solving a Real-World Problem

Paul van den Bogaard

January 2007

Sun Microsystems, Inc.

Copyright © 2007 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements. Use is subject to license terms. This distribution may include materials developed by third parties.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and in other countries, exclusively licensed through X/Open Company, Ltd. X/Open is a registered trademark of X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, Solaris, and OpenSolaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

This product is covered and controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Table of Contents

Introduction.....	4
The Application Domain.....	4
A Short Introduction to DTrace.....	4
Example.....	5
The Initial Situation.....	5
Watching the Children of the Process.....	9
doorfs: What Is It?.....	18
Name Service Cache Daemon.....	22
References	23

Introduction

I ran into a secure shell (SSH) issue when preparing some demonstrations. At login, more than 10 seconds passed before I was prompted for a password. Since the demonstration would include watching statistics while some programs were running, I needed multiple shells. Because changing these settings over and over mandated frequent, repeated logins, these mandated pauses were more than a mere nuisance.

In this article you will learn how to use Dynamic Tracing (DTrace) in the Solaris™ Operating System to resolve the issue of these pauses. The article describes the steps taken and the reasoning behind them. Each step results in a DTrace script and each script gives more insight into the problem. Each new insight raises more questions, while also narrowing the focus of attention. The result will be gaining a better understanding of the situation. Together with some application-specific domain knowledge the final questions can be answered and the problem resolved.

I assume the reader has no specific background in DTrace and its D programming language, nor of the application domain. I do assume the reader can read a C-like programming language source. For a complete overview of DTrace, its language, and standard providers, please check the *Solaris™ Dynamic Tracing Guide*¹. For a complete description of the `dtrace` command, see the manual page `dtrace(1M)`. The OpenSolaris™ DTrace community² is a valuable resource for all kinds of DTrace-related information, example scripts, documentation, and so on.

The Application Domain

The SSH application is used to safely log in to a remote system. During the connection setup, sensitive data like user name and password is sent in an encrypted form. Once the connection is set up, all data that is sent from either machine to the other is also encrypted.

This SSH client contacts the `sshd` daemon on the host machine. The daemon uses port 22 as the default port on the server to see the incoming request. Other configuration items that are `sshd`-specific can be found in the `sshd_config` file, which is in the directory `/etc/ssh`. For a complete description of the `sshd` daemon, the configuration file, and the SSH client, see the corresponding manual pages for `sshd(1M)`, `sshd_config(4)`, and `ssh(1)`.

A Short Introduction to DTrace

DTrace is a comprehensive dynamic tracing framework for the Solaris OS. DTrace provides a powerful infrastructure to permit administrators, developers, and service personnel to concisely answer arbitrary questions about the behavior of the operating system and user programs. DTrace consists of multiple elements, which are built into the Solaris 10 kernel: the D language, the `dtrace` command, the providers, some DTrace-related libraries, and the DTrace framework.

The scripts are written in the D language. These scripts are started with the `dtrace` command. This command will check and compile the script into an intermediate language that is interpreted by the `dtrace` virtual machine embedded inside the kernel. On a default Solaris 10 installation, you need to have root access in order to execute DTrace scripts. It is advisable to use the new privilege rights management feature of the Solaris 10 OS to enable non-root users to use DTrace.

A script contains one or more probe-predicate-action triplets called clauses.

DTrace probes are points of instrumentation inside the kernel. Related probes are grouped into DTrace providers, each of which performs a particular kind of instrumentation. Specifying a probe in your script enables it to collect data whenever a probe is fired.

If you enable a probe it will fire when the corresponding event occurs. For example, the `syscall::write:entry` probe fires when any process on the system calls the `write` system call. The predicate part of a clause evaluates to a boolean value of true/false and the probe is only activated when the predicate becomes true. The predicate part therefore enables you to set the appropriate context in which the action should be executed. The action part is the actual code that runs when the probe fires and the predicate holds (evaluates to true).

A probe is fully specified by the four tuples `provider name`, `module name`, `function name`, and `name`. In a probe specification these elements are separated by a colon. Any of these can be unspecified. The format is `provider:module:function:name`. Reference to these can be done by the use of the predefined variable names: `probeprov`, `probemod`, `probecfunc`, and `probename`.

If a tuple element is unspecified the probe will match all possible entries for that tuple. Unspecified matches everything, like a wildcard.

Example

```
syscall::write:entry
/pid == 1/
{
    printf("Fired\n");
}
```

In this script, the probe specifies the entry of all write functions in any module (not specified) from the `syscall` provider. In other words, this probe fires when a `write` system call is made. The predicate `/pid == 1/` places a restriction: only if this `write` call is done by that process with process ID (PID) of 1.

If process 1 does a `write` call, the corresponding action is executed: `Fired` is printed right after this `write` was called, but just before it is executed.

The Initial Situation

The current status is that it takes quite some time when trying to log on to the system through SSH. On that system, a daemon (`sshd`) is running in the background, waiting for requests to arrive on port 22. When a request comes in, it will fork a child. This child then handles the request and helps the client to gain access to the system. It will check the credentials and only when they are found to be okay will it start a shell. Only at that time will there be a working environment.

The observation "It takes quite some time," has been made. The question that follows is, "What consumes time?" During the lifetime of a process, the process either uses CPU or waits for a resource. CPU consumption can easily be determined by a utility like `vmstat`, `mpstat`, or `prstat`. Since the process of interest is known, the `prstat` command is the obvious choice. When used with the `-m` (enable microstate accounting) and `-L` (give information for all threads) options, a quick impression can be formed about the CPU consumption of a process. For further information on `prstat` see

`prstat(1M)`. One way to determine the process ID of the `sshd` daemon is to log on to the system. Now, from the shell, give the `ptree` command as follows:

```
$ ptree $$
462  /usr/lib/ssh/sshd
    2457  /usr/lib/ssh/sshd
        2460  /usr/lib/ssh/sshd
            2462  -bash
                2478  ptree 2462
```

The `ptree` command shows the hierarchy of processes that ends in your shell. As explained above, this tree starts with the `sshd` daemon of interest. The PID that needs to be used is 462.

Using `prstat` now shows:

```
$ prstat -m -L -p 462
  PID USERNAME  USR  SYS  TRP  TFL  DFL  LCK  SLP  LAT  VCX  ICX  SCL  SIG  PROCESS/LWPID
  462 root         0.0  0.0  0.0  0.0  0.0  0.0  100  0.0   0   0   0   0   sshd/1
Total: 1 processes, 1 lwps, load averages: 0.05, 0.04, 0.04
```

This way of monitoring shows the percentage of time spent in the states `USR` (user time), `SYS` (system time), `TRP` (trap handling), `TFL` (text page fault handling), `DFL` (data page fault handling), `LCK` (wait for user locks), `SLP` (sleeping), and `LAT` (waiting for CPU). This also shows that process 462 is 100% in `SLP` state; that is, sleeping and waiting for some resource. Letting the `prstat` command run for a while will update this information every five seconds. If there is no sudden burst of connection requests this output will not change. In other words, CPUs are barely consumed.

The only way to obtain a resource is through a system call. If a process is waiting it must be in a system call. By showing which system calls are consuming time, you can determine probable candidates for further inspection.

The original question, "What consumes time?" is now refined to, "Which system call(s) consume(s) time?" This question can be answered by proper use of `DTrace`.

`DTrace` has three variables to depict time. If elapsed time is of interest, the `timestamp` variable is the right one to use. This one holds the number of nanoseconds after some moment in the past. The value has no relation with wall clock time. Therefore it is best used to measure intervals.

To measure the elapsed time during a system call, you need to get one timestamp for when the system call begins and a second timestamp for when the system call returns. Here is a code snippet:

```
syscall:::entry
{
    ts = timestamp;
}
```

When a system call is entered (`syscall:::entry`) a timestamp is taken and stored in the variable called `ts`. To get an interval, the second clause is needed:

```
syscall:::return
{
    ela = timestamp - ts;
}
```

If a system call returns (`syscall:::return`), the elapsed time is calculated by taking a new timestamp and subtracting the value in `ts` from it. The result is stored in another variable called `ela`.

How are these two clauses related, since the PID (process identifier), TID (thread identifier), and system call information are not specified?

The two code snippets above define two variables. If nothing else is specified, these two variables have global scope (all processes, all threads, every probe in this script). Surely that is not what is needed. First, to ensure that every thread running on the system has its own variable the `self->` construct must be used.

Second, the probe `syscall:::entry` fires for every system call. This means all system calls done by every process running on this system are tracked. Our interest is only for those system calls made by that one `sshd` process.

By including a predicate that depicts this, the action `ts = timestamp;` will only be executed for system calls started by that particular `sshd` process. The predicate for this is `/pid == $target/`. `$target` is a predefined variable. This variable has that value specified on the command line by the `-p` option. For example, `dtrace -p 12345 myscript.d` ensures that the variable `$target` will have the value `12345` when `myscript.d` is executed and there is a process with a `pid` of `12345`. It will also hold the `pid` of the process started by `dtrace` if `dtrace` was started with the `-c` option.

The complete code for this clause becomes:

```
syscall:::entry
/pid == $target/
{
    self->ts = timestamp;
}
```

DTrace has some predefined variables like `pid` and `tid`. `tid` is the thread identifier of the thread that triggered this probe. `pid` is the process identifier of the process to which this thread belongs. This means that for each probe that is triggered, you know what triggered it. The construct `/pid == $target/` is a predicate that is true only if the probe fired due to your (`$target`) process of interest.

Once the system call returns the probe, `syscall:::return` fires. If this is due to your process of interest, then `dtrace` needs to execute the corresponding action. However it can happen that a system call is in progress when you start your DTrace script. In that case, you did not see the `syscall:::entry` probe fire. Therefore, you do not have a valid value in the `self->ts` variable. To circumvent this problem the appropriate predicate should become `/self->ts/`. Only if there is a value in this variable will we be able to determine elapsed time. The `self->ts` belongs – and is visible – to a single thread. We are interested in that thread. Therefore, the part `pid == $target` is not needed any more. The complete code for this functionality is now:

```
syscall:::entry
/pid == $target/
{
    self->ts = timestamp;
}

syscall:::return
/self->ts/
{
    ela = timestamp - self->ts;
}
```

The original idea was to measure the total elapsed time for all system calls. The name of the system call is the key used to index an array. The arrays provided by DTrace are associative arrays. As in C, the square brackets are used as the notation for arrays. The following example defines and uses the array `total`:

```
dtrace -n 'syscall:::return { total[probename] = 1; }'
```

A special kind of DTrace array is the aggregation. An aggregation is the only type that can be used with aggregation functions (`count`, `min`, `max`, and so on). It is this data type that must be used to determine total elapsed time by system call. Adding the line `@tot[probefunc] = sum(ela);` to the `syscall:::return` probe defines and enables use of an aggregation that holds the total amount of elapsed time per system call. To understand this construct, note that this would look like `tot[probefunc] = tot[probefunc] + ela;` if done in C.

```
syscall:::return
/self->ts/
{
    ela = timestamp - self->ts;
    @tot[probefunc] = sum(ela);
}
```

If the DTrace script finished either because it did an `exit()` call or because it was interrupted, the `END` probe fires. It is this probe that must be used to print the content of aggregated data. If no explicit `printa` calls are done in the `END` clause, or if there is no explicit `END` clause, then `dtrace` will print the content of every aggregation.

`timestamp` returns a value measured in nanoseconds. The final code will be used to find elapsed time in the seconds range. To make the calculated numbers more readable, the `normalize` function is used. The `normalize(@tot, 1000);` code line ensures all the values available in this associative array are divided by 1000. To dump these values to standard output the `printa` function is provided. This will print the aggregation key and the corresponding value according to the format specification if provided. In `printa("%12s %@12u\n", @tot);` the format specifier is `%12s %@12u\n`, which means first print the key using a minimum of 12 characters, and then print the value for this key as an unsigned integer using a minimum of 12 characters.

The complete DTrace script is:

```
syscall:::entry
/pid == $target/
{
    self->ts = timestamp;
}

syscall:::return
/self->ts/
{
    ela = timestamp - self->ts;
    @tot[probefunc] = sum(ela);
}
END
{
    normalize(@tot, 1000);
    printa("%12s %@12u\n", @tot);
}
```

Executing this script produced the following output:

```
./sshd1.d -p 8143
  sigaction          11
 lwp_sigmask        17
   gtime            25
 setcontext         31
   fcntl            39
  waitsys           42
   pipe             49
   ioctl            59
  getpid            61
  open64            169
  accept            170
  close             190
  fstat             209
  putmsg            215
  doorfs            234
  open              708
  fork1             9445
 pollsys            12957382
```

From this output it seems obvious that `pollsys` is the culprit, with a total accumulated time of 12.957 seconds.

However, hold on! Recall what `pollsys` does? Could this be normal behavior? Indeed, the natural behavior of the `sshd` process is to wait for a request on port 22. It is normal that so much time is spent in this call. Waiting for a request on a port is normally implemented by using the `select` `libc` routine, which calls the `poll` system call. If this was not the case, that would be abnormal behavior. So time spent in `pollsys` will not explain the 10 seconds. This script ran for approximately 13 seconds, of which `pollsys` explains 12.957. Since looking at this process cannot explain the time lost, another process must show this behavior. Obviously this is one of the (grand)children of the process that was just discussed. Which child is it that should get the attention? How do we make this determination? This calls for a new script.

Watching the Children of the Process

In order to understand this hierarchy of `sshd` the `proc` provider must be used. The `create` probe provided by this provider fires once a process is created. The initial script can be as follows:

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

proc:::create
/pid == $target/
{
}
```

This probe fires if a process is created. The PID causing this can be the process of interest as specified with the `-p` option. However this does not cover the creation of processes by children. These children have a different PID that cannot be known upfront!

The creation of children needs to be carefully monitored in order to understand the hierarchy that starts from that initial `sshd` process. These associations are stored in an associate array.

The `create` probe has one parameter of type `psinfo_t` (see Chapter 25 on `proc` provider in the *Solaris Dynamic Tracing Guide*). One element of this structure is `pr_ppid` that depicts the parent `pid`. Another is the `pr_pid`: the process `pid` of the child. Using an associative array called `forked`, the following code can be used to combine the information:

```
/* If "our" sshd process is creating a child */
proc:::create
/pid == $target/
{
    /* remember the time the child is created */
    forked[args[0]->pr_pid ] = timestamp;
}

/* If we know the triggering process as a child, we now know one of its
children.
*/
proc:::create
/forked[ pid ]/
{
    forked[args[0]->pr_pid ] = timestamp;
}
```

This shows that the probe specification and the action are both equal. Therefore these two clauses can be combined into one clause. On a general UNIX[®] system there will be many children created. Doing a lookup in an associative array for each such event can be costly. Understanding SSH (and looking at the `ptree` output above) it is clear that all the processes of interest have the name `sshd`. This can be used to reduce the processing required, by making the predicate a little more complex:

```
proc:::create
/execname == "sshd" && forked[pid]/
{
    ...
}
```

Now let's combine these two clauses:

```
proc:::create
/pid == $target || (execname == "sshd" && forked[pid])/
{
    forked[args[0]->pr_pid ] = timestamp;
    printf("%5d created %5d\n", pid, args[0]->pr_pid);
}
```

This clause will catch all forks of `sshd` (grand)children that are called `sshd`. The array `forked` will contain the start time (since some arbitrary moment in the past) of that child. One more piece of data is needed to get information out of this: the moment in time that these children die; or, in other words, the information about how long a process lived.

```
proc:::exit
/forked[pid]/
{
    printf("%d lived for %10d\n", pid, (timestamp - forked[pid]));
    forked[pid] = 0;
}
```

Executing this script in one window, starting an SSH client to log on, and executing a `ptree` in another window, command, and exiting gives the following results:

```
# dtrace -qp 6767 -s t.d
6767 created 9526
9526 created 9527
9527 lived for 17996197 nsec
9526 created 9529
9529 lived for 15227079 nsec
9526 created 9531
9531 lived for 14698576 nsec
9526 created 9533
9533 created 9534
9534 lived for 11462312 nsec
9533 created 9535
9535 lived for 2319426491 nsec
9533 lived for 2407494647 nsec
9526 lived for 17243049061 nsec
```

Here's the output of the other window:

```
Password:
Last login: Tue Oct 31 15:39:48 2006 from 172.19.3.1
Sun Microsystems Inc. SunOS 5.10 Generic January 2005
bash-3.00# ptree $$
6767 /usr/lib/ssh/sshd
  9526 /usr/lib/ssh/sshd
    9533 /usr/lib/ssh/sshd
      9535 bash -o vi
        9539 ptree 9535
bash-3.00# exit
```

The process with `pid 6767` is the `sshd` daemon to be watched. It spawns one process (`pid: 9526`) that will handle our SSH connection. This process also spawns a couple of processes that live for a short time (`pids 9527, 9529, and 9531`). The final `sshd` process (`pid 9533`) is created after the password was entered. This can be deduced from the `ptree` listing and the times: `9535` is the shell from which the `ptree` command was started. This shell lived for about 2.5 seconds. The actual `sshd` process of interest lived for 17.24 seconds. Clearly this one is the only candidate to inspect since all others do not live long enough to explain 10 seconds of time lost. In conclusion, the first process being created by the `sshd` daemon is the process of interest, for which the system calls must be inspected.

There is one final note to make about the preceding code: Monitoring the `sshd` daemon like this will also show other children created because other people were trying to log onto this system while the `sshd` process was being monitored. During this particular test I knew this machine was dedicated to me. Therefore given this context, the above procedure was correct.

However, if multiple simultaneous logons are possible, some extra code is needed. When `proc:::create` fires, there is exactly one argument. As discussed previously, this argument is of type `psinfo_t`. The `psinfo_t` structure contains an additional element that describes the `uid` (real and effective).

A rewrite of the previously explained DTrace script is:

```
proc:::create
/pid == $target || (execname == "sshd" && forked[pid])/
{
    forked[args[0]->pr_pid] = timestamp;
```

```

    printf("%5d created %5d by %d\n", pid, args[0]->pr_pid, args[0]->pr_uid);
}

proc:::exit
/forked[pid]/
{
    printf("%5d lived for %d nsec\n", pid, (timestamp - forked[pid]));
    forked[pid] = 0;
}

```

Here the `printf` function is changed so it also displays the user ID (UID). Running this script and watching its output shows:

```

# dtrace -qp 6767 -s t.d
6767 created 9572 by 0
9572 created 9573 by 0
9573 lived for 17585966 nsec
9572 created 9575 by 0
9575 lived for 15074605 nsec
9572 created 9577 by 0
9577 lived for 14347357 nsec
9572 created 9579 by 0
9579 created 9580 by 100
9580 lived for 11359409 nsec
9579 created 9581 by 100
9581 lived for 11674291567 nsec
9579 lived for 11752588808 nsec
9572 lived for 26118751753 nsec

```

Now it becomes clear why there is "one extra" `sshd` process. The first process spawned by the daemon being watched runs with UID 0 (root). This one spawns a new `sshd` process, which has `uid 100` (an ordinary user on this system). The final code must be a little more complex to handle this situation.

When two different users are trying to access this system at almost the same time, the output (indented to show parent-child relationship) looks like the following example. (Note that the elapsed time is now divided by 1000000 and therefore represents milliseconds.)

```

dtrace -qp 6767 -s t.d
6767 created 9667 by 0
    9667 created 9668 by 0
        9668 lived for 17
    9667 created 9670 by 0
        9670 lived for 14
    9667 created 9672 by 0
        9672 lived for 15
6767 created 9674 by 0
    9674 created 9675 by 0
        9675 lived for 15
    9674 created 9677 by 0
        9677 lived for 14
    9674 created 9679 by 0
        9679 lived for 15
    9674 created 9681 by 0
        9681 created 9682 by 100
            9682 lived for 14

```

the first process creation for the first request

the process to handle the second request since entering `passwd` and exiting again was done before continuing the first connection; the setup for this second request follows.

Changing to the UID of the first

```

    9681 created  9683 by 100
        9683 lived for 5983
    9681 lived for 6063
9674 lived for 18429

    9667 created  9688 by 0
        9688 created  9689 by 111
            9689 lived for 11
        9688 created  9690 by 111
            9690 lived for 4366
        9688 lived for 4446
9667 lived for 28808

```

The first one took 18 seconds. Now finished

Changing to the UID of the second

This includes two serialized login sessions at least doubling the wait time into the 20+ seconds. Here 18.4 seconds is for the first session plus 10 seconds for the second session.

What is the pattern that can be used to make the script multi-user login aware? The process of interest is that process created by the `sshd` process that is being watched. The process of interest is also that process that creates a child that now has a `UID` equal to that of yourself.

In order to do this, the child of a `fork` system call could call the `setuid` system call to change its `UID` just before executing another process (the shell). Expanding the previous script to the one listed below will show when a `setuid` is requested:

```

syscall::setuid:entry
/forked[pid]/
{
    printf("%s(%d) chown-ed to %d\n", execname, pid, arg0 );
}

```

Every time the `setuid` is called, this clause prints the name of the executable that did this, together with the `pid` and the first argument: the `UID` to change to. This results in the following output:

```

6767 created 10211 by 0
10211 created 10212 by 0
10212 lived for 17 nsec
10211 created 10214 by 0
10214 lived for 15 nsec
10211 created 10216 by 0
10216 lived for 14 nsec
sshd(10211) chown-ed to 0
sshd(10211) chown-ed to 0
sshd(10211) chown-ed to 0
sshd(10211) chown-ed to 0
10211 created 10218 by 0
sshd(10218) chown-ed to 100          <<---
10218 created 10219 by 100
10219 lived for 11 nsec
10218 created 10220 by 100
10220 lived for 3188 nsec
10218 lived for 3266 nsec
10211 lived for 17807 nsec

```

Looking carefully at this output it becomes clear that the process of interest (the one created by the process being watched, as we learned previously) forks a child. This child will call the `setuid` system call to obtain the proper user ID. It is therefore the parent of the process that calls `setuid` that is the one of interest.

The code that will give the process of interest is as follows:

```
1 BEGIN
2 {
3     pidOfInterest = -1;
4     uidOfInterest = $1;
5     printf("UID: %d\n", uidOfInterest);
6 }
7
8 /* No need to remember target, only those processes spawned by it are
9    potentially interesting
10 */
11 proc:::create
12 /pid == $target/
13 {
14     forked[args[0]->pr_pid] = timestamp;
15     printf("Target %5d created %5d\n", pid,args[0]->pr_pid);
16 }
17
18 /* If a process spawned by $target now creates a child of its own it
19    should be remembered, since one of these will do the setuid we are
20    looking for
21 */
22 proc:::create
23 /execname == "sshd" && forked[pid]/
24 {
25     tree[args[0]->pr_pid] = pid;
26 }
27
28 proc:::exit
29 /forked[pid]/
30 {
31     forked[pid] = 0;
32     tree[pid] = -1;
33 }
34
35 proc:::exit
36 /pid == pidOfInterest/
37 {
38     printf("%5d lived for %d nsec\n"
39           ,pid, (timestamp-forked[pid])/1000000);
40 }
41
42 syscall::setuid:entry
43 /(tree[pid] > -1) && (arg0 == uidOfInterest)/
44 {
45     pidOfInterest = tree[pid];
46     printf("%s(%d) chown-ed to %d\n";, execname, pid, arg0 );
47 }
```

Here is the output from the DTrace script:

```
$ dtrace -q -p 6767 -s t.d 100
UID: 100
Target 6767 created 10334
Target 6767 created 10341
sshd(10348) chown-ed to 100
10334 lived for 1914186181 nsec
The output of our ptree command:
ptree $$
6767 /usr/lib/ssh/sshd
  10334 /usr/lib/ssh/sshd
    10348 /usr/lib/ssh/sshd
```

```
10350 -bash
      10354 ptree 10350
```

This code offers proof that this process works. In theory it can happen that multiple users log on using the same user ID. In that case, this script is not able to determine the difference. It is left as an exercise to change the script so it can handle that situation.

Some new constructs need to be introduced. These will be discussed by line number as shown in the previous code listing. Lines 1-6 introduce the `BEGIN` clause specification. The action in this clause is executed before any of the other probes specified can be triggered. The `$1` notation on line 4 depicts the first parameter for the DTrace script, as specified on the command line. In this example `$1` represents the `user id` of interest. Although it is not necessary to store this `$1` macro variable in another one, it makes the code more understandable.

Also the variable `pidOfInterest` is initialized to `-1`. This value is chosen since there will never be a `pid` on a system with this value. `-1` therefore is a nice candidate for "no `pid` has been found up to now."

The original `proc:::create` probe is now split over two different clauses. The first clause (line 11) has a predicate that enables it to measure time. The second clause (line 22) is needed because the interest is now shifted to something else. `pid` is a potential candidate of interest. The process with this `pid` is in the forked array. Therefore it was spawned by our target. The child has a `pid` that is used as the key into this associative array. This enables the lookup of the parent in the clause specified on line 42!

When the `syscall::setuid:entry` probe fires, the `pid` that triggered the probe is stored in the array tree, and this `pid` is a potentially interesting `pid`. This system call is used to change the effective user ID into the one of interest, so the parent process of the process that triggered the probe is the process for which the system calls need to be timed.

Finally, in this script there are also two equal probes `proc:::exit`. Once more, the action taken is determined by the predicate. If the exit is of a process of interest (clause at line 28) then the information related to it is no longer important: The action part cleans it. The other clause at line 35 is a dummy helping to show the proof.

One more step is needed to find the information that will offer more insight into the problem. The previously listed script needs to be enhanced to list the elapsed time per system call of the process of interest. The extra code needed to realize this follows:

```
syscall:::entry
/pid == pidOfInterest/
{
    self->ts = timestamp;
}

syscall:::return
/pid == pidOfInterest && self->ts/
{
    @ela[probefunc] = sum(timestamp - self->ts);
}

END
{
    normalize(@ela, 1000);
}
```

```
    printa("%12s  %@12u\n", @ela);
}
```

When this script is run, the output is:

```
UID: 100
```

```
Target 6767 created 10413
sshd(10420) chown-ed to 100
10413 lived for 1918517727 nsec
^C
```

c2audit	13
shutdown	16
getpeername	16
nfs	19
getgid	20
waitsys	25
lseek	27
dup	32
setcontext	32
ioctl	39
pipe	47
llseek	64
getuid	108
sigaction	113
ptime	115
lwp_sigmask	139
fstat64	166
fcntl	185
open64	198
read	199
stat64	222
getpid	257
write	493
putmsg	518
close	605
fstat	825
doorfs	1157
open	1556
munmap	1925
pollsys	3910441

This result does not meet expectations. Not even close.

The time-consuming system calls of interest happened before the `setuid` call was made. Once it is known which process is interesting, the behavior that makes this process interesting has already happened. We need to be able to look back in the past. DTrace provides speculations to enable looking back once an event happens – for example, to determine what led to an error. Unfortunately this feature cannot handle aggregations.

In order to include data from the past, every system call done by any of the processes of interest needs to be recorded. Fortunately DTrace supports compound keys for associative arrays. By prepending the `pid` to the system call, all data collected can be connected to a certain `pid`. If the `pid` is known an external script can be written to filter out all data that does not relate to this `pid`. However, you do need to use external scripting to accomplish this.

The resulting DTrace script does not need the `pidOfInterest` variable any more. The resulting updated clauses for the `syscall:::entry/return` probes are:

```
syscall:::entry
/forkeed[pid]/
{
    self->ts = timestamp;
}

syscall:::return
/self->ts/
{
    @ela[pid,profunc] = sum(timestamp - self->ts);
}
```

The only processes that are examined are those that are potentially of interest. Also, the aggregation now uses a compound key `pid, profunc`. In order to get the correct output, the `printa` call in the `END` clause needs a different format string: `%6d %12s %@12u\n`. A `%6d` is added. This one contains the format instruction for the first element of the key (the `pid`). The `%12s` is the instruction for the second part of the key. The final `%@12u` is still the format instruction for the data associated with the compound key.

Since the `pid` of interest should be known to enable an external program to filter out the relevant data, the `syscall::setuid:entry` now becomes:

```
syscall::setuid:entry
/(tree[pid] > -1) && (arg0 == uidOfInterest)/
{
    printf("pidOfInterest: %d\n", tree[pid]);
}
```

All other instances of `printf` are removed from the script. Running this script and using two different SSH clients to simultaneously log in results in the following output:

```
pidOfInterest: 10694
^C
10694      lwp_self          0
10701      lwp_self          0
10701      shutdown          0

... lots of calls taking 0 milliseconds. Followed by some using a small
... amount of milliseconds finally followed by the "huge consumers"

10701      write            9
10694      write            12
10701      open             17
10694      open             17
10701      read             35
10694      read             35
10694      pollsys           7132
10701      doorfs           7229
10701      pollsys           9018
10694      doorfs           10123
```

Our `pid` of interest is 10694. The last line in the output shows that this process used 10.123 seconds in the system call `doorfs`. This number indeed matches the time that was sought. Finally, this script is ready and shows the information that was needed. The `doorfs` call consumes the "missing" time.

doorfs: What Is It?

Since this information comes from watching the `syscall` provider, `doorfs` must be a system call. In order to understand what a system call should do, its manual pages must be read. System calls are documented in section 2. However, `man -s 2 doorfs` does not provide a document.

In the Solaris OS, the `door file system` facility is available for fast and efficient inter-process communication (IPC). It can only be used for communication between processes running in the same Solaris environment. (In the current days of virtualization there can be multiples of these environments running on one single server.)

Knowing that the abbreviation `fs` is commonly used for file system, the `doorfs` call seems to be related to this IPC mechanism. Indeed it is an undocumented call used by the `door` library functions. See `man -s 3DOOR door_create`. A `door` is created by a server. The well-known name of this `door`, behind which a service is provided, is visible in the file system. By opening this file, a file descriptor is created, which is also known as the `door` descriptor in this context. This `door` descriptor is then used as the identifier during the use of the actual `door`.

Let us now focus on this system call to get a better understanding of what is happening. The accumulation of time in this `doorfs` call could be due to some calls taking a huge amount of time or to a huge number of calls that all return almost immediately. To see the distribution of time, the following DTrace script is used. Here the `lquantize` aggregation function is used to create a linear histogram. The upper bound is known to be a little above 10 seconds. The lower bound is clearly 0, since elapsed time is measured. The `lquantize(this->ela, 0, 12, 1)` fits the value `this->ela` in a histogram. In this histogram, the cumulation of values starts at 0, and ends at 12, with an interval length of 1. Thus, `[0,1)`, `[1,2)`, `[2,3)`, `...`, `[11,12)` are the ranges covered by the elements of the histogram.

The previous script has its `syscall::doorfs:entry` and `return` changed, as shown in the following code. Because the aggregation is now used with a single key, the format string in the `printa` should also be changed.

The `this->` construct is used to create and use a local variable. This one is valid in the scope of the block in which it is defined: the action.

```
syscall::doorfs:entry
/forked[pid]/
{
    self->ts = timestamp;
}

syscall::doorfs:return
/self->ts/
{
    this->delta = (timestamp - self->ts) / 1000000000;
    @ela[pid] = lquantize(this->delta, 0, 12, 1);
}

END
{
```

```

    printa("%6d %@12u\n", @ela);
}

```

The output is now:

```
pidOfInterest: 10799
```

```
10799
```

```

value  ----- Distribution ----- count
  < 0  |
    0  | @@@@ 303
    1  |
    2  |
    3  |
    4  |
    5  |
    6  |

```

Here 10799 was the `pid` of interest. Most calls took between zero and one second. Exactly two calls took five to six seconds. The total time needed is 10 seconds, quite close to two times five. Given that five is a "human number," the cause of the problem may turn out to be due to human error.

To understand more, it is necessary to learn what these two `door` calls do to the `door` identifier. Since a `door` identifier is returned by a call to `open`, this one can be associated with a file name. (One complication in doing this is that file descriptors can be re-used after being closed.)

To find out more about this undocumented, probably catchall, `doorfs` system call, we need to visit <http://src.opensolaris.org/source/>. Looking at the C source code of the `door` system call shows extra information on `doorfs` (see http://cvs.opensolaris.org/source/xref/on/usr/src/uts/common/fs/doorfs/door_sys.c):

```

static int
doorfs(long arg1, long arg2, ..., long arg5, long subcode)
{
    switch (subcode) {
        ....
    }
}

```

The sixth parameter is called `subcode` and drives the decision in the switch. The following change of the `syscall::doorfs:entry` and `syscall::doorfs:return` probes show us the value of this and the other parameters for those calls that take longer than five seconds:

```

syscall::doorfs:entry
/forked[pid]/
{
    self->ts = timestamp;
    self->arg0 = arg0;
    self->arg1 = arg1;
    self->arg2 = arg2;
    self->arg3 = arg3;
    self->arg4 = arg4;
    self->arg5 = arg5;
    self->arg6 = arg6;
}

syscall::doorfs:return
/self->ts > 0/
{
    this->delta = (timestamp - self->ts) / 1000000000;
}

```

```

@ela[pid] = lquantize(this->delta, 0, 12, 1);
if (this->delta > 5) {
    printf("%ld %ld %ld %ld %ld %ld \n", self->arg0, self->arg1,
        self->arg2, self->arg3, self->arg4, self->arg5, self->arg6);
}
}

```

Unfortunately the `if () { }` construct is not supported in DTrace. This can be done with proper use of predicates. The `syscall::doorfs:return` clause will now be replaced by two different clauses that have the same probe:

```

syscall::doorfs:return
/self->ts > 0/
{
    self->delta = (timestamp - self->ts) / 1000000000;
    @ela[pid] = lquantize(self->delta, 0, 12, 1);
}

syscall::doorfs:return
/self->delta >= 5/
{
    printf("%ld %ld %ld %ld %ld %ld \n", self->arg0, self->arg1,
        self->arg2, self->arg3, self->arg4, self->arg5);
}

```

The output of this script is:

```

3 4290756968 4278122472 4294967295 4281999360 3
3 4290756840 4278122472 4294967295 4281999360 3
pidOfInterest: 10933
^C
10933

```

value	Distribution	count
< 0		0
0	@	303
1		0
2		0
3		0
4		0
5		2
6		0

This shows that the subcode has the value 3 for both `doorfs` calls that take approximately five seconds. Looking at the source code, it is obvious that this is a clean C program using macro names instead of hard-coded values. Fortunately, each Solaris 10 system should have the file `/usr/include/sys/door.h`. Looking at this header file it becomes clear that the macro called `DOOR_CALL` has the value 3. From the C source it follows that the function `door_call` is called:

```

switch (subcode) {
case DOOR_CALL :
    return door_call(arg1, (void*) arg2));
}

```

Since all parameters were printed, it can be deduced that `arg1` also has the value 3 and `arg2` is some high number, likely a pointer. A little further into this source file, the specification for `door_call` can be found:

```
int door_call(int did, void *args)
{
    ...
}
```

Here the first parameter is called `did`. Most likely this stands for `doorid`. The circle can now be closed. Indeed there is a `libdoor` function called `door_call`, which takes a `door id` as its first parameter and a pointer as its final second parameter. By storing the file name as a value in an associative array where the key is the file descriptor, this file name can be retrieved once the `door` descriptor is seen. The code needed to store the file names becomes:

```
syscall::open*:entry
/forked[pid]/
{
    self->filename = copyinstr(arg0);
    self->hasFile = 1;
}

syscall::open*:return
/self->hasFile == 1/
{
    filenames[arg0] = self->filename;
    self->hasFile = 0;
}
```

There are two types of open calls: `open()` and `open64()`. The `open*` notation uses a wildcard to have this probe specification match these two probes. The `copyinstr()` is a DTrace function. The file name resides in user space (the executable being watched). This DTrace script runs in the kernel. Since this string will be referenced later a private copy needs to be made to ensure it can be addressed later on.

Only if the system call returns is its return value known. The return value of the open calls both depict the file descriptor. At that time the filename will be stored with the file descriptor as its key for later retrieval.

The thread local variable `self->hasFile` is used to show the validity of `self->filename`. A special value "" could be used to do this; although not advisable, this value is a legal value for a filename.

Both the `syscall::doorfs:entry` and `return` probe must be changed in order to look up the filename that is used:

```
syscall::doorfs:entry
/forked[pid]/
{
    self->subcode = arg5;
    self->delta = 0;
}

syscall::doorfs:entry
/self->subcode == 3/
```

```

{
    self->ts = timestamp;
    self->did = arg0;
}

syscall::doorfs:return
/self->subcode == 3/
{
    self->delta = (timestamp - self->ts) / 1000000000;
}

syscall::doorfs:return
/self->delta > 4/
{
    printf("door call to %s used %d seconds\n",
           filenames[self->did], self->delta);
}

```

The output of this script is:

```

door call to /var/run/name_service_door used 5 seconds
door call to /var/run/name_service_door used 5 seconds

```

`name_service_door` is called twice. In both cases it takes about five seconds. Understanding what `name_service_door` is used for will very likely lead to the solution to the problem.

Name Service Cache Daemon

Here ends the use of DTrace. To complete this story, a small explanation follows, showing that the 10 seconds are indeed due to a name lookup configuration error.

With the Solaris OS, the `/etc/nsswitch.conf` file is used to determine the order in which resources should be inspected to determine a name. For example, to check the `passwd` that comes with a user name, to find the IP address that maps to a `hostname`, and so on.

The `sshd` daemon is a utility that takes care of user name and password mapping, and provides a service for non-local users. The time lost was before the daemon asked for a password. Therefore it is unlikely that this mapping was the cause. Therefore, the first candidate was the name server. The `/etc/nsswitch` file showed that first the `/etc/hosts` file – and then, if that was not successful, the domain lookup service (DNS) – would be used to find `hostname` or IP address. Inspecting `/etc/hosts` file showed this file was empty. Therefore a service from DNS was in place. The `/etc/resolv.conf` file is the configuration file for DNS.

One thing this tells us is the IP address that provides this service. Inspecting this file showed that this IP address had a typo. When this was corrected, the SSH client returned immediately. Clearly the problem was solved. However, the answer to the question of those 10 seconds was not yet found.

All these name lookup schemes can use a cache daemon, called the name service cache daemon (`nscd`). This is the daemon that advertises the `door` that was found (`name_service_door`). Like almost all daemons, this one has a configuration file: `/etc/nscd.conf`.

Inspecting this one shows that `nscd` defaults to a `timeout` value of five seconds while waiting for a DNS lookup. So here is the answer: The first attempt to resolve something with DNS timed out. Therefore the `sshd` daemon tried again and once more saw an error after five seconds. Since the password prompt finally appeared, surely the data that needed to be resolved was not critical.

In the end the problem was solved. This would have been almost impossible to do if the DTrace toolkit had not been available.

References

1) *Solaris Dynamic Tracing Guide*

<http://docs.sun.com/app/docs/doc/817-6223>

2) OpenSolaris Community: DTrace

<http://www.opensolaris.org/os/community/dtrace>

Licensing Information

Unless otherwise specified, the use of this software is authorized pursuant to the terms of the license found at http://developers.sun.com/berkeley_license.html