DtLeakAnalyzer.pdfProducing and Analyzing memory leak reports with DtLeakAnalyzer

Date: 05/Nov/2018
Applicable version: 1.1.0

Introduction

DtLeakAnalyzer is a collection of dtrace scripts and a post-processing java program that can analyse the produced traces and provide hints for identifying memory leaks.

In summary it supports 4 types of tracing sessions & reports.

- **Single tracing session**
  - We use a dtrace script to collect traces for memory allocator functions once and analyse them. Usually this is enough to point out a memory leak. If not we can use the combined method below.
- **Combination of multiple tracing sessions**
  - We use the same dtrace script multiple times for different time periods. A bash script is provided for convenience that can do this for specific time periods. Then all traces are analysed in combination. This method provides better chances of pointing out a memory leak. If not we can use the combined method with additional trace files described below.
- **Combination of multiple short and long term tracing sessions**
  - On top of the method above, we now also run multiple tracing sessions of another dtrace script, that does some internal dtrace processing on its own. Usually we are time-limited to the amount of tracing we can do with the methods above (due to the size of the trace files), however the additional dtrace script is designed to be able to run for longer periods of time, collecting more information. In this mode the "normal" short term traces are used to train the system in interpreting the long term traces.

On top of the 3 methods and report types described above, DtLeakAnalyzer provides an additional dtrace script for **tracing calls that led to process memory growth** (by tracking calls to the brk system call) and the processing of these traces to produce a report.

The methods above in combination provide very good support for identifying memory leaks. In the sections below, all methods are explained in detail.

The produced reports, if desired, should be processed with a tool like c++filt which improves the readability of call stacks.

In multi-processor systems it might be necessary to run the process-to-be-traced in a single processor (depending on the system, the trace events might be received out-of-order. If you see the trace event counter not increasing by one for each successive entry in the trace log file, your system has this issue. each trace entry has this counter <__`counter ….` __> *which is the first number after the* <__ *characters)*

Single tracing session

This is the simplest tracing scenario where we trace all memory allocations and deallocations and process the results to point out call stacks that allocate memory which is not allocated.

For example:

```
./trace-memalloc.d 14291 > trace-memalloc.log
```

The above command, where 14291 is the process ID of the process we want to trace, will start tracing all memory allocations and deallocations to the trace-memalloc.log file.

We have to stop the tracing manually by pressing CTRL+C. Beware because this kind of tracing can produce very large log files very fast.

Then we need to process the trace file in order to point out stacks that allocated memory that was not released and other potential issues.

The program that processed all the types of traces is called from an executable jar file:
`dtleakanalyzer.jar`

The program takes the following arguments (output of the program)

```
arguments: -f <type> <input file> <output file>

arguments: -d <directory> <output file>

arguments: -p <directory> -d <directory> <output file>

<type> = memalloc or brk.

Example: <prog> -f memalloc inputFile outputFile
```

And in the single tracing case, the following command will process the trace file and produce a report:

```
java -jar dtleakanalyzer.jar -f memalloc trace-memalloc.log trace-memalloc.log.report
```

A typical output of the command is the following :

```
Started memory allocator analysis for file trace-memalloc.log on:Thu Nov 08 08:03:54
CET 2018

Finished memory allocator analysis for file trace-memalloc.log on:Thu Nov 08 08:04:05
CET 2018

Call statistics

Found 142511 malloc calls

Found 0 calloc calls

Found 0 realloc calls

Found 142427 free calls
```

Double free issues

```
Found 0 double free stacks in total
```

Free non-allocated memory issues (may also be potential memory leaks)

```
Found 2617 stacks that freed memory that was not allocated during the period of the trace

Found 17 unique stacks that freed memory that was not allocated during the period of the trace

Found 495 unique stacks that correctly freed memory

Found 0 unique stacks that have never been found to correctly free memory
```

Memory leak issues

```
Found 2701 potential memory leaks in total

Found 44 unique potential memory leak stacks (suspects)

Found 553 unique stacks that allocated memory that was correctly freed

Found 1 unique stacks that were never correctly deleted/freed (strong suspects)
```

Let's go through each section :

- **Call statistics**
  - This shows the number of free, malloc, realloc and calloc calls
- **Double free issues**
  - This points out double free operations that can cause undefined behaviour in a program. These operations will do a free on a memory address that has already been previously freed, without a new memory allocation on this address.
- **Free non-allocated memory issues**
  - This section will point out calls that free memory that was not allocated while we were tracing the program. These might be perfectly normal calls as the memory might have been allocated before we started tracing the program. The last metric reported (unique stacks that have never been found to correctly free memory) should be checked as it might point out to wrong memory management or wrong delete operations, which might indirectly cause a memory leak. This is an interesting heuristic as it points out stacks that have never been found to correctly free memory.
- **Memory leak issues**
  - This section provides information about potential memory leaks. It will point out unique call stacks that allocated memory that was not freed (suspects). This might be perfectly normal as the memory might have been freed after we stopped tracing. Finally this section will report unique call stacks that were never freed while we were performing the traces. This is a very interesting heuristic as it points out places in the code where memory was allocated and it was never found to be freed.

Finally, the detailed report file (`trace-memalloc.log.report`) contains additional information showing:

- all unique call stacks for all of the reported categories
- A **combined call stack** for all suspected leaks and strongly suspected leaks
    - This presents in a single combined call stack all suspected leaks and how many times each call appeared. This is useful in order to have a single visual representation on where on the code most of the suspected and strongly suspected leaks occur.

Some parts of the report run on a test program are shown below:

<Report excert>

So in summary, even with a single tracing session, there are very good chances that any memory leak(s) will be pointed out. The suspect memory leaks are places in the code where memory that was allocated is not freed (which might be perfectly normal) and the strongly suspect memory leaks are reported by a heuristic which reduces the suspect memory leaks by removing all allocation stacks that at some point had allocated memory which was successfully freed later.

The free unallocated memory and strongly free unallocated memory metrics have value in comparing their numbers with suspect and strongly suspect memory leaks (for applications that perform periodic tasks) and the strongly suspected wrong free operations might lead to wrong memory management, indirectly causing a memory leak.

Combining multiple tracing sessions

If a single tracing session did not point out your memory leak, then you should proceed with this approach, performing multiple tracing sessions of different time lengths.

Besides having a better data set to work on (i.e. more traces) this technique helps further pointing out memory leaks in applications that perform periodic tasks.

When we monitor such an application with dtrace we will almost always have some unaccounted free operations (that free memory that was allocated before we start tracing) and some unaccounted memory allocation operations (that allocate memory which will be deleted after we stop the tracing). If we trace the application for a time period longer than the period of the periodic task it is performing we will have on average a specific number of unaccounted free and memory allocation operations. This number should not change significantly if we monitor the application for 5 periods or for 50 periods.

An application that has a memory leak while performing a periodic task will continue to increase its unaccounted (un-freed) memory allocations. So by performing multiple tracing sessions of ever increasing time periods (for example 5 times (x) 10, 20, 40, 80 and 160 second tracing sessions) we can point out the call stack that keeps increasing its unaccounted memory allocations against all other call stack unaccounted memory allocations which should remain statistically around a constant number.

In order to perform such a tracing session, there is a supporting script, `execute-memalloc-trace`, which calls the relevant dtrace script for a specific process id multiple times for different time ranges.

Then we process all the results with the same processing program, this time specifying the directory where the multiple traces are stored.

If you wish to start the helper script that calls the dtrace script multiple times for different time periods please check and modify execute-memalloc-loop. After you specify the time ranges and the number of repetitions you can start the program as:

```
source ./execute-memalloc-loop <PID>
```

this will run the tracing program several times and will produce several log files named trace-`memalloc.<TIME-RANGE>.<REPETITION>`, for example `trace-memalloc.20.1` means that this is the first repetition of a 20 second trace.

Then in order to process all these files together, put them in a folder and use the command

```
java -jar dtleakanalyzer.jar -d multiple-traces multiple-traces.report
```

This will produce the same output we saw in the section above for each individual file and its own report and then produce a combined report in the last argument, `multiple-traces.report`.

Reading the combined report from the multiple tracing sessions

The report summarizes the same sections as before but in a combined way for every file for each call stack. This was we can see easily point out call stacks that keep increasing their un-freed memory allocations as the tracing time increases.

The beginning of such a report starts with

```
Combined memory allocator analysis for files:

trace-memalloc.10.1 {0}

trace-memalloc.10.2 {1}

trace-memalloc.10.3 {2}

trace-memalloc.20.1 {3}

trace-memalloc.20.2 {4}

trace-memalloc.20.3 {5}

trace-memalloc.40.1 {6}
```

This outputs the files that the report was generated and allocates reference numbers for them (the {0}, {1} and so on) in order to refer to them later for convenience. The files are sorted based on their filename, which sorts them also in terms of time ranges (if the provided helper script was used in order to produce the traces).

For example the following entry from the combined log file

```
Very strongly suspected leak stack found {0}=2, {1}=1, {2}=1, {3}=3, {4}=3, {5}=3, {6}=4,  times (it has never allocated memory that has been deallocated for all trace files)


libc.so.1`malloc

<remaining of the call stack here>

…

…
```

indicates that this stack was found 2 times in trace file {0}, 1 time in trace file {1} and so on. This is very useful in order to get a combined picture of what happened in each tracing session.

The previous heuristic for reporting strongly suspected wrong free operations and strongly suspected memory leaks is further enhanced by now looking at all traces from all files to match relevant stacks.

For the strongly suspected **wrong free operations**, it will check if the relevant call stack has successfully freed memory in any of the other traces. If not it will be further pointed out as a "**Very strongly suspected**" stack, as this means that this call has never successfully freed memory that was allocated while monitoring for all traces.

For the strong suspect **memory leaks**, it will check if the relevant call stack has been successfully freed in any of the other traces. If not, it will be further pointed out as "**Very strongly suspected**" stack, as this means that this call stack has allocated memory that has never been freed in all traces, which is a strong indication of a memory leak.

Let's go through the sections of the report (each section is marked between the *** characters):

```
*** Double free cases ***

<stacks>
```

This section summarizes the double free stacks found for all files. These problems are not directly related to memory leaks but indicate problematic memory management on the application and should be fixed.

*** **Suspected wrong free cases** (stacks that freed memory that was not allocated during the tracing)***

<stacks>

This section points out stacks that freed memory that was not allocated during the tracing for each file. These cases might be perfectly normal but it is useful information to see how many of these calls and stacks we had.

*** **Strongly suspected wrong free cases** (the suspected call stacks freed memory that was not allocated during the tracing and have not been found to correctly free memory during the tracing) ***

<stacks>

This section points out stacks that freed memory that was not allocated during the tracing and have not been found to correctly free memory during the tracing. The stacks that has never been seen to correctly free memory for all trace files are indicated as “Very strongly suspected”.

*** Suspected leaks (stacks that allocated memory that was not freed during the tracing)***

<stacks>

This section points out stacks that allocated memory that was not freed during the tracing for each file. These cases might be perfectly normal but it is useful information to see how many of these calls and stacks we had.

*** Strongly suspected leaks (stacks that allocated memory that was not freed during the tracing and have not been found to allocate memory that was freed during the tracing)***

<stacks>

This section points out stacks that allocated memory that was not freed during the tracing and have not ever been found to allocate memory that was freed during the tracing. The stacks that has never been seen to have the memory allocated freed for all trace files are indicated as “Very strongly suspected”.

Then the report will finish with the

Total memory allocations that were not deleted per file :{0}=1513 {1}=1494 {2}=1526 {3}=1617 {4}=1662 {5}=1583 {6}=2701

Which indicates the number of memory allocations that were not deleted. This can be useful information in order to identify how many of these allocations we have for different tracing sessions and time periods.

A useful practical approach is to “grep” or search on your favourite editor for strongly or very strongly suspected memory leaks. Such an output is :

Very strongly suspected leak stack found {0}=4, {1}=2, {2}=4, {3}=3, {4}=2, times

Very strongly suspected leak stack found {0}=1, {1}=0, {2}=0, {3}=0, {4}=0, times

```
Very strongly suspected leak stack found {0}=0, {1}=1, {2}=0, {3}=1, {4}=0,  times
Very strongly suspected leak stack found {0}=0, {1}=0, {2}=0, {3}=1, {4}=0,  times
…
```

where we can clearly see how many times on each trace file a (very) strongly suspected leak exists. This is a good way to have a quick overview of the findings.

In summary, this technique further enhances our chances of detecting a memory leak by comparing memory allocations that were not freed in different time spans. In applications that perform periodic tasks, the call stack that has a memory leak will keep increasing its “unfreed” memory allocations compared to other call stacks that have unfreed memory allocations. Moreover, the previous heuristic of reporting strong memory leak suspects and strongly suspected wrong free operations is further enhanced by looking all call stacks from all traces, further pointing out strongly suspected cases as “very strongly suspected”

Combining short term and long term traces

The combined tracing method described above traces each individual memory allocation and de-allocation. These types of traces, depending on the application, may prohibit running the traces for an extended period of time. In some cases I have encountered 1.5 Gigabyte trace files after only 40 seconds of tracing.

This method describes a way of complementing the previous traces with longer term memory allocator traces. These additional traces are produced by the `trace-memalloc-proc.d` dtrace script.

This tracing script does not trace each individual memory allocation and de-allocation, instead it counts call stacks that allocated and de-allocated memory. This type of information is different because we only have counts of stacks that allocate memory (i.e. how many times a specific part of the code allocated memory) and counts of call stacks that de-allocated memory (i.e. how many times a specific part of the code "freed" memory).

In order to match call stacks that de-allocate memory with call stacks that allocate memory, we use the short term traces (described in the previous section).

By reading the short term traces we train the system in understanding which parts of the code de-allocate memory allocated by other parts of the code and this information is used in order to interpret the allocation and de-allocation stacks of the longer term traces.

All memory de-allocations remove call stacks that did memory allocations and then the report points out suspect call stacks doing memory allocations that were not potentially freed by a relevant memory de-allocation call stack. We train the system to understand which parts of the code de-allocate memory from parts of the code that allocated memory and use this information to interpret the long term traces.

Creating the complementary long term traces & report

In order to perform such a tracing session, similarly as in the multiple traces method explained in the previous section, there is a supporting script, `execute-memalloc-trace-proc`, which calls the relevant dtrace script for a specific process id multiple times for different time ranges.

Then we process all the results with the same processing program, this time specifying in addition the directory where the multiple long term traces are stored.

If you wish to start the helper script that calls the long-term dtrace script multiple times for different time periods please check and modify execute-memalloc-loop-proc. After you specify the time ranges and the number of repetitions you can start the program as:

```
source ./execute-memalloc-loop-proc <PID>
```

this will run the tracing program several times and will produce several log files named `trace-memalloc-proc.<TIME-RANGE>.<REPETITION>`, for example `trace-memalloc.1280.1` means that this is the first repetition of a 1280 second trace.

Then in order to process all these files together with the short term traces and produce a report, put them in a folder (in this example `multiple-long-term-traces`) and use the command

```
java -jar dtleakanalyzer.jar –p multiple-long-term-traces -d multiple-traces multiple-traces-long-term.report
```

This command will process all the short term and long term traces and produce a report written to `multiple-traces-long-term.report`

Reading the combined report from the multiple short-term and long-term tracing sessions

The additional long-term report points out suspect memory leaks which are call stacks that have allocated memory which from our understanding of the short term traces has not been de-allocated.

Let's look at the various sections of the report

```
*** Suspect memory leak stacks (such a memory allocation has never been found to be freed in the short term traces) ***

<stacks>
```

This section of the report points out call stacks that have allocated memory that has potentially not been freed. The short term traces are analysed in order to discover allocate and free relationships between call stacks and is used here in order to judge if a call stack has allocated memory that is potentially not freed.

```
*** Unknown free stacks (may potentially free memory from the suspect memory leaks reported here) ***

<stacks>
```

This section of the report shows call stacks that free previously allocated memory that have not been identified in the short term traces. This section is useful because these call stacks might free memory allocated by some of the reported suspect memory leaks above.

So in summary, we train the system to understand which parts of the code free memory that was allocated by other parts of the code and use this information to interpret the long term traces, which only have memory allocation call stack counts and memory de-allocation call stack counts.

Note that this is a heuristic based on the (reasonable) assumption that specific parts of the code de-allocate memory from other finite specific parts of the code. The more we have covered these cases in the short term traces the better our interpretation of the long term traces will be. This method is very useful because it can expand our knowledge obtained from the short term traces to extended periods of time.

Monitoring calls that caused a process memory growth

This technique (described in http://www.brendangregg.com/Solaris/memoryflamegraphs.html) traces calls that caused process memory growth.

This is done by script `trace-memory-growth.d <PID>` which will trace all calls that caused a process memory growth for the provided process id.

The processing program will further process these traces in order to provide a summary of how much memory process growth was caused by which parts of the code. This does not necessarily mean that these parts of the code have a memory leak, it just means that these calls happened when the heap was full and the process needed more memory.

It is useful to trace these calls because we monitor the application from a different point of view and we can combine the information from this report with the information we have from the memory allocator analysis described in the previous sections.

Tracing memory growth

In order to start tracing process memory growth we use the `trace-memory-growth.d` script:

```
./trace-memory-growth.d 12345 > memory-growth-trace-file
```

and press ctrl+C when we want to stop the traces (where 12345 is the PID of the process we want to trace)

Producing and reading the memory growth report

Then we process the traces in order to produce the memory growth report.

```
java -jar dtleakanalyzer.jar -f brk memory-growth-trace-file memory-growth-trace-file.report
```

The output of the command is something similar to:

```
Started process memory increase analysis for file xxx on:Fri Nov 09 15:28:04 CET 2018

Finished process memory increase analysis for file xxx on:Fri Nov 09 15:28:04 CET 2018

Call statistics

Found 894 brk calls that increased the process virtual memory

Found 0 brk calls that decreased the process virtual memory

Found 894 brk calls that were neutral in terms of memory

Found 0 brk calls that failed

Found in total 2 unique brk stacks
```

And summarizes the number of individual calls that caused a memory allocation which in turn caused a process memory growth.

The produced report (`memory-growth-trace-file.report`) summarizes all identified call stacks. Below is an example from a test session:

*** Unique brk call stacks ***

Unique brk stack found 805 times, total size:8060928

libc.so.1`sbrk

libc.so.1`_morecore+0xdc

libc.so.1`_malloc_unlocked+0x1fc

libc.so.1`malloc+0x4c

libc.so.1`calloc+0x58

test-memalloc-leak`void*callCallocLeak()+0x8

test-memalloc-leak`main+0x74

test-memalloc-leak`_start+0x108

Unique brk stack found 89 times, total size:892928

libc.so.1`sbrk

libc.so.1`_morecore+0xdc

libc.so.1`_malloc_unlocked+0x1fc

libc.so.1`realloc+0x228

test-memalloc-leak`void*callRealloc(void*)+0x10

test-memalloc-leak`main+0x60

test-memalloc-leak`_start+0x108

Presenting brk stacks in a combined call stack

```
test-memalloc-leak`_start+0x108
          test-memalloc-leak`main+0x74
                    test-memalloc-leak`void*callCallocLeak()+0x8
                              libc.so.1`calloc+0x58
                                        libc.so.1`malloc+0x4c
                                                  libc.so.1`_malloc_unlocked+0x1fc
                                                            libc.so.1`_morecore+0xdc
                                                                      libc.so.1`sbrk      ***** Found 805 times, overall size
increase: 8060928 bytes *****
          test-memalloc-leak`main+0x60
```

```
                    test-memalloc-leak`void*callRealloc(void*)+0x10
                              libc.so.1`realloc+0x228
                                        libc.so.1`_malloc_unlocked+0x1fc
                                                  libc.so.1`_morecore+0xdc
                                                            libc.so.1`sbrk      ***** Found 89 times, overall size increase:
892928 bytes *****
```