Analyzing reports with dtleakanalyzer

Checking deletes on unallocated memory

The first part just notes memory that was unallocated, but not allocated during the time we monitor the program.

Example:

```
Info: found deallocation from unallocated memory:0x9cb54e0

Info: found deallocation from unallocated memory:0x1084bc18

Info: found deallocation from unallocated memory:0xb09e738

Info: found deallocation from unallocated memory:0xba5d510
```

This is likely not an issue, but some sample last entries (as to be near the end of the test, allowing some time of execution) should be checked to spot any wrong delete statements.

As we start monitoring the execution of the program after it has started, any delete statement to memory that was not allocated during the time the application was monitored might be absolutely correct, however some of the last sample entries should be checked, just to confirm this assumption heuristically.

This can be done by grepping the trace file for the memory address:

for example checking the trace for for the memory address `0x6717348`

```
        Line 5740427: <__188002;2018 Oct  5 09:57:02;1;delete;0x6717348

        Line 5740449: <__188003;2018 Oct  5 09:57:02;1;new;0x6717348;12;
```

Showed only 2 entries, a de-allocation and an allocation right afterwards.

We can go to the first entry (the de-allocation) to check the call stack and investigate if the delete is correct.

Checking wrongly unallocated memory

The second part analyses areas where memory was allocated with a new[] and then de-allocated with a delete (not delete[]) statement.

The entries (call stacks doing such a delete statement) are presented in their appearance frequency, where first we see the most frequent such stacks.

Example:

```
Found wrong delete stack 10074 times. Stack:

libCrun.so.1`void operator delete(void*)

…
<stack>
…


Example of this allocation. Allocation stack:


DTLeakLogEntry [entryNumber=6, date=2018 Oct  5 12:45:34, type=NEW, threadId=1,
address=0x79be490, additionalInfo=60, callStack=libCrun.so.1`void*operator new(unsigned)+0x94

…
<stack>
…

]
```

The report entry shows the call stack that wrongly unallocated memory (unallocated memory with a delete instead of a delete[]) and states how many times it detected this call stack.

Then it presents one example of how the memory was allocated. This is shown after the "`Example of this allocation. Allocation stack:`"

These cases might not cause memory leaks, however they should be checked for correctness.

Checking unallocated memory

The third part of the report presents cases where memory was allocated and not unallocated. The cases are presented in the order of frequency, with the first one the most common call stack that allocated memory that was not unallocated, during the time the application was monitored.

Example:

```
Analyzing 2180 potential memory leaks

Processing completed.

Detected 355 potential memory leaks


Suspect leak stack found 142 times

libCrun.so.1`void*operator new(unsigned)+0x94

…
<stack>
…
```

Firstly the report outputs how many unique call stacks left unallocated memory.

➔ `Detected 355 potential memory leaks`

And then present the call stacks in order of frequency.

Depending on the application logic such allocations might be perfectly correct, as the memory might be unallocated later or it might not be unallocated at all during the application execution lifetime. However, it provides a good indication of where an actual memory leak might occur.