Dynamic Memory Debugger

Amit Gaurav
amit.gaurav@gmail.com

Originally Completed: September 15, 2017
Last Updated: September 17, 2017
Current Revision: 1.0

Abstract

This document outlines the memory debugging tool that can be used to dynamically or statically generate memory allocations done between two time frames or during lifetime of a process. Understanding this document will help in catching issues such as memory corruption and memory leaks. Below sections describe the design of the tool and the steps to find issues using the tool.

Contents

1 Introduction **3**

2 Components **4**

3 Protection Bytes **5**

4 Overridden Functions **6**
4.1 malloc . . . 6
4.2 calloc . . . 6
4.3 realloc . . . 7
4.4 memalign . . . 7
4.5 free . . . 7

5 Deployment **8**

6 Allocation Report **9**
6.1 Getting Report . . . 9
6.2 Internals . . . 10
6.2.1 Generating Reports . . . 10
6.2.2 Start/Stop Allocations . . . 10
6.3 Report Table . . . 11

7 References **12**

1 Introduction

Debugging memory corruptions due to malloc in most areas is cumbersome and time taking since the manifest in such cases may not really be the cause. It is imperative to have to tool or a solution to obviate going into this endless loop of finding memory issues every time it occurs. At the same time, it is expected that such tool does not interfere with the functionality or performance of the typical code executions.

This tool uses the well known method of wrapping every memory allocation with protection bytes (or magic bytes) so that every allocation is guarded. In case of a breach, issue will be caught during memory release or core analysis. The question now is: *why reinvent the wheel and not use the available technologies in the market?* Let us review each solution and their shortcomings.

1. **Rational Purify**: It requires setting up the whole build area with all the debug symbols and with all the instrumented libraries. We are not really looking to debug at this scale. The setup is time taking and the purify results will be a lot to monitor and analyze.
2. **Valgrind**: Let's accept that Valgrind is slow. We caught a memory corruption due to a race condition issue, which Valgrind would never have caught. There is a lot of memory record management in this tool. Though, it is useful during testing and should be executed periodically, we are really looking for a piece of code that can also be executed in a production environment.
3. **dmalloc**: This also requires recompiling the source code with dmalloc headers and libraries. Though we tried building it once, but it returned errors. Not tried again.

The tool also generates memory report for all the allocations done during a time period. The period may be defined between two specific intervals, or between two manual actions.

2 Components

1. **libdmem.so**: This is the code C-based shared library that has overridden functions. The glibc functions malloc, calloc, realloc, and free are overridden with customized routines to capture every new allocation and free from the process.

2. **MemoryAnalyzer.py**: This is a python script that is used to start/stop memory tracking and generate reports. Internally, it uses gdb, hexdump, and some other system utilities to get allocation information.

3 Protection Bytes

Using this tool, every memory allocation from the program start is guarded by protection bytes. Along with these bytes, there are also other useful and necessary information stored during every allocation. All this information is encapsulated with header and footer structures. The header one is allocated before the user memory pointer and the footer one after the user memory allocation. As of now, the total extra memory overhead is 32 (header) and 32 (footer) = 64 bytes for a typical malloc operation. The in-memory structures for these bytes are a little different for malloc and memalign calls, however, the base structure remains the same for both of them.

Below is a typical memory allocation using this tool:

...	Header	<User Memory>	Footer	... ...	Header	<User Memory>	Footer	...

The header structure contains all the necessary information to capture the memory related data. Additionally, it also maintains the memory allocation type.

```
typedef struct dmem_header_t
{
    char protection[8];         // MAGIC BYTES (8 bytes)
    char unused[8];             // Unused as of now (8 bytes)
    size_t alignment;           // Alignment, in case of memalign (8 bytes)
    size_t size;                // Size of allocated user memory (8 bytes)
}
```

1. *"protection"* contains MAGIC BYTES for header. Currently it is "**OKBOKBOK**" (8 bytes long).
2. *"alignment"* refers to the alignment value in case of memalign operation.
3. *"size"* denotes the size requested by the user.

The footer structure contains protection bytes and caller memory address information (helps in identifying memory leaks and getting statistics)

```
typedef struct dmem_footer_t
{
    char protection[8];         // MAGIC BYTES (8 bytes)
    size_t memcheck;            // Flag to indicate memory allocation is tracked (8 bytes)
    size_t tag;                 // Tag of the allocation (8 bytes)
    void *caller;               // Return address of the caller (8 bytes)
}
```

1. *"protection"* contains MAGIC BYTES for footer. Currently it is "**OKEOKEOK**" (8 bytes long).
2. *"memcheck"* variable if set to the size requested, also indicates that memory tracking is enabled.
3. *"tag"* indicates the current tag of the allocation. Value is 0 in case of no tag.

4. *"caller"* denotes the address of the caller from where the memory allocation function was called. Typically, this returns the return address of the current function.

4 Overridden Functions

4.1 malloc

For every malloc operation, an additional 32 bytes header and 32 bytes footer structures are also allocated. Internally, this function calls *__libc_malloc* to allocate memory. A typical malloc with the tool would look like:

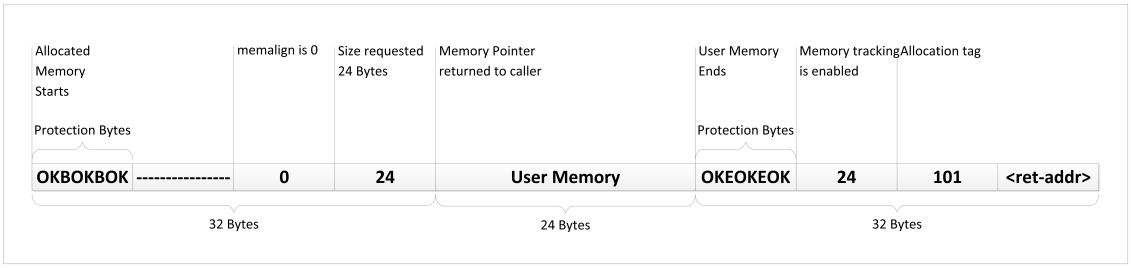

Figure 1: Typical Malloc Allocation

The alignment field will be 0 in case of malloc operation. 8 bytes in the header are unused and do not contain any relevant information.

4.2 calloc

Calloc operation is almost equivalent to malloc, with the exception of initialization. A typical memory layout from calloc would be:

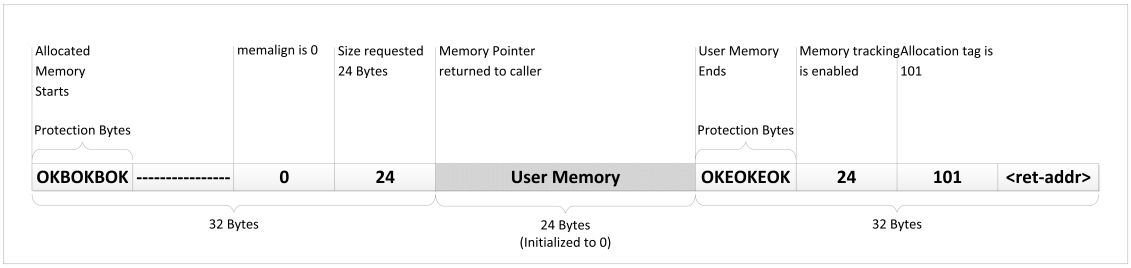

Figure 2: Typical Calloc Allocation

Note that only the user memory is initialized to 0.

4.3 realloc

Memory layout from realloc operations is equivalent to that of malloc operations.

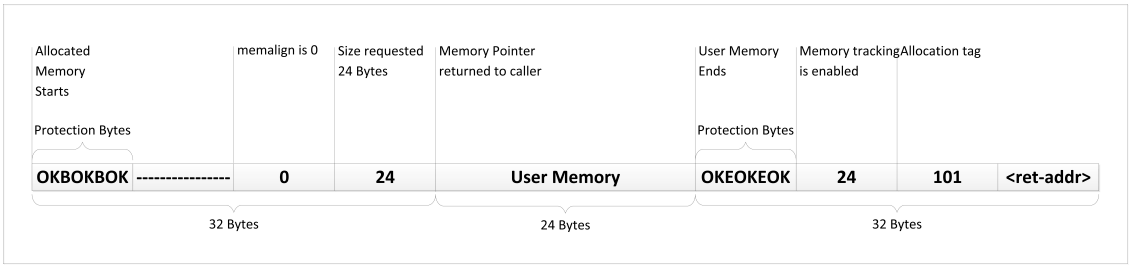

Figure 3: Typical Realloc Allocation

4.4 memalign

Memalign is tricky. Both the start pointer to allocated structure and the user pointer inside allocated structure must meet the alignment criteria. If the alignment is greater than 32 bytes, the size of header will be equal to that alignment instead of default 32 bytes. This is done to ensure that the user pointer returned to the caller is also aligned to that alignment. The *alignment* field of the header structure *lw_dcmd_header_t* contains the alignment of the allocated memory. The footer size remains the same (32 bytes). Internally, this function calls *__libc_memalign* for allocating aligned memory. Here is the memory layout after memalign operation.

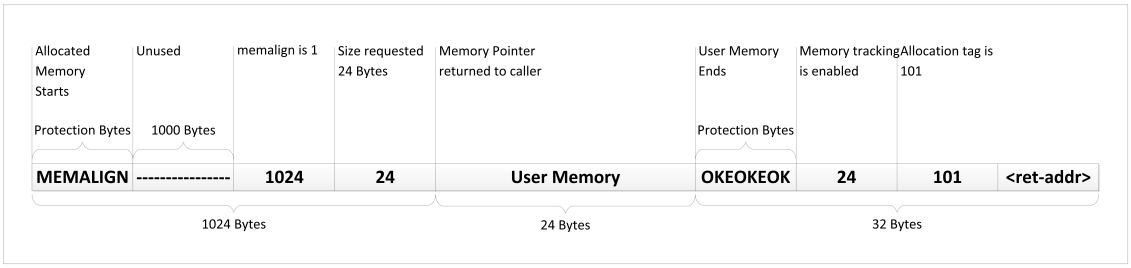

Figure 4: Typical Memalign Allocation

4.5 free

Free will free up the allocated memory. The header and footer allocations are validated for any corruption. If everything looks ok, only the protection bytes in header and footer are reset. Rest all the entries are not touched. *__libc_free* is used to free up the allocated space.

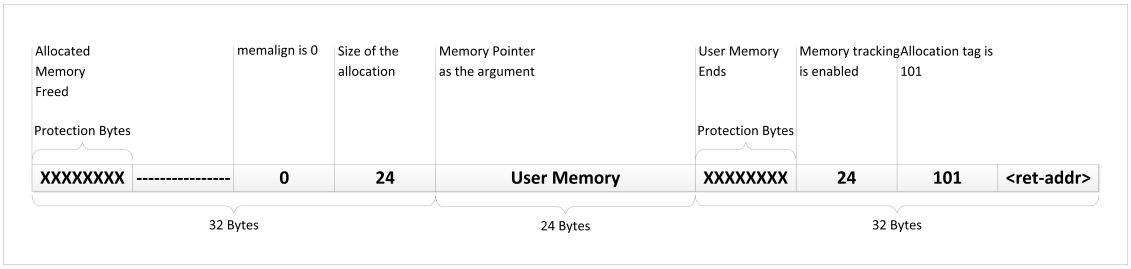

Figure 5: Typical Free Deallocation

5 Deployment

The library can be deployed along with any executable on Linux x86-64 systems. User needs to LD_PRELOAD the library *libdmem.so* before running the executable. Make sure there are no existing internal functions in the executable or associated library that override the GLIBC functions.

Upon execution the library will start customized allocations with its own headers and footers. You can verify allocations by attaching the process to GDB and examining the headers and footers.

6 Allocation Report

For all the allocations done using the library *libdmem.so*, you can get a customized allocation report indicating the source of all active allocations. A typical report would have the frequency of all active allocations from a specific address in the code. The table also differentiate between allocations of different sizes from the same memory address. You can also start tracking allocations manually with a customized tag and stop tracking anytime you want. If you run the utility to generate allocation report with that tag, all the current active allocations with that tag will be displayed in the report.

This information is specifically useful in diagnosing memory leaks. If the frequency from a specific address is much more than anticipated, that may indicate that allocations has not yet been freed inside the source code. Note that this utility does not provide all allocations and free till date.

6.1 Getting Report

The report can be obtained by running the utility *MemoryAnalyzer.py* with the core of the process as an argument. The utility will display tabular wise current active allocations in the process before the core was generated. The utility currently, as of now, only works on the core. If PID is given, it generate a gcore of the PID before processing information.

To start/stop tracking, use the PID of the process as an argument to this utility. This utility also checks for any memory corruption in the process. If found, it will report accordingly. Run the following command to generate reports from the utility:

```
# MemoryAnalyzer.py --execfile /usr/bin/random-daemon --corefile /var/core/daemon.core --generate-repor
OR
# MemoryAnalyzer.py --pid <PID> --generate report
```

A typical help for this utility is:

```
# MemoryAnalyzer.py -h
usage: MemoryAnalyzer.py [-h] [--execfile EXECFILE]
                         (--pid PID | --corefile COREFILE)
                         (--start-tracking | --stop-tracking | --generate-report)
                         [--tag TAG] [--count COUNT]

Get active memory allocations

optional arguments:
  -h, --help           show this help message and exit
  --execfile EXECFILE  Path to the executable file
  --pid PID            PID of the process
  --corefile COREFILE  Path to the core
  --start-tracking     Start tracking memory allocations
  --stop-tracking      End tracking memory allocations
```

```
--generate-report    Generate allocations report
--tag TAG            Tag to define a memory allocation
--count COUNT        Count of displayed allocations (decreasing order)
```

NOTE: The utility creates an additional directory *"mem_analyzer"* inside the current utility directory for internal processing. You may retain or delete the directory after the execution.

6.2 Internals

Here is the brief overview of the internal functionality of the utility.

6.2.1 Generating Reports

1. Uses GDB to get all heap regions. Typically uses 'info files' on the PID/core.
2. For all heap regions, this utility captures all the memory information in a binary format.
3. Uses 'hexdump' utility to search for tagged/customized allocations. Note that this is a much faster method to extract all allocations.
4. From the fetched allocations, gets the return address and processes counters.
5. Uses GDB to translate the return addresses into symbols.
6. Generates report using all this information.

6.2.2 Start/Stop Allocations

1. Uses GDB to suspend the process
2. Set the global variable *dmem_global_mem_check* to 1 or 0.
3. Resumes the process.

Note that this is safe since the variable is only used to indicate memory tracking and no operation, whatsoever, really causes any altered behavior due to this enable/disable operation.

6.3 Report Table

An active allocation report table currently contains the following fields:

1. *Address* contains the caller address from where the memory was allocated.
2. *Size* refers the requested size for the allocation.
3. *Count* is the frequency of such allocations.
4. *Symbol* is the translated function name (+ offset) for the *Address* in Column 1.

The report also displays the total time taken to generate the report.

A typical report looks like:

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Address         |  Size (Bytes) |       Count   |       Symbol
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
0x7f65ddef6efb  |       40      |       189376  |       CreateWorkItem+68
0x7f65dcc91579  |       24      |       2809    |       CRYPTO_malloc+73
0x7f65dd8fa034  |       32      |       786     |       tsearch+219
0x7f65d9675564  |       16      |       412     |       sqlite3MemMalloc+20
0x7f65d9675564  |       24      |       246     |       sqlite3MemMalloc+20
0x7f65dd86183a  |       101     |       154     |       add_module+663
0x7f65d9675564  |       128     |       106     |       sqlite3MemMalloc+20
0x7f65dd861457  |       33      |       82      |       add_alias+295
0x7f65dd861457  |       39      |       72      |       add_alias+295
0x7f65dd861457  |       40      |       65      |       add_alias+295
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

In the above report, the maximum count is for the 40 bytes memory that was allocated inside the function CreateWorkItem (instruction offset +68). The count is too high, given that work items need to be freed after work completion. This is indeed a potential case of memory leak where the work item is not getting freed after the work completion.

7 References

TBD