

Less Intrusive Memory Leak Detection inside Kernel

Jun Xu, Xiangrong Wang, Christopher Pham
IPD – ARF, Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134 USA
{junxu, xiangrow, chpham}@cisco.com

I. INTRODUCTION

Memory leak is a major resource issue which could lead to many system malfunctions and negative performance impacts [1]. A memory leak occurs when memory is not freed after use, or when the pointer to a memory allocation is deleted, rendering the memory no longer usable [2]. It can exhibit in many forms, contiguously or fragmentally, in flatten memory architecture or those with virtual space. Reckless use of dynamic memory allocation can lead to memory management problems, which cause performance degradation, unpredictable execution or crashes.

Various tools exist to detect memory leaks. These tools work by replacing generic memory functions in the library, such as `malloc()`, `free()` and other memory calls in C language. Each tool has code that intercepts calls to these functions during program execution and sets up logging information for each memory allocation/de-allocation request. Some tools can further implement memory protection fences to catch illegal memory accesses. However, the above techniques to analyze software memory always require source code instrumentation and some of the leak detection programs are very large and require a virtual memory image of the program being analyzed which makes it very difficult to use.

Based on the research of run-time kernel memory management and the assumption of memory leaks, the idea from the Contingency Analysis of Memory [3] appears to be a less intrusive method to track down memory leaks. The authors will first portray a common ground of memory leaks and present our algorithm. Then, an example will follow to identify the detection process. Finally, solutions to some common issues will be provided in the discussion section.

II. APPROACH

A. Assumption

In order to detect memory leaks, the authors assume

the following are always true:

- A leak is always a leak, i.e. it can not be freed by any process at any time;
- No process is using a leaked block;
- From the system point of view, leaked block is still valid.

Even though the memory usage is volatile in an active system, the leaked memory blocks will remain the state from the time when they become dangling. One typical purpose of the code instrumentation is to retrieve the memory usage information, e.g. the return value from the `malloc()` or the input to the `free()` functions. Normally the information is dynamically provided. However, from the above assumption, there is not such a requirement for dynamically retrieving the memory information about the leaked memory blocks. In this situation, the information can be provided by the system at later time without replacing `malloc()` or `free()`.

From the above assumption, the snapshots of the memory map would be considered as containing full status information to these leaked blocks, so the following algorithm can apply to scan the memory snapshot.

B. Description of the algorithm

All allocated memory should be referenced from memory areas in certain forms. In other words, the allocated memory blocks should have their pointers found in some other memory areas, which including the heap, data or bss, etc.. Since the allocated memory block information can be obtained from the system, authors can identify the true block pointer value from the normal pointer-like value.

If there is no reference to an allocated memory, that block is most likely leaked. (Some exceptions will be discussed later in Section 3). Logically, the concept can be extrapolated to that valid memory blocks should be successive from a few basic points in the format of referencing chains. Since the chains are built based on the successive memory reference, the authors refer them as the contingency chain. None of the allocated memory should be out of the chains, otherwise there is memory

leak. Similarly, if several blocks were chained together via a referencing pointer, but the whole chain is not addressable from any other pointers or the basic points, they should also be considered leaks. In reality, there could be many ways to implement the algorithm of forming the complete chain. For example, two basic methods are:

- Memory scan/search. Each valid memory region (e.g. heap) should be searched to form the contingency chain. Certainly there are many implementations for the searching methods. In general, this implementation could be slow but less intrusive.
- Dynamically build up the chains. This could be associated with memory access.

No matter which method will be implemented, there are several contingency chains to be built before proceeding to the memory leak identification. These chains are different from the kernel memory management chain, and will be used to compare to the memory management chain in order to detect memory leaks.

In the comparison of the contingency chains with the memory management chain, if a block is valid in the memory management chain while not in the contingency chains, it is considered as a candidate leaked block. Due to the volatility of memory usage, a revalidation is mandatory for confirming a potential leak as a positive warning.

C. An Illustrative Example

In Figure 1, there is a memory leak in the pseudo C code segment. Assume at line 3, `malloc()` successfully returned a block at address `0x1234` to variable 'a'. At line 4, the assignment of 'NULL' to variable 'a' make previous value in 'a' being overwritten and the allocated block is lost. Via the contingency analysis, it could find that there is no entry in the contingency chain referencing address

```
1  foo {
2      void *a;
3      a = malloc(sizeof(100));
4      a = NULL;
5  }
```

Figure 1. Pseudo C code with memory leak

`0x1234`. While the block at address `0x1234` is still valid from the memory management list, so the block is marked as a leak.

III. DISCUSSION

Normally, the above algorithm can effectively report memory leaks inside kernel space, though some tasks are CPU intensive. However, compared to lack of methods for the detection of memory leaks inside kernel, the

described algorithm provides an effective and reasonable detection at moderate costs.

In the corner case where memory block just chains into a closed loop, since there is no root and external reference, the algorithm can also be used for the detection of such memory leaks. Furthermore, the comparison of the memory management list and the contingency chain could also reveal potential dereferencing of dangling pointers, which point to the freed memory blocks.

Like the conservative garbage collection [4], false positive warnings are inevitable and might be reported. Especially certain coding style may confuse the algorithm, thus cause the false positive warnings. In case the difference between the false positive warning and the positive warning is not clear, a limited stress testing can clarify the warnings. Generally, the number of positive warned memory blocks will consistently increase while false positive warnings might not exhibit such characteristics.

IV. CONCLUSIONS

From our initial experiment, we proved the algorithm is generally useful for detecting memory leaks in the kernel of operating systems. The algorithm also has the following unique features:

- No code or library instrumentation or replacement;
- Memory Contingency Analyses (explained below) is used to find "no referenced valid block";
- Convert memory mapping from dynamic to static without compromise in accuracy;
- Granularity is down to each Program Counter.
- With a little more effort, the algorithm can be easily expanded to customized memory management structure as well.

REFERENCES

- [1] Pham, Xu, Zhou, "Source Code Analysis As A Cost Effective Practice to Improve the High Reliability of Cisco Products", Supplemental Proceedings, ISSRE2001, pp 144-152.
- [2] Cal Erickson, Memory Leak Detection in Embedded Systems, Linux Journal, Issue 101, Sep. 1, 2002.
- [3] M. Kahana, Endel Tulving, Fergus I. M. Craik, "The Oxford Handbook of Memory", Chapter4, Oxford University Press; ISBN: 0195122658; 1st edition (May 2000).
- [4] Hans-Juergen Boehm, "Space-efficient conservative garbage collection." In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, pages 197-206.