The Easy-Portable Method of Illegal Memory Access Errors Detection for Embedded Computing Systems

Ekaterina Gorelkina
SRC Moscow, Samsung Electronics
e.gorelkina@samsung.com

Sergey Grekhov
SRC Moscow, Samsung Electronics
grekhov.s@samsung.com

Alexey Gerenkov
SRC Moscow, Samsung Electronics
a.gerenkov@samsung.com

Abstract

Nowadays applications on embedded systems become more and more complex and require more effective facilities for debugging, particularly, for detecting memory access errors. Existing tools usually have strong dependence on the architecture of processors that makes its usage difficult due to big variety of types of CPUs. In this paper an easy-portable solution of problem of heap memory overflow errors detection is suggested. The proposed technique uses substitution of standard allocation functions for creating additional memory regions (so called *red zones*) for detecting overflows and intercepting of page faulting mechanism for tracking memory accesses. Tests have shown that this approach allows detecting illegal memory access errors in heap with sufficient precision. Besides, it has a small processor-dependent part that makes this method easy-portable for embedded systems which have big variety of types of processors.

1 Introduction

Currently software programs running on modern embedded computing systems have quite complicated behavior in sense of memory manipulation that make the process of debugging very time consuming. In order to simplify debugging procedure, various helper utilities were designed. The most known are Valgrind [1] and using `MALLOC_CHECK_` environment variable for GNU C library [2].

Valgrind replaces the standard C memory allocator with a custom implementation and inserts extra instrumentation code around almost all instructions, which enables to detect read and write errors when a program access memory outside of an allocated block by a small amount. This approach has a strong dependency on the type of processor of computing system. This makes usage of Valgrind on different computing systems quite difficult due to relatively complicated porting procedure.

Recent versions of GNU C library are tunable via environment variables. Setting variable `MALLOC_CHECK_` to values 1, 2 or 3 allows detecting simple memory errors like double free or overrun of a single byte. The drawback of this approach is low precision of error detection and performance degradation which is caused by using less efficient implementations of allocating functions.

In contrast to [1] and [2], the proposed method solves the problem of memory errors detection with sufficient precision and has a small processor-dependent part which minimizes the time porting of this method on various processor architectures (that is especially valuable for embedded systems). It also does not require program re-compilation. As a result, this method can detect heap object overflow problem for wide range of processors of embedded computing systems. Testing results also have shown the low execution overhead of suggested technique in comparison to Valgrind tool.

2 General Description on Idea

Suggested method relates to detection of typical memory errors: heap object overflow, heap object underflow, access to un-allocated memory.

The basic idea of the method is to intercept memory allocation requests, add special region (red zone) to each heap object, protect allocated memory for reading and

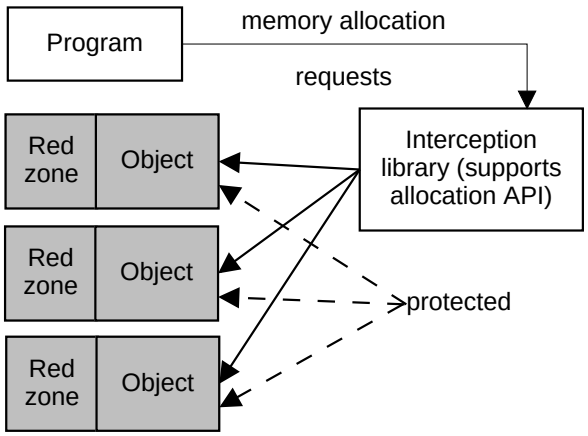

Figure 1: Allocation requests of considering program are intercepted by a special library which allocates each object with red zone and protect them both from reading and writing

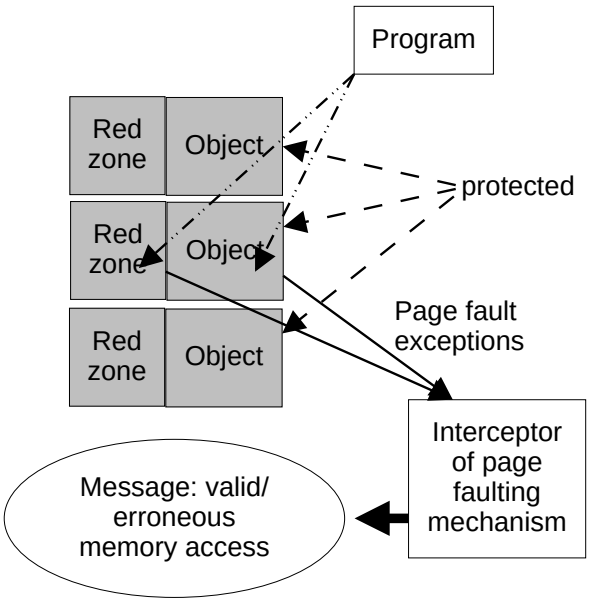

Figure 2: Considering program accesses protected objects or its red zone and generate page fault exception; analyzing parameters of this exception allows detecting overflow errors

writing and use mechanism of handling incorrect memory accesses in address space (here and after page faulting mechanism) in order to track accesses to the memory and detect the erroneous ones. The principal scheme of two main parts of this idea - intercepting allocation functions and intercepting page faulting mechanism - are illustrated in Figure 1 and Figure 2 respectively.

The description of Figure 1 is following. During program execution it requests and releases dynamically allocated memory. The special interception library handles these requests and processes them by making allocations, adding a special region (red zone) and protecting both of them from reading and writing. These red zones are used for detecting typical overflow errors (as it will be explained further in details).

The description of Figure 2 is following. After allocating necessary memory program tries to access some of created objects. Since all of them are protected from reading and writing, such attempt will generate page fault exception. It is a special event which occurs when specified memory can not be read or written. The parameters of this event include the address of access and the type of access (read/write). Thus, such accesses can be classified into valid and invalid by comparing the address of access with addresses of allocated heap objects.

After handling page fault exception the protection of the accessed object should be annulled and the program execution should be resumed (if memory access was valid) or stopped (if memory access was erroneous).

2.1 Interception of Allocation and De-allocation Requests

As it was mentioned earlier, at run-time all memory allocation requests are intercepted by a special library. Herewith, it is allocated a memory region of requested size plus additional memory region (red zone).

Figure 3 demonstrates the difference between original and intercepted allocation request. In the first case the allocation request is processed by allocation library and object is created. In the second case the allocation request is processed by interception library. The result of processing is the requested object plus red zone (a special memory region for detecting errors). Herewith, both allocated object and red zone are protected from reading and writing.

If program uses, for example, malloc() and free() functions from standard GNU C library for allocating memory, then invocation of these function can be substituted in Linux by `special_malloc()` and `special_free()` using `LD_PRELOAD` environment variable. These special functions create/release heap objects with additionally allocated red zone and protect them from reading and writing using mprotect() function.

Since, in general case, memory can be protected only page by page, object will be allocated within one or

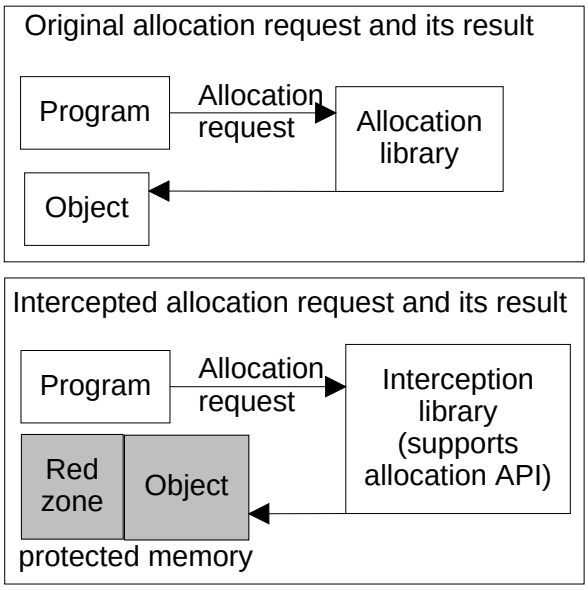

Figure 3: The difference between original allocation request and intercepted allocation request: in the second case the red zones are added to allocated objects and both regions are protected from reading and writing

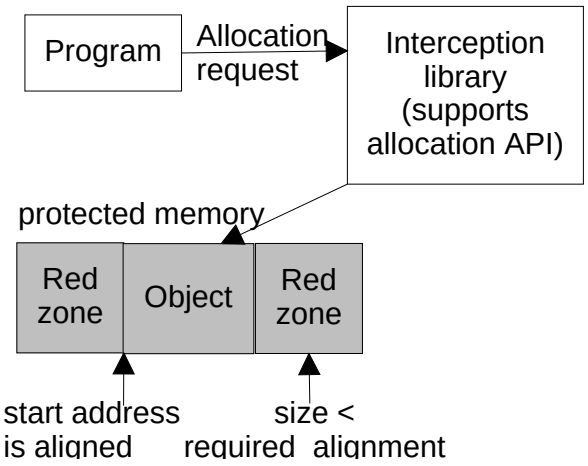

Figure 4: Due to restrictions for malloc() return value there can be two red zones: before and after specially allocated heap object

more virtually continuous memory pages. It can be easily seen that when size of heap object is multiple of page size then the size of corresponding red zone will be zero (in this situation it will be impossible to detect illegal access). This issue can be resolved the following way. Let's denote the size of one memory page in computing system as P and the size of the requested allocation as S. Then N - the number of pages required for special allocation - should satisfy the two following conditions: $N*P \geq S+P/4$ and $(N-1)*P < S+P/4$. The additional component P/4 prevents the situation when S is a multiple of P and red zone size is zero. Thus, the final formula for calculating N is following:

$$N = (S+P/4)/P + sign[(S+P/4) mod P] \quad (1)$$

where all mathematical operations are integer-value, *sign* means the function which return 0 is arguments is 0 or 1 if argument is greater than 0, and *modP* means function which return the residue of division of argument by P.

Another issue when processing allocation request is the alignment of returned memory. By default, the malloc() function from GNU C library returns the align address of allocated memory. This is caused by specific requirements of returned address to be suitable for any kind of variable. Thus, in fact, the address of each specially created heap object should be aligned (for example, to 4 bytes on ARM processor) and there will be two red zones: before and after the heap object. The size of second red zone (after heap object) varies on the size of heap object and required alignment (see Figure 4).

As it was mentioned in chapter II, heap object is unprotected after processing page fault exception and detecting a valid memory access. Moreover, the application execution is resumed. Thus, if application accessed unprotected red zone, then it is impossible to detect memory access by intercepting page faulting mechanism. The solution of this problem is following. After the allocation of red zones, they should be filled with identical values - stamps - which are used for detecting write accesses. If application wrote to red zone then most probably the stamps are changed. Checking the consistency of stamp before freeing the memory enhances the precision of overflow detection. This procedure can be implemented during the processing of intercepted of deallocation requests. Let us note that this solution can not detect read accesses to unprotected memory. This issue is unresolved within the scope of proposed method.

2.2 About Interception of Page Faulting Mechanism

Page faulting mechanism is a part of operating system. The typical approach for intercepting of its functions is using dynamic instrumentation tools. Well-known Systemtap dynamic instrumentation tool based

on Kprobe [3] allows creating handlers for intercepted kernel functions, however it can not track the calls of `do_page_fault()` function because of restrictions of Kprobe. Another well-known tool LTTng [4] uses static instrumentation and, thus, requires kernel recompilation that can be quite inconvenient in case of embedded systems. Therefore authors used dynamic instrumentation tool SWAP [5] developed in Samsung Research Center in Moscow.

This tool allows at runtime obtaining arguments of interception kernel functions and performing custom actions before executing the body of intercepted function. Particularly, it is possible to intercept `do_page_fault()` function and obtain the address of accessed memory and the type of access (read o write).

Let us note that page fault exceptions occur in normal situations and are needed for valid system functioning. Therefore, the procedure of processing of accesses to protected heap objects should not damage the original handling of page faults in kernel.

The interception mechanism can be implemented as a kernel module. The information about allocated heap objects can be transferred from the interception library via /proc or /dev filesystem.

2.3 Handling Accesses to Protected Heap Objects

According to the general idea of the method, the page faulting mechanism should be intercepted for detecting access to allocated and protected heap objects, obtaining the parameters of this access (address and type) and classifying this access as valid or erroneous. As it was written in previous chapter, the `do_page_fault()` function can be intercepted and custom actions can be performed before executing its body. These actions relates to classifying of detected memory access and resuming or stopping execution of considering application depending on the classification results.

The algorithm of classification is following.

Step 1. If address of accessed memory is inside allocated heap object, then: previously unprotected object is protected (if it exists); protection of this object is annulled; program execution is resumed (see Figure **??**); otherwise go to step Step 2

Step 2. If address of accessed memory is inside red zone of allocated heap object, then protection of this object and unprotected remains unchanged, message about incorrect memory access is reported, and program execution is stopped (see Figure 6); otherwise go to step Step 3.

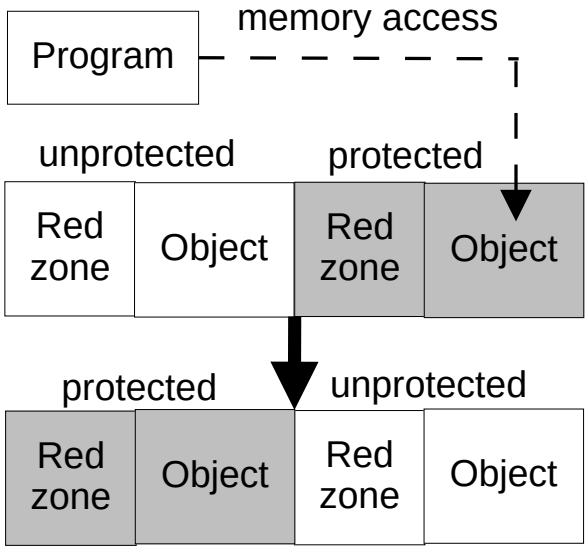

Figure 5: Step 1 of Algorithm 1: if address of accessed memory is inside a valid heap object then unprotect it and its red zones

Step 3. If address of accessed memory is outside of any allocated heap object and red zones but inside a valid allocated memory region (for example, stack, read-only data or executable code) then protection of heap objects remains the same and program execution is resumed (see Figure 7); otherwise go to step Step 4.

Step 4. If address of accessed memory is outside of any allocated heap object and corresponding red zone and any valid memory region, then error message about access to unallocated memory is created, and program execution is stopped (see Figure 8)

The error message created on Steps 2 and 4 of Algorithm 1 consists of the address of illegally accessed memory and the type of access, the title of the binary object that made illegal access, the address of instruction which caused the error (within the binary). This information can be used for creating a user-friendly report about occurred memory access problem: debug information together with mentioned content of error message allows displaying the line of the source code which produced error.

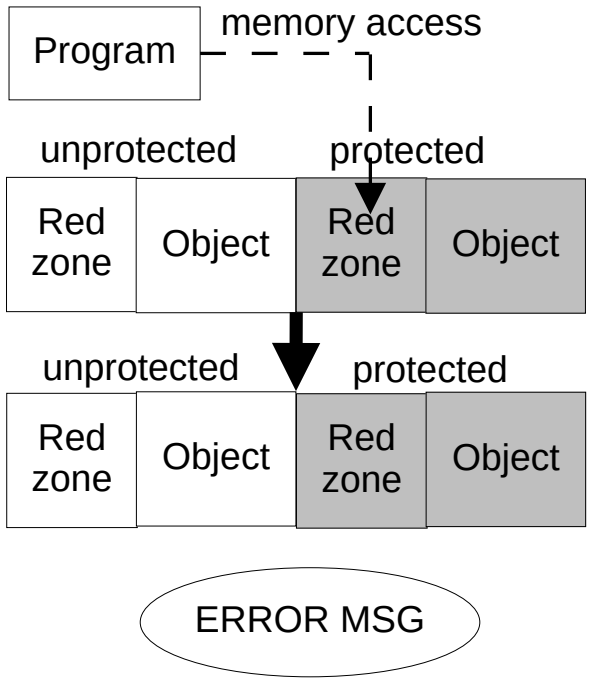

Figure 6: Step 2 of Algorithm 1: if address of accessed memory is inside a red zone then keep current objects' protection, report about error and stop execution of the program

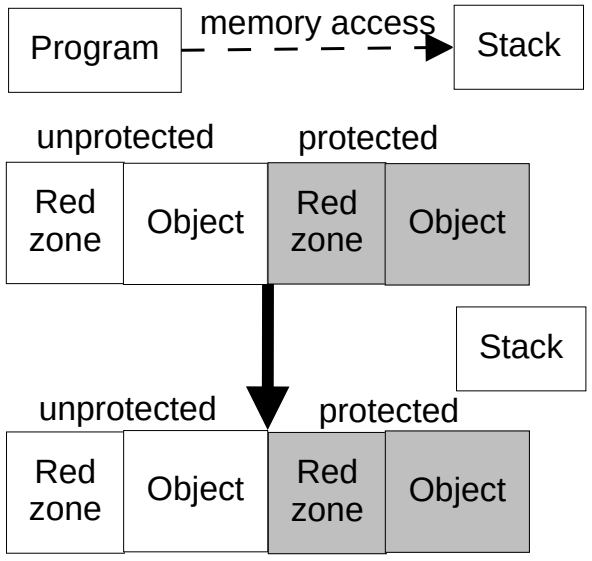

Figure 7: Step 3 of Algorithm 1: if address of accessed memory is outside of heap, but inside a valid memory region (e.g. stack) then keep current heap objects' protection and resume program execution

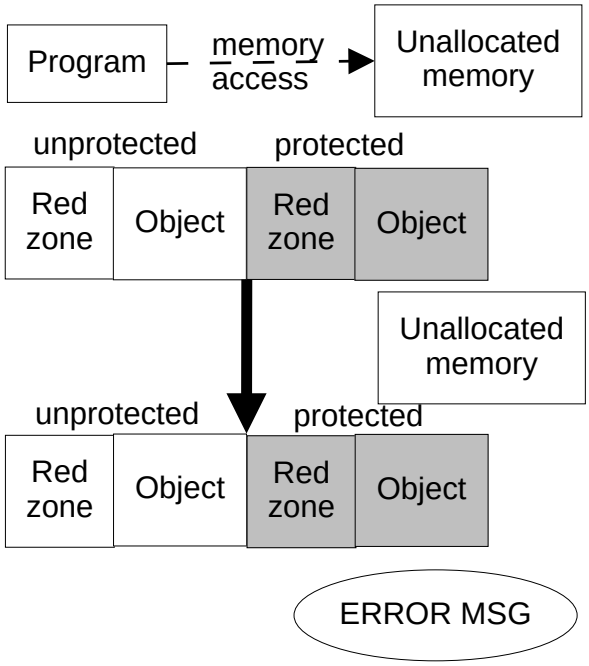

Figure 8: Step 4 of Algorithm 1: if address of accessed memory is outside of heap and any valid memory region then keep current heap objects' protection, report about error and stop program execution

3 Method Advantages and Drawbacks

The first main advantages of suggested method of memory overflow errors detection is its fast portability to various processor architectures. As it can be seen from previous chapters, the processor-dependent part of the method includes only obtaining of address of accessed memory by interception of page faulting mechanism. Thus, in comparison to the methods which use interpreting of binary instructions, the suggested technique can be easily moved across various processors. This is especially valuable for developer of embedded systems.

Since the suggested method uses dynamic instrumentation engine SWAP, the two following advantages are obtained:

- developer should not recompile erroneous application before starting the search of memory errors
- suggested method does not need debug information for the program running on embedded system.

As it can be seen from the method description, the program which produces memory errors runs on embedded

system and is not changed during memory errors detection.

However, there exist several drawbacks. Firstly, method can not track accesses to the unprotected red zone with help of page faulting mechanism. Thus, read operation which accesses the unprotected red zone will be never detected. The write operation which accesses the unprotected red zone can be partially detected by using stamps. At the same time, this disadvantage has a positive aspect. Since memory accesses made within the unprotected object are not tracked, the execution of the program is going faster. This is also valuable for embedded systems.

Two more drawbacks are common for most of tools of overflow detection:

- it is impossible to detect the out-of-bounds situation when application accesses a valid memory region.
- it is impossible to detect the out-of-bounds situation within the allocated structure or class

4 Testing Results

The environment of all tests was following:

- Nvidia Tegra board, ARM-based, kernel 2.6.29
- Beagle board, ARM-based, kernel 2.6.33

During testing it was discovered that both target boards have similar behavior and produce identical results for test applications. Thus, authors provide testing results without specifying the particular environment implying that they are equal for both embedded devices.

The proposed method was tested on artificial test applications. The test applications have similar structure: three heap objects of size 4092, 4096 and 4100 respectively are allocated. During each separate test some particular out-of-bounds situations are created. For example, access the first object, and after that access the red zone of the next object. The following Table 1 provides the results of testing in comparison to Valgrind.

As it can be seen, the accuracy of suggested method is worse in comparison to Valgrind. However, it is still quite high and method can be used for debugging.

Tool	Total number of tests	Passed tests (accuracy)
Valgrind	192	144 (75%)
Proposed method	192	114 (59%)

Table 1: The accuracy of detecting errors for artificial test applications

Test environment	Time of execution (seconds/degradation degree)
Test application without memory checking tool	0.0025 / Not available
Test application with memory checking tool based on proposed method	0.466 / 186.4
Test application with memory checking tool Valgrind	2.1283 / 851.32

Table 2: Time of execution of test application with and without memory checking tools

The goal of the next test was to measure the overall degradation in performance when executing test application with memory checking tools. The test application allocates 5 arrays of 256 bytes and accesses sequentially each of them in the cycle. The average degradation of performance of test application is shown in Table 2.

It can be seen from the table, the method proposed in this article shows better performance. This advantage compensates the worse accuracy of error detection.

The following Table 3 provides information about architecture-dependent code of implementation of proposed method in comparison to Valgrind.

Tool	Total number of lines of code	Total number of architecture-dependent code (fraction)
Valgrind	> 900000	> 170000 (19 %)
Proposed method	> 180000	> 5000 (3 %)

Table 3: The amount of code dependent on the architecture of CPU.

Tool	Result of detection
Valgrind	Detected, up to specifying the line of source code which produced the error
Proposed method	Detected, up to specifying the line of source code which produced the error

Table 4: Detecting memory access error for bug in glib-2.15

It can be seen that Valgrind has bigger processor-dependent part that makes the procedure of its porting more difficult. In comparison the method proposed in the article has smaller part which depends on type of CPU, thus it is easier to be ported on embedded systems with different processors.

The last test consisted of detecting an error for the bug from glib-2.15 library bug tracking system (see [6] for more details). The erroneous behavior of library was produced when insufficient amount of memory was allocated for saving binary data encoded to base64 text form. As a result, accessing beyond the end of array caused segmentation fault exception. The error was fully detected by the proposed method with indication the line of the source code which produced the error. Table 4 shows the results of testing for Valgrind and proposed method.

Both tools produced identical results by detecting the error and specifying the line of code which produced this error.

5 Conclusion

In this paper authors proposed a method for heap overflow memory errors detection which is easy-portable on various processors' architectures of computing systems. The method shows sufficiently good results in memory overflow errors detection. Despite of some of its drawbacks, it can be efficiently used on embedded systems.

6 Acknowledgments

Authors would like to thank Jaehoon Jeong, senior engineer of System S/W Group in Samsung Electronics, and Hyunju Ahn, senior engineer of System S/W Group in Samsung Electronics, for fruitful and extensive discussion on the proposed approach and Mikhail P. Levin, the head of Advanced Software Group of Samsung Research Center in Moscow, for assistance in preparation of the article.

References

[1] Valgrind project (`http://valgrind.org/`)

[2] Malloc manual page (`http://www.kernel.org/doc/man-pages/online/pages/man3/malloc.3.html`)

[3] Systemtap tool (`http://sourceware.org/systemtap/`)

[4] Alexey A. Gerenkov, Ekaterina A. Gorelkina, Sergey S. Grekhov, Sergey Yu. Dianov, Jaehoon Jeong, Oleksiy Kokachev, Leonid V. Komkov, Sang Bae Lee, Mikhail P. Levin, System-wide analyzer of performance: Performance analysis of multi-core computing systems with limited resources, IEEE EUROCON 2009, pp. 1299-1304

[5] Mathieu Desnoyers, Michel R. Dagenais, LTTng: Filling the Gap Between Kernel Instrumentation and a Widely Usable Kernel Tracer, LFCS'2009

[6] Red Hat bug tracking system (`https://bugzilla.redhat.com/show_bug.cgi?id=474770`)