2016 International Conference on Engineering and Telecommunication

Various buffer overflow detection means for Elbrus microprocessors

Roman M. Rusiaev[1], Murad I. Neiman-zade[1,2], Alexandr V. Ermolitsky[1], Valery I. Perekatov[1,2], Vladimir Yu. Volkonsky[1]

[1]AO "MCST"
Moscow, Russia

[2]Moscow Institute of Physics and Technology (MIPT)
Moscow, Russia

{Roman.M.Rusiaev, muradnz, era, perekatov, vol}@mcst.ru

***Abstract* - A brief review of Protected Execution Mode (PEM) for user-space applications featured in Elbrus architecture is described first. Then, AddressSanitizer, a well-known utility by Google Inc, is considered as an example of a pure software technique of memory control. Comparative analysis of these solutions is given with performance flaws, applicability and boundary violation detection quality.**

Keywords – Elbrus architecture; protected mode; AddressSanitizer; buffer overflow errors

I. INTRODUCTION

Every C/C++ programmer inevitably encounters memory/objects boundary violation errors that become a really annoying problem in large projects. Violated memory fragment could be global/static data, available throughout the entire application execution time, or a part of heap space, that is, dynamically allocated and relented via operating system, and it also could be a bulky local variable allocated in user stack via single instruction. All boundary violation errors, related to any of the mentioned memory fragments, are called Buffer Overflow errors here. Elbrus microarchitecture suggests several means of boundary violation detection, like PEM with full hardware support, or applying traditional software solutions represented by AdressSanitizer in this review.

II. HARDWARE SUPPORT FOR PROTECTED EXECUTION MODE IN MICROPROCESSORS OF ELBRUS FAMILY

PEM is based on several concepts that are reviewed in brief below.

A. Strictly typed pointers

Descriptor is the key notion for PEM. It is a pair of data and type specificators, where the latter is implemented in hardware as a few additional bits (tags) for every small portion of memory or register. The available types include:

- Procedure Label – its value contains pointer-to-procedure entry point;
- Array Pointer – a structure with 3 fields: "base address", "size", "offset"; only "offset" field is available for modification in non-privileged execution;
- Stack Array Pointer — same as array pointer with the additional field of "procedure stack level".

There are more types available, such as Number, Diagnostic value, Uninitialized value, etc.

Pointer-type descriptors comply with the following restrictions:

- address arithmetic is performed with special instructions; regular arithmetic facing non-numeric data will throw exception; the same happens for pointer arithmetic and non-pointer data
- if offset is less than zero or greater than field size, exception is thrown

There are restrictions on tag bits as well:

- data is transferred along with tags, from register/memory to register/memory
- tags cannot be modified in arbitrary way, some transformations are allowed (pointer-to-numeric), some are prohibited (numeric-to-pointer)

B. Module separation

Module is an independent executable part of a project. It could be either a statically linked program, or a dynamically linked shared library with position-independent code. Module Context is defined as a tuple of module Code and module Data, where Data is understood as a set of objects with lifetime equal to one of the project executables. On the project startup, all shared libraries are independent Modules each having its own Context.

In PEM, each module is described by a pair of special registers pointing to the Code and Data sections. These registers are used in hardware boundary control verifying that module Code and Data are never accessed outside their boundaries. Register for Data is called Global Descriptor (GD), whereas register for module Code is called Compilation Unit

978-1-5090-4553-2/16 $31.00 © 2016 IEEE
DOI 10.1109/EnT.2016.32

Descriptor (CUD). When CALL instruction is executed, destination address is checked on CUD boundaries by hardware; if callee address is outside CUD area, both CUD and GD are automatically changed to the Context of callee module; if callee is outside all CUD areas, exception will be thrown.

It should be also mentioned that Elbrus architecture stores return IP in a separate stack rather than in user stack; it is called a “chain stack”. The chain stack is inaccessible with unprivileged load/store operations, that makes it impossible to overwrite return address or perform any of the known ROP attacks, thus greatly increasing security. Interaction of return and call operations with the chain stack (IP pop and push, respectively) is performed automatically by hardware.

III. HARDWARE CONTROL OF BUFFER OVERFLOW IN PEM

Extended information stored in Pointer Descriptor is the key distinction of data addressing in PEM, since it holds all necessary information to perform full and exact checks of boundary violation of every object access. PEM allows memory accessing only via Pointer Descriptors, all operations that obtain and modify Pointer Descriptors are strictly regulated.

The only ways of building a new descriptor are:

- demanding a part of user stack within procedure stack window bounds,
- narrowing the existing Pointer Descriptor bounds down
- requesting address of global object via current GD
- performing system call to return Pointer Descriptor to a new part of heap space.

Each of the ways is either performed under hardware control (the first three) or by Operating System (the forth), and OS is building the new descriptor in privileged mode. Also, as mentioned earlier, the only part of Pointer Descriptor that could be modified in unprivileged mode is “offset”. As a result, every Pointer Descriptor stores the exact bounds of memory area/object, which are protected from unprivileged modifications.

Any access via Pointer Descriptor outside its bounds causes exception, and this is true for any objects; but for stack objects, there is an additional control based on “procedure stack level” field in Stack Pointer Descriptor. This field stores an integer equal to the user stack depth of the procedure that has created this object, i.e. _init has psl == 0, psl(callee) = psl(caller) + 1. There is an additional restriction imposed by PEM on stack objects, which is as follows: any Stack Pointer Descriptor should never exist when creating procedure returned control. As a result, we need to control any attempts to write Stack Pointer Descriptor in memory allowing it to be stored in the current stack frame to be transferred to callee as a parameter, but not allowing it to be written in the frame of caller. Hardware control ensures throwing exception, when we execute

ST pdescr_stackobject → [pdescr_where]

if psl(pdescr_stackobject) < psl(pdescr_where)

Storing of Stack Pointer Descriptor in global memory or in heap is controlled by a special technique called the “last wish” that requires intervention of operating system. This technique is like Garbage collection; it allows storing of Stack Pointer Descriptor until procedure is returned control, but marks these occasions with a “last wish” flag in the chain stack frame of procedure, and then performs privileged allocated memory scan for undeleted Descriptors.

IV. SOFTWARE BUFFER OVERFLOW CONTROL

The AddressSanitizer utility [1, 2] initially developed as a part of the LLVM project and later in the GCC compiler [3] was also implemented in optimizing compiler for the Elbrus architecture. AddressSanitizer consists of two parts: the first is a part of the compiler, and the second is a compiler-rt library [4]. The basic idea is to instrument the code of user application at compile time - Sanitizer inserts the code that performs dynamic checks during execution. Also, the library replaces some standard functions by their equivalents, which check correctness of memory usage. If error is detected, a special library function is called to print a detailed report about the error in the user program.

Let us give some definitions that will be required below. A "red zone" is the area of memory that is necessary for control of borders of the allocated user memory. The memory that was not requested by the user application is called "poisoned".

Virtual address space of the application is split into three zones: the first contains addresses available to the user, the second (the so-called "shadow memory") stores metadata for the first zone, the third is not available for user applications and represents a purely technical solution to prevent user application access to the shadow memory. There is conformity between the elements of the first two zones - every 8 bytes of the first zone correspond to 1 byte of the second one, i.e. values in the shadow memory describe the state of user memory. There are the following options:

- A zero value of the shadow memory byte means that all 8 bytes of the corresponding user memory are unpoisoned.
- A negative value of the shadow memory byte means that all 8 bytes of the corresponding user memory are poisoned.
- A positive value k of the shadow memory byte means that the first k bytes of the corresponding user memory are unpoisoned, and the rest $8-k$ bytes are poisoned.

Let us consider a short version of the AddressSanitizer algorithm. At compile time for each memory access operation in user application compiler inserts an additional code that reads a corresponding byte k of the shadow memory and then

verifies its values. If *k* is positive, the additional code is executed to verify that the last byte of the accessed memory is unpoisoned. If the last condition is not met or *k* is negative, a special function from the compiler-rt library is called to report error in application.

Error detecting methods differ for various types of memory. In the case of dynamic memory, border control is performed by means of AddressSanitizer library. All functions allocating and deallocating dynamic memory, such as *malloc* and *free*, are replaced with equivalent versions provided by the library. Any dynamic memory allocation request leads to formation of user memory surrounded by poisoned red zones. Access to dynamic memory outside of allocated areas, i.e. to the red zones, leads to a function call to report heap overflow. When dynamic memory is released, the corresponding shadow memory is poisoned by special values, thus "use-after-free" errors can be detected.

In the case of global memory, border control is performed by means of AddressSanitizer compiler tool and library. All globals are complemented by the red zones. When application or shared library starts, a special library function is called to register globals and to poison their red zones. When shared library is unloaded, the red zones of the corresponding globals are unpoisoned. Since each memory access operation is checked for its correctness, access to the red zone of any global will lead to the termination of the application with error message.

Finally, in the case of stack memory, border control is performed by means of compiler tool. All stack objects are complemented by the red area. Compiler adds code that poisons respective red zones of the stack of objects. Access to memory outside of stack object leads to call of function reporting stack overflow error. When control returns from the function, all stack red zones are unpoisoned.

V. COMPARATIVE ANALYSIS OF PEM AND ADDRESS SANITIZER METHODS

Strict typing rules that PEM imposes on applications are the main limitations to its applicability. PEM Pointer Descriptor cannot be obtained from regular integer-like C/C++ pointer, because there is no information about the corresponding object. There is a special instruction to cast from pointer to integer type, but if integer value is converted to pointer, then any memory access through this pointer will generate exception.

Regarding software detection of memory border violation with AddressSanitizer, the following issues should be noted.

First, it should be noted that when instrumenting user application with AddressSanitizer during each memory access operation, a code is generated, which verifies memory border correctness. This leads to a significant increase of the executable size and decrease of the application performance. Moreover, memory requirements for user application significantly grow due to the creation of the red zones for all objects.

Another flaw of using the red zones is inability to cover the entire user address space. For example, if a red zone for some object is 32 bytes, and memory access relative to its location exceeds this value, it is impossible to detect such a violation by means of AddressSanitizer, whereas program execution in protected mode leads to exception due to violation of memory borders.

Another significant drawback of AddressSanitizer is inability to detect errors of memory access on the address border that correspond to two adjacent bytes of the shadow memory. The first shadow byte contains information that the address described by its value is defined as unpoisoned, while the second byte indicates that the corresponding address is poisoned. This situation is illustrated by the following example in C:

```
...
char a[10]; /* suppose that address of “a” is aligned by 8 */

int *p = (int *)&a[7];
...
/* somewhere here the array “a” should be initialized */
...
return *p; /* here we load 4 bytes and a byte of them is not valid */
```

User error here is reading an address pointed by the variable “p”. The first three bytes read will be in the assigned user stack memory, but the last byte will be outside of the allocated stack memory border. On memory border check, the address located in “p” variable will be mapped to the shadow memory, it will correspond to that byte in the shadow memory, which contains a zero value. Thus, this error will remain unnoticed. During execution of the program with that code in protected mode, the described problem will be detected due to the use of descriptors, which have exact information about the size of the memory available to user.

VI. PROTECTED MODE AND ADDRESS SANITIZER PERFORMANCE COMPARISON

As noted, instrumenting a user application with AddressSanitizer leads to a significant code size grow and slowdown of execution time of an instrumented application. The instrumented code that checks user memory borders also negatively affects compilation speed. Below we present the results of performance measurement by compilation and execution times for several benchmarks from the «Spec cpu2000» bundle [5]. The measurement was made on PC-401 computer with Elbrus-4C 750 MHz processor. Benchmarks were compiled with –O2 optimization level. Compilation of the tasks was made for protected execution and instrumented with AddressSanitizer. Table 1 shows compilation and execution time ratios for the two modes.

TABLE I. Ratio between AddressSanitizer and the protected mode compile/execution times

Benchmark name	Compile time ratio	Execution time ratio
164.gzip	1.846	2.258
256.bzip2	2.492	5.796
179.art	2.141	3.078
183.equake	2.761	2.613
188.ammp	1.908	1.690
gmean	2.113	2.096
avg	1.852	2.570

As shown in the Table, the task compilation time instrumented with AddressSanitizer is 1.85x bigger in average than that of the protected mode, and the execution time of the instrumented tasks is 2.57x bigger in average.

VII. CONCLUSION

Both approaches have pros and cons. The protected mode of Elbrus microprocessors detects buffer overflow errors regardless of the offset value; all checks are performed by hardware without allocation of red zones, and thus have a small overhead. In addition to the described mechanisms, the protected mode can detect other errors, such as using uninitialized data or "wild pointers", but this is beyond the scope of this article. At the same time, AddressSanitizer supports detection of other types of memory access errors, such as use-after-free or memory leaks (using LeakSanitizer extention). Also, AddressSanitizer provides detailed reports about errors, which makes debugging easier.

Initially, Elbrus protected mode was developed as a means of protection, but not detection of errors, thus it lacks detailed reports, however basically this can be done at the software level. The protected mode imposes serious requirements in terms of casting integers to pointers, which limits its applicability. On the other hand, AddressSanitizer has a significant performance penalty that makes it unusable for production version of applications.

Further work will be targeted at studying a broader class of memory access errors and developing tools for the Elbrus architecture.

REFERENCES

[1] http://clang.llvm.org/docs/AddressSanitizer.html
[2] https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm
[3] http://gcc.gnu.org/gcc-4.8/changes.html
[4] http://compiler-rt.llvm.org
[5] https://www.spec.org/cpu2000