IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 17, Issue 3, Ver. 1 (May – Jun. 2015), PP 82-88
www.iosrjournals.org

A novel algorithm to protect and manage memory locations

Buthainah F. AL-Dulaimi, Sawsan H. Jaddoa
(College of Education for Women/University of Baghdad, Iraq)

Abstract: *Most of security vulnerabilities continue to be caused by memory errors, and long-running programs that interact with untrusted components. While comprehensive solutions have been developed to handle memory errors, these solutions suffer from one or more of the following problems: high overheads, incompatibility, and changes to the memory model. Address space randomization is a technique that avoids these drawbacks, but do not offer a level of protection. To overcome these limitations, we develop a new approach in this paper that supports comprehensive randomization, whereby the absolute locations of all (code and data) objects, as well as their relative distances are randomized. In particular, we have successfully deployed precise method in the implementation of a language run-time system. Our approach is implemented as a fully automatic source-to-source transformation, the address-space randomizations take place at load-time or runtime, so the same copy of the binaries can be distributed to everyone - this ensures compatibility with today's software distribution model.*

Keywords: *Garbage collection, memory errors, memory protection, memory reallocation, transformation, program locations.*

I. Introduction

Intuitively, a memory error occurs in programs when the object accessed via a pointer expression is different from the one intended by the programmer. The intended object is called the referent of the pointer. Memory errors can be classified into spatial and temporal errors:

a- A spatial error occurs when dereferencing a pointer that is outside the bounds of its referent. It may be caused as a result of:

- Dereferencing non-pointer data, e.g., a pointer may be (incorrectly) assigned from an integer, and dereferenced subsequently. The same integer value, when interpreted as a pointer, will reference different variables (or code) for each execution of the program.
- Dereferencing uninitialized pointers. This case differs from the first case only when a memory object is reallocated.
- Valid pointers used with invalid pointer arithmetic. The most common form of memory access error, namely, out-of-bounds array access, falls in this category.

b- **A** temporal error occurs when dereferencing a pointer whose referent no longer exists, i.e., it has been freed previously. If the invalid access goes to an object in free memory, then it causes no errors. But if the memory has been reallocated, then temporal errors allow the contents of the reallocated object to be corrupted using the invalid pointer. Note that the results of such errors become predictable only when the purpose of reuse of the memory location is predictable. It may appear that temporal errors, and errors involving uninitialized pointers, are an unlikely target for attackers. In general, it may be hard to exploit such errors if they involve heap objects, as heap allocations tend to be somewhat unpredictable even in the absence of any randomizations. However, stack allocations are highly predictable, so these errors can be exploited in attacks involving stack-allocated variables. [1]

Unfortunately, existing techniques to exploit memory errors are associated with high overheads and nontrivial programming effort is often required so that they can work with these tools. Precompiled libraries can pose additional compatibility problems. The existing approaches are concerned with preventing all invalid memory accesses; we present an approach with a more limited goal: it only seeks to ensure that the results of any invalid access are unpredictable and this goal can be achieved with a much lower runtime overhead. Our approach is based on the address of the location of code and data objects that are resident in memory. The approach is using randomization techniques and these techniques can provide protection against most known types of memory error exploits, they are vulnerable to several classes of attacks including relative-address attacks, information leakage attacks, and attacks on randomization. The approach developed in this paper is aimed at protecting against all memory error exploits, whether they are known or unknown. Our approach makes the memory locations of program objects (including code as well as data objects) unpredictable. This is achieved by randomizing the absolute locations of all objects, as well as the relative distance between any two objects. Our implementation uses a source-to-source transformation. Instead, the transformation produces a self-randomizing program: a program that randomizes itself each time it is run, or continuously during runtime. This means that the use of our approach doesn't, in any way, change the software distribution model that is prevalent today.

II. Static Data Transformations

Static Data Transformations

One possible approach to randomize the location of static data is to recompile the data into position-independent code (PIC). A drawback of this approach is that it does not protect against relative address attacks, e.g., an attack that overflows past the end of a buffer to corrupt security-critical data that is close to the buffer. Moreover, an approach that relies only on changes to the base address is very vulnerable to information leakage attacks, where an attacker may mount a successful attack just by knowing the address of any static variable, or the base address of the static area. At the beginning of program execution, control is transferred to an initialization function introduced into the transformed program. This function first allocates a new region of memory to store the original static variables. This memory is allocated dynamically so that its base address can be chosen randomly. Note that bounds-checking errors dominate among memory errors. Such errors occur either due to the use of an array subscript that is outside its bounds, or more generally, due to incorrect pointer arithmetic. For this reason, our transformation separates buffer-type variables, which can be sources of bounds-checking errors, from other types of variables. Buffer-type variables include all arrays and structures containing arrays. In addition, they include any variable whose address is taken, since it may be used in pointer arithmetic, which can in turn lead to out-of-bounds access. All buffer-type variables are allocated separately from other variables. Inaccessible memory pages (neither readable nor writable) are introduced before and after the memory region containing buffer variables, so that any buffer overflows from these variables cannot corrupt non-buffer variables.

Code Transformations

As with static data, one way to randomize code location is to generate PIC code, and map this at a randomly chosen location at runtime. Specifically, our randomization technique works at the granularity of functions. To achieve this, a function pointer is associated with each function. It is initialized with the value of function. These functions are reordered in a random manner, using a procedure similar to that used for randomizing the order of static variables. Random gaps and inaccessible pages are inserted periodically during this process in order to introduce further uncertainty in code locations, and to provide additional protection. The transformation ensures that these gaps do not increase the overall space usage for the executable by more than a specified parameter. After relocating functions, the initializations of variables are changed so as to reflect the new location of each function. As a final step, the function pointer table needs to be write-protected. [2]

III. Manage Memory Location

Automatic memory management simplifies most implementation tasks, and among programming languages currently in widespread use, most automates memory management through garbage collection (GC). A popular approach for adding GC is to use conservative GC, assumes that any word in a register, on the stack, in a static variable, or in a reachable allocated object is a potential pointer that counts toward the further reach ability of objects. A conservative GC is typically implemented as a library, which makes using it relatively easy. Conservative GC can trigger unbounded memory use due to linked lists that manage threads and continuations. The key constraint a C program must obey to GC is that the static types in the program source must match the program’s run-time behavior, at least to the extent that the types distinguish pointers from non-pointers. The implementation of precise GC, meanwhile, is responsible for handling union types and the fact that the static type of a C variable or expression often provides an incomplete description of its run-time shape. Thus, the concerns for precise GC in the source program can be divided into two parts: those that relate to separating live pointers from other values, and those that relate to connecting allocation with the shape of the allocated data. [3]

For GC to work, all live objects must be reachable via pointer chains starting from roots, which include static and local variables with pointer types. For precise GC, the values of variables and expressions typed as non-pointers must never be used as pointers to GC-allocated objects, because objects are not considered to be reachable through such references. Memory can be allocated but the allocated memory is treated by the GC as an array of non-pointers. Object-referencing invariants must hold whenever a collection is possible, but some code (such as a library function) might safely break the temporarily between collections. The GC may also allow a pointer-typed location to refer to an address that is not a GC-allocated object. The transformation to support precise GC must detect allocations and replace them with GC allocations, in the process associating tags (for mark and repair functions) with the allocated data. [4, 5, 6]

IV. Garbage Collection

In computer science, garbage collection (GC) is a form of automatic memory management. The garbage collector, or just collector, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program. Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of approaches, including other techniques such as stack allocation and region inference. Like other memory management techniques, garbage collection may take a significant proportion of total processing time in a program and can thus have significant influence on performance. The basic principles of garbage collection are:

- Find data objects in a program that cannot be accessed in the future.
- Reclaim the resources used by those objects.

Many programming languages require garbage collection as part of the language specification (for Java, C#) these are said to be garbage collected languages. Other languages were designed for use with manual memory management, but have garbage collected implementations available (for C, C++). While integrating garbage collection into the language's compiler and runtime system enables a much wider choice of methods GC systems exist. The garbage collector will almost always be closely integrated with the memory allocator. Garbage collection frees the programmer from manually dealing with memory deallocation. Some of the bugs addressed by garbage collection can have security implications. As a result, certain categories of bugs are eliminated or substantially reduced:

- Dangling pointer bugs, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is dereferenced. By then the memory may have been reassigned to another use, with unpredictable results.
- Double free bugs, which occur when the program tries to free a region of memory that has already been freed, and perhaps already been allocated again.
- Certain kinds of memory leaks, in which a program fails to free memory occupied by objects that have become unreachable, which can lead to memory exhaustion.
- Efficient implementations of persistent data structures

Typically, garbage collection has certain disadvantages:

- Garbage collection consumes computing resources in deciding which memory to free, even though the programmer may have already known this information. The penalty for the convenience of not annotating object lifetime manually in the source code is overhead, which can lead to decreased or uneven performance.
- The moment when the garbage is actually collected can be unpredictable, resulting in stalls scattered throughout a session. Unpredictable stalls can be unacceptable in real-time environments, in transaction processing, or in interactive programs. Incremental, concurrent, and real-time garbage collectors address these problems, with varying trade-offs.

Tracing garbage collection

The overall strategy consists of determining which objects should be garbage collected by tracing which objects are reachable by a chain of references from certain root objects, and considering the rest as garbage and collecting them.

Reference counting

It is a form of garbage collection whereby each object has a count of the number of references to it. Garbage is identified by having a reference count of zero. An object's reference count is incremented when a reference to it is created and decremented when a reference is destroyed. The object's memory is reclaimed when the count reaches zero. As with manual memory management, and unlike tracing garbage collection, reference counting guarantees that objects are destroyed as soon as their last reference is destroyed, and usually only accesses memory which is either in CPU caches, in objects to be freed, or directly pointed by those, and thus tends to not have significant negative side effects on CPU cache and virtual memory operation. [7, 8]

V. The Proposed Transformation

The proposed transformation algorithm and the implementation

The proposed transformation algorithm is outlined in the following steps, and its implementation is illustrated afterward in Fig. 1 below.

Step 1: Prompt the user to enter the number of random memory locations to generate.

```
cout<<"How many random memory location to generate:";
```

cin>>howMany;
Step 2: create a dynamic array to store the random numbers using malloc.
intptr =(int *) malloc(howMany * sizeof(int));
step 3: In this example we have tested to be sure malloc succeeded. If not, we are outof memory
if (intptr == NULL){
cout<<"memory allocation error exiting\n";
exit(1);}

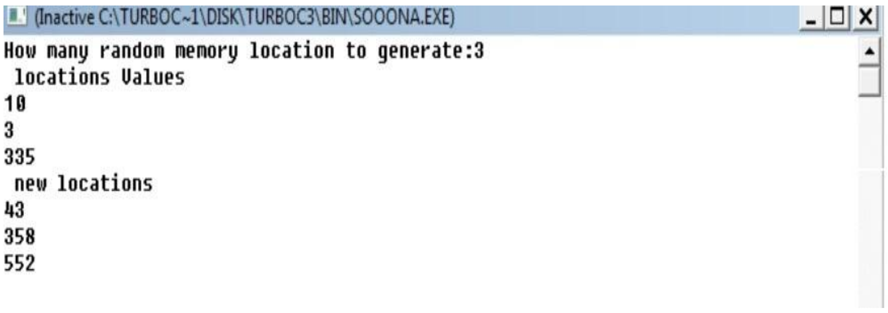
Step 4: generate the random locations and print them
for (inti=0;i<howMany;i++){
intptr[i]=random(1000);}
cout<<" locations Values\n";
PrintArray(intptr,howMany);

Step 5: double the size of the array usingrealloc
intptr = (int *)realloc(intptr, 2*howMany);

Step 6: generate new array add to random numbers and print them
for (i=0; i<howMany; i++){
intptr[i]=intptr[i]+random(1000);}

```
        cout<<" new locations\n";
        PrintArray(intptr,howMany);
```

Step 7: free the memory location we used
free(intptr);



```
(Inactive C:\TURBOC~1\DISK\TURBOC3\BIN\SOOONA.EXE)
How many random memory location to generate:3
 locations Values
10
3
335
 new locations
43
358
552
```

Fig. 1. The implementation of the transformation

Our approach provides the following benefits:

- Ease of use. Our approach is implemented as an automatic, source-to-source transformation, and is fully compatible with legacy C code. It can interoperate with preexisting (untransformed) libraries.
- Comprehensive randomization. At runtime, the absolute as well as relative distances between all memory-resident objects are randomized.
- Portability across multiple platforms. The vast majority of our randomizations are OS and architecture independent. This factor eases portability of our approach to different platforms.
- Low runtime overhead. Our approach produces low overheads.
- Ease of deployment. Our approach can be applied to individual applications without requiring changes to the OS kernel, system libraries or the software distribution models.

VI. The Proposed Garbage Collection

The proposed strategy

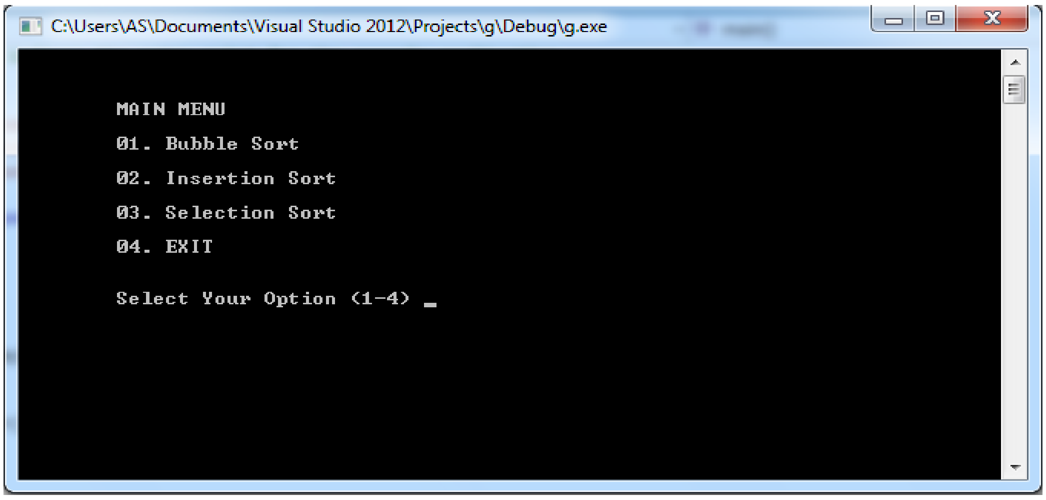
- The program is consists of three classes the first one is “oper” stands for “operation” which is the main class and the second is “RC” stands for “reference counter” which plays the role of deleting the object, the last class is “SP” stands for “smart pointer” which represent the functions of the smart pointer in our example.
- Class “oper” consists of an array that will store the elements for three functions each one represent a specific type of sorting algorithms (bubble, insertion, and selection). Also “display” function, the default constructors and destructor.

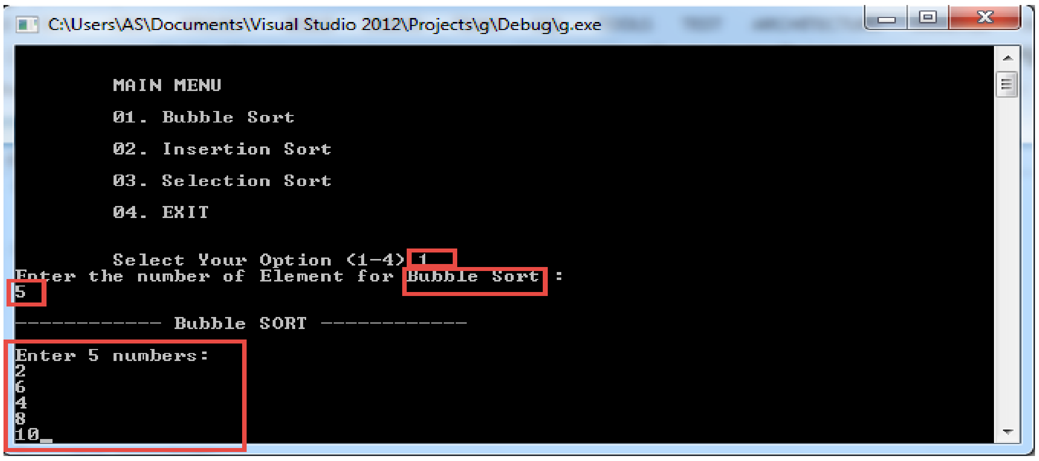
- Class “RC” consists of one variable which is “count” it is to count the number of pointers that refer to a specific location, also two functions “AddRef()” increments the count value, and “Release” decrements the count value.
- The last class is “SP” the genuine of the program it connect the parts of the program to implement the desired idea; its consists of the pointer “pData” which refers to any type of data because its created as template and that’s what the “SP” class accepts as an input – any type of data – and “reference” pointer which refers to an object from the class “RC”.

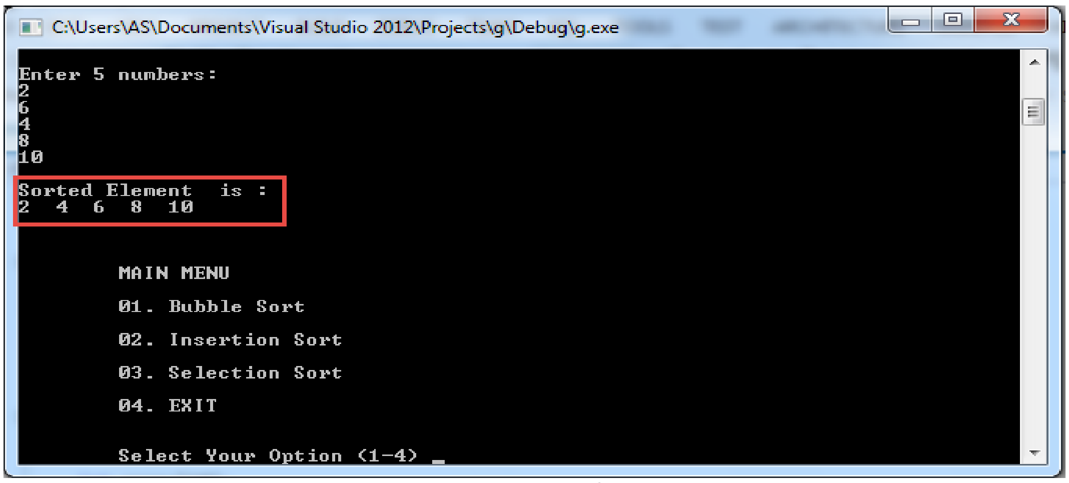
The constructors of the class which calls the function “AddRef()” from the class “RC” so that the “count” variable increments every time a smart pointer is created, weather the destructor deletes the smart pointer when the value of “count” reaches “0” given from the function “Release”. Then an overloaded functions defined that gives the class the abilities to return the location that the smart pointer refers to “operator*”, return the value of the content of the smart pointer “operator->”, and the assignment operator “=” which rather than giving the value of the location that the smart pointer refers to for more than one object, it calls the function “AddRef()” so the value of “count” increases.

The Implementation

Fig. 2 is the “main()” function, in this function a smart pointer created with an operator object , by running the program four options is there for the user. The first three of them represent the three types of sorting functions and the last one is exit which deletes the pointer and ends the program, choosing one sorting function leads us to write the numbers to be sorted as in Fig. 3. And then “display” function displays the numbers with the correct arraignment as shown in Fig. 4. And so on the program takes the values from the user and uses one of the sorting functions depending on the choice of the user as in Fig. 5 and demonstrating results in Fig. 6. Until the user decide to end the process the objects delete their selves and the smart pointer also do that and the program ends as in Fig. 7.

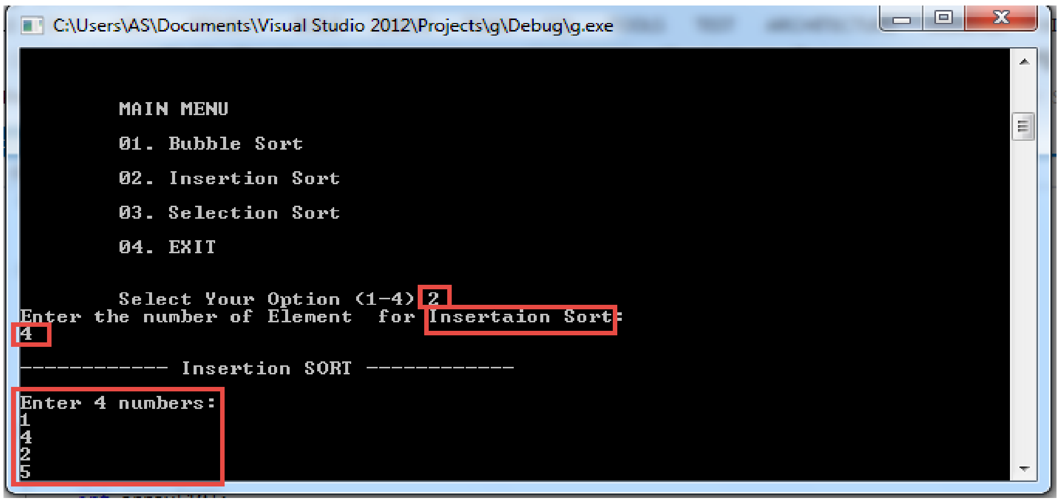

```
C:\Users\AS\Documents\Visual Studio 2012\Projects\g\Debug\g.exe

        MAIN MENU
        01. Bubble Sort
        02. Insertion Sort
        03. Selection Sort
        04. EXIT

        Select Your Option (1-4) _
```
Fig. 2. The main menu


```
C:\Users\AS\Documents\Visual Studio 2012\Projects\g\Debug\g.exe

        MAIN MENU
        01. Bubble Sort
        02. Insertion Sort
        03. Selection Sort
        04. EXIT

        Select Your Option (1-4) 1
Enter the number of Element for Bubble Sort :
5

------------ Bubble SORT ------------

Enter 5 numbers:
2
6
4
8
10_
```
Fig. 3. the bubble sort



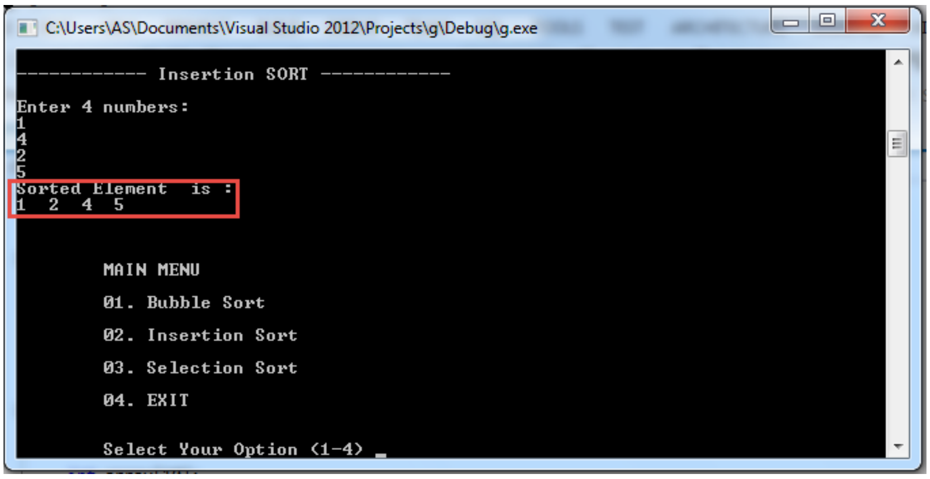
```
C:\Users\AS\Documents\Visual Studio 2012\Projects\g\Debug\g.exe

Enter 5 numbers:
2
6
4
8
10
Sorted Element  is :
2   4   6   8   10


        MAIN MENU

        01. Bubble Sort

        02. Insertion Sort

        03. Selection Sort

        04. EXIT


        Select Your Option (1-4) _
```

Fig. 4. The result of sorting



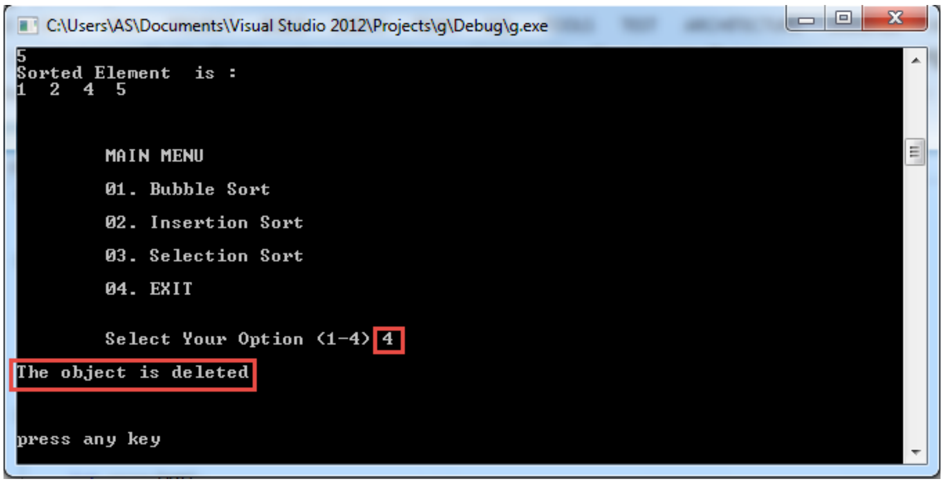
```
C:\Users\AS\Documents\Visual Studio 2012\Projects\g\Debug\g.exe


        MAIN MENU

        01. Bubble Sort

        02. Insertion Sort

        03. Selection Sort

        04. EXIT


        Select Your Option (1-4) 2
Enter the number of Element  for Insertaion Sort:
4

------------ Insertion SORT ------------

Enter 4 numbers:
1
4
2
5
```

Fig. 5. Using sorting choice



```
C:\Users\AS\Documents\Visual Studio 2012\Projects\g\Debug\g.exe
------------ Insertion SORT ------------

Enter 4 numbers:
1
4
2
5
Sorted Element  is :
1   2   4   5


        MAIN MENU

        01. Bubble Sort

        02. Insertion Sort

        03. Selection Sort

        04. EXIT


        Select Your Option (1-4) _
```

Fig. 6. The sorting result



```
C:\Users\AS\Documents\Visual Studio 2012\Projects\g\Debug\g.exe
5
Sorted Element  is :
1  2  4  5


        MAIN MENU

        01. Bubble Sort

        02. Insertion Sort

        03. Selection Sort

        04. EXIT

        Select Your Option (1-4) 4
The object is deleted


press any key
```

Fig. 7. Stopping the process

VII. Conclusions

Our approach is based on permuting the order of static variables at the beginning of program execution. Address space randomization is a technique which provides broad protection from memory error exploits in C and C++ programs. However, previous implementations of ASR have provided a relatively coarse granularity of randomization, with many program objects sharing the same address mapping, so that the relative distance between any two objects is likely to be the same in both the original and randomized program. This leaves the randomized program vulnerable to guessing, partial pointer overwrites and information leakage attacks, as well as attacks that modify security-critical data without corrupting any pointers. Our approach is implemented using a source-to-source transformation that produces a self-randomizing program, which randomizes its memory layout at load-time and runtime. This randomization makes it very difficult for memory error exploits to succeed. Automatic memory management offers many benefit for C code, and conservative GC and reference counting are fine choices for many programs. Those memory-management techniques are a poor match, however, for long-running programs that have complex reference patterns and that host extensions or untrusted code. GC needs five times the memory to compensate for this overhead and to perform as fast as explicit memory management.

References

[1]. S. Bhatkar, D. C. Du Varney, and R. Sekar, Address obfuscation: An efficient approach to combat a broad range of memory error exploits, In USENIX Security Symposium, Washington, DC, August 2003.

[2]. S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer, CIL: Intermediate language and tools for C program analysis and transformation, In Conference on Compiler Construction, 2002.

[3]. J. Baker, Antonio Cunei, FilipPizlo, and Jan Vitek, Accurate garbage collection in uncooperative environments with lazy pointer stacks, In International Conference on Compiler Construction, 2007.

[4]. M. Hirzel, A. Diwan, and J. Henkel, On the usefulness of type and liveness accuracy for garbage collection and leak detection. ACM Transaction Program. Language System, 24(6), 2002, 593–624.

[5]. S. M. Pike, B. W. Weide, and J. E. Hollingsworth, Checkmate: cornering C++ dynamic memory errors with checked pointers. SIGCSE Technical Symposium on Computer Science Education, 2000, 352–356.

[6]. A. W. Magpie, Precise Garbage Collection for C, PhD thesis, University of Utah, June 2006.

[7]. J. Richard, H. Antony, M. Eliot, The Garbage Collection Handbook: The Art of Automatic Memory Management, CRC Applied Algorithms and Data Structures Series. Chapman and Hall/CRC, 19 Aug. 2011, ISBN 1-4200-8279-5.

[8]. W. Fu and Carl Hauser, A Real-Time Garbage Collection Framework for Embedded Systems, ACM SCOPES '05, 2005, Portal.acm.org. Retrieved 9 July 2010.