Size-aware Buffers for preventing Buffer Overflow (SABO)

Yugarsi Ghosh, Sailik Sengupta

• The problem

The language C provides memory allocation and deletion functions that allow manipulation of memory at runtime. The size of this allocated memory is not available at the point of access. Thus, one cannot verify if the memory being accessed is within the boundaries of its initial definition. This has major security implications. As a domino effect, functions that write to buffers (allocated memory via the aforementioned functions) cause memory corruption via buffer overflows. This project seeks a way to prevent this in already developed code.

• The approach

Dynamically linking functions with LD_PRELOAD provides a strategy to _overload_ memory allocation functions in C. Using this approach, whenever memory manipulation occurs, we get a chance to control the code flow. At this point, we store the pointer to the memory and the size it was allocated. This allows us to check (at a later point in time) if access is made to a memory location outside what was allocated and prevent it.

The following functions have been overridden with the help of LD_PRELOAD trick. The following lines will describe the various components of the secure library and how various vulnerable functions has been modified to ensure security without compromising the desired functionality.

a. The global allocator (ga) table –

It contains number of entries of a data structure “sabi” which holds

i) The start address of the chunk of memory allocated be malloc.

ii) The size of the memory chunk allocated by malloc.

b. Malloc – This function has been modified to update the global allocator table,

With the start address and size of the chunk of allocated memory every time it is called from any function in a C program.

c. strncpy – This function takes three arguments the source pointer, destination pointer and the destination pointer size. This function is unsafe when a non null terminated string is copied to the destination. We ensure that the last byte is a null terminated '\0'.

d. strcpy – This function takes two arguments, the source pointer and destination pointer. We get the value of maximum size allocated to destination pointer "n" from the ga table and call the strncpy function with parameter size = n.

e. gets – This function has one argument which is a source pointer which stores characters from the standard input. We get the value of maximum size allocated to the source pointer "n" from the ga table. We restrict the number characters of user input to n-1 and terminate it with a '\0'.

f. fgets - This function takes three arguments the source pointer, the destination pointer size, and the file pointer. This function is unsafe when a non null terminated string is copied to the destination. We ensure that the last byte is null terminated.

g. strcat - This function takes two arguments, the destination pointer and the source pointer. We get the value of maximum size allocated to the destination pointer "n" from the ga table. We restrict the size of destination to n-1 and we ensure that the last byte is null.

h. free - This function has been modified to remove the entry from the global allocator table before the allocated chunk of memory is freed from the system.

i. realloc – This function has been modified to call our secure "free" function and then our modified malloc function to reallocate a chunk of memory.

- **Usage**

To benefit quick and easy integration of our library, we have a simple 2 step process.

- Generate a file for dynamic linking.

```
gcc −Wall −shared −fPIC –o libsabo.so libsabo.c −ldl
```

- Load our library when executing your program via LD_PRELOAD

```
LD_PRELOAD=/libsabo.so ./a.out
```

- To further facilitate this process we have a shell code 'runner.sh' that does the above mentioned steps. An example of using this code follows:

```
. / runner.sh −f overflow_prevention/test3.c −a ‘‘` python −c ‘‘print ‘a’*499 +
‘b’*18 + ‘ ’ + ‘37 aa ’ + ‘755’ ”””
```

The code can be found on github (made public after submission deadline).

- **Limitations**:

a. Memory accessed via the '[]' and '+' operator directly cannot be verified since C does not allow operator overloading (except what is already defined in the language).

b. The global array limits the library's efficiency in terms of memory when used for large code bases.

c. Time taken for searching a value in the global array and removing items (for 'free') is linear.

- **Improvements**:

a. To conclude, we address a few of the limitations mentioned and suggest a few enhancements that can be made.

b. Use of hash map for the global array allows constant time look up and deletion.

c. The mechanism keeps track of all pointers and their allocated sizes. This data along with usage tracking can help in doing automated garbage collection for C.