Size aware buffer overflow (SABO) prevention library (CSE 545)

Sailik Sengupta, Yugarsi Ghosh

1 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.

2 The approach

Dynamically linking functions with LD_PRELOAD provides a strategy to overload memory allocation functions in C. Whenever memory allocation occurs (via 'malloc' and 'realloc'), the call is routed via the newly created library. This library contains a global array of structures that enables us to store a pair of values– 'starting address' and the 'size' it was allocated.

Using the global array, we solve the problem of buffer overflow in 'strcpy', 'strncpy', 'strcat', 'sprintf', 'gets', and 'fgets'. All of these functions write to a memory (char *) that was allocated using the memory allocation functions in C. We write wrappers over these calls. The backbone of these wrappers is based on the following algorithm.

```
/*
** @Input
**      'p' starting address of writable memory sent from calling
**       code
**
** @Output:
**     -1 if 'p' is not present in the global array
**      size of writable memory starting at p
*/
size_t isWritable( void *p ) {
    int i;
    for( i=0; i<count; i++) {
        if ( ga[i].ptr <= p && p <= ga[i].ptr + ga[i].size ) {
            return (ga[i].ptr + ga[i].size) - p;
        }
    }
    return -1;
}
```

The wrappers obtain the amount of writable memory starting at the address mentioned using the above method (if any). With this information, they safely write the amount of data starting at ‘p’ that will not cause any buffer overflow. Lastly they ensure that all these strings are null-terminated (since these are wrappers to string functions).

The ‘free’ function removes the entry of the pointer mentioned from the global array and ‘free’s the memory. This ensures that an attacker cannot use an old entry in the global array to overwrite memory.

3 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=<PATH_TO_GENERATED_LIBRARY>/libsabo.so
  ./a.out <program_args>
  ```

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).

4 Limitations

We highlight a list of limitations here:

- Memory accessed via the ‘[]’ and ‘+’ operator directly cannot verified since C does not allow operator overloading (except what is already defined in the language).
- The global array limits the library’s efficiency in terms of memory when used for large code bases.
- Time taken for searching a value in the global array and removing items (for ‘free’) is linear.

5 Improvements

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

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