CSE-545 Project Report

Shashank Singh

April 2016

1 Buffer Overflow Problems

Let's say we have something similar to below piece of code in some un-noticed secton of a large codebase:

```
#include <string.h>
#include <stdio.h>
int main(int argc, char ** argv){
    char *name = malloc(256);
    strcpy(name, argv[1]);
    printf("Hello %s\n", name);
    return 0;
}
```

It is easy to note that since strcpy does not check for size, it will continue to copy contents until it sees a '\0' to memory pointed by 'name'. Malicous (or even simply erronous or larger than predicted) contents in argv[1] can cause Buffer Overflows that could lead to segmentation faults, breach of data integrity or even malicious code execution. The ideal way to prevent such overflows would be to use the strncpy function instead of strcpy where we can sprecify the maximum number of characters to copy:

```
    strncpy(name, argv[1], 256);
```

However, complete re-factoring of code might take considerable time and efforts. Is there an easier way to convert all these risky overflow calls like strcpy, strcat and gets to safer counterparts like strncpy, strncat and fgets without significant changes in source code? Particularly, can LD_PRELOAD feature in Linux systems be used for linux binaries (ELF)? Is it reliable (and hopefully fast) enough? This project is an attempt to deal with these questions.

2 Approach

The main idea is to create a wrapper library with custom implementations of overflow functions that should internally try to call safer functions if possible. There are two types of buffers : stack and heap.

```
// assigned memory in the stack - address offset stored at compile time
char sbuff[256];
strcpy(sbuff, argv[1]);
// assigned memory in heap. address allocated runtime via malloc/calloc
char * hbuff;
hbuff = malloc(256);
strcpy(hbuff, argv[2]);
```

When the above code compiles, in its respective function, the compiler makes room for 256 bytes for sbuff and 1 word(pointer) for hbuff (based on 32/64 bit architecture) on the stack. It then has a call for malloc which finds 256 bytes of memory in the heap and returns its address which is then stored in hbuff. By default, **malloc** will be dynamically resolved by the linker (unless specified otherwise). This is where our wrapper library comes into the picture with **LD_PRELOAD**.

```
LD_PRELOAD=<libraries to load> <binary>
```

LD_PRELOAD is a special environment variable tells the linker that we want to load these binaries first before others (like the standard libc libraries). If we can make our own malloc/calloc/realloc and free wrappers that store the heap memory allocations in a data structure and also store the size of memory allocated for those pointers, we could also write our own implementations of strcpy, strcat and fgets with the simple algorithm as below:

```
function <overflow_func>(char * dest, char * src)
    Check if we know how much size is allocated to 'dest'
    If we know
        <libc_size_safe_function>(dest, src, max_size)
    else
        <libc_overflow_func>(dest, src)
```

This is really a simple idea for dealing with heap overflows. For dealing with stack based overflows, the idea that I had initially was that since we would be dealing with arrays in stack buffers, and we could get the size of an array by:

```
size_t max_size = sizeoff(arr)/sizeof(arr[0]);
```

It turned out to be pre-mature because sizeof() operator is resolved at compile time and can be resolved only within the scope of the function it is declared. Arrays are passed to other functions by implicit conversion to pointers and the max_size would not give what is expected then. I couldn't find another reasonable approach to that; all that the wrapper can get is an address to the stack with no known size, addresses which are decided on compile time. This is currently for future work. All my current efforts were directed towards making a reasonably robust heap protector first that could work with almost all programs. Unfortunately this isn't that reliable for big programs, but it can be used for logging, learning and protecting smaller programs.

3 Implementation

3.1 Data Structure

A Linked list [O(n)] was used to store addresses of memory allocated along with their respective sizes. Code can be found in file **linkedlist.c**. This was the most crucial component and attempt was to keep it simple yet robust. Test cases for the data structure are present in file test_datastructure.c.
An ideal datastructure for this project would have been a Hashtable implementation [O(1)] in C. However, it is a point for future work as it was not feasible to use any library out of the box without tweaking because of the below use-case:

If we know that address 200 was allocated with 500 bytes of memory, and if we are to ask the datastructure if it knows the size of memory allocated at address 300, it should respond with a size of 400 (500 - (300 - 200)). This corresponds to test case in file test3.c.

3.2 Wrappers

Wrapper functions conforming to the specs **here** are in **wrapper.c** file:

```
// Calls libc-malloc internally using dlsym.
// Stores (address , size) tuple in datastructure
void * malloc(size_t );

// Calls libc-free internally using dlsym.
// Removes (address , size) tuple from datastructure
void free(void *);
```

```
// Calls libc-calloc internally using dlsym.
// Stores (address - size) tuple in datastructure
void * calloc(size_t, size_t);

// Calls libc-realloc internally using dlsym.
// Updates the datastructure accordingly:
//   If new address returned, remove prev and add new
//   If same resized, update size
//   If new, save
//   If freed, remove
void * realloc(size_t, size_t);

// Checks if it knows the size of dest
// If yes, calls safer strncpy internally and ensures null termination.
// else, calls libc-strcpy internally
char * strcpy(char * , char * );

// Checks if it knows the size of dest
// If yes, calls safer strncat internally and ensures null termination.
// else, calls libc-strcat internally
char * strcat(char * , char *);

// Checks if it knows the size of dest
// If yes, calls safer fgets internally (with stdin as stream).
// else, calls libc-gets internally
char * gets(char *);
```

3.3 Testing

Testing done with test cases (attached with the report in test_cases folder) on a Ubuntu 14.04 64 bit machine with gcc 4.8.4. Basic, written test cases passed and basic programs like ls, cp, rm etc work with the library. Programs dealing with Multi-Threads might NOT pass test as they are significantly larger programs with which this library has issues still to be investigated.

4 Usage

4.1 Compilation

To compile the library, you will need **gcc**. Execute the following command in terminal:

```
gcc -shared -fPIC -fno-builtin -o ⟨libsafer.so⟩ wrapper.c linkedlist.c -ldl
```

This will produce libsafer.so, a library compiled for your platform by the above command. You can use the -g flag with gcc if you want to keep debug information (recommended).

4.2 Execution

To make a pre-compiled binary to execute calling this wrapper, execute your binary in the following manner:

```
LD_PRELOAD=⟨./libsafer.so⟩  ⟨./binary⟩
```

Although it is not reliable enough yet (and thus not recommended), you can export LD_PRELOAD so that all subsequent binaries that are dynamically linked will use the wrapper.

```
export LD_PRELOAD=⟨./libsafer.so⟩
```

5 Problems and Limitations

5.1 dlsym recursion problem

Observation is that often in larger applications(possibly because of multi-threading), dlsym internally calls calloc. But our wrapper calloc internally uses dlsym to get a handle to libc-calloc implementation. This leads to an infinite loop and segmentation faults. There are a couple of solutions mentioned online (see **Resources** section) but they did not work for me and are scope for future work. The main problem being - how can we know for sure that while wrapper is using dlsym, dlsym should not be using the wrapper. Also, if we do not want to use dlsym, we might have to write a custom malloc robust enough for all purposes. That would be significant work all over again, and would it perform well enough for practical use? Could we use malloc hooks here? Points for future work.

5.2 Platform Limitation

Currently tested only on Linux systems with glibc installed. LD_PRELOAD works only when you are using the **ld** linker or when an ELF file calls ld at runtime. It might not work with binaries compiled and linked with clang et all/other linkers. Currently it is architecture independent but platform dependent. There could be other options in other environments to do something similar but that is for future work.

5.3 Language Limitation

Tested with C and C++. Binaries compiled with g++ can execute this library the same way as those with gcc. Compiling this library with g++ would require some syntax changes and no tangible benefits yet (possible for using STL data-structures?). Could work with other languages that might be using C library calls internally but not tested.

5.4 Compiler Limitations

Often, gcc / g++ can optimize code and thus it might not give you the expected behaviour. Example below:

```
strcat(ptr, argv[1]);
// compiler might optimize and not call strcat for below
strcat(ptr, "\n");
```

This might result in the wrapper strcat not being called the second time and lead to an overflow. gcc with **-fno-builtin** option helps specify not to make such optimizations.

5.5 set-uid set-guid Limitations

For security reasons; for set-uid type ELF files, LD_PRELOAD does NOT have any effect unless the library has the same set-uid bits as that of binary and it should be in the default search space of ld (read man ld). If this was not the case, anyone could write a malicious wrapper and run a set-uid file.

6 Future Work

This wrapper is still unreliable to use with any decent sized program that uses multi-threading (tested on vim, gdb, python and they give seg-faults). Some problems are identifiable as recursive loops when dlsym calls malloc/calloc. Others require more investigation. But it works for most of the simpler commands like ls, cp, rm, pwd, etc and the limited test scripts attached.

- Provision for reliable multi-thread use, which should also prevent recursive loops (dlsym calls calloc, calloc calls dlsym....)
- Check the pros/cons of **malloc_hooks** approach and if it makes more sense to take that approach
- If there are still cases of recursive loops via dlsym, try to include a fairly reliable and efficient implementation of malloc and free in wrapper itself and remove dependency on dlsym.
- Test for usage with significantly large and used programs like vim, gdb, python interpreter, etc.
- Improve datastructure to a faster hashtable.
- Scope for Windows/MacOS
- Can this be extended to detect stack memory overflows?

7 Relavant Resources

1. **http://elinux.org/images/b/b5/Elc2013_Kobayashi.pdf**
2. **http://blog.bigpixel.ro/2010/09/interposing-calloc-on-linux/**
3. **Heap consistency Checking**