# CSE 545 Project: Size Aware Overflow Functions

Chaitanya Palaka 1209261868

**The problem:**

A major security issue in the C/C++ languages are buffer overflows. These occur when a program, while writing data to a buffer, overruns the buffer boundary and overwrites adjacent memory locations. This is a problem especially in languages like C and C++ because they allow the programmer to directly control memory. Two common types of overflow exploits are *stack overflows* and *heap overflows* where the overflowed buffer resides on the stack and heap respectively. Stack overflow exploits usually deal with overwriting the saved EIP value on the stack and overwriting it with a specific memory address which points to some malicious code, typically shellcode. Heap overflows work similarly, but instead of overwriting saved EIP values which directly control program flow, they work by overwriting metadata related to storing and linking free and used 'chunks' of dynamically allocated data and ultimately overwrite a program function pointer to influence program execution. Dynamic memory allocation in C is managed by the malloc family of functions (malloc, free, realloc, etc).

A key insight into buffer overflow vulnerabilities is that they are fundamentally caused by functions used to copy bytes from one location to another, which DONT keep track of the destination and source buffer sizes. Common dangerous functions are : strcpy, strcat and gets. Therefore, if we modify these dangerous functions to keep track of the size of the buffers they are manipulating, buffer overrun problems can be prevented entirely.

**Implementation:**

The method I have used is to write a small library with alternative definitions to these dangerous functions (strcpy, strcat, gets) which will override the C standard library's function definitions' for the same by linking my library (ovfix.so) to the environment variable LD_PRELOAD. This forces my library to be loaded first for a program, rather than libc itself.

Firstly, we should note that the three places which buffers may be allocated are : the stack, the heap, and the data segment as global variables. My library fixes overrun problems in the first two, but global variables are considered as well.

(i) Heap: My first idea was to redefine malloc so that every time malloc is called by the program, I will store chunk sizes and locations in a separate structure and use this structure to assert that the buffers on the heap are not overflowed when copying data. This is a decent idea, but I realized I could be more space-efficient (i.e not using a potentially huge internal structure) by using knowledge of malloc metadata as well as a helper function defined in malloc.h called *malloc_usable_size*. Malloc is implemented to store 'boundary tags' between the used chunks of memory (unused memory chunks are stored slightly differently, but we dont need to consider that for our problem). This metadata contains the size of the allocated chunk before and after the chunk, as well as a bit specifying whether the chunk is free or in use. This is the info malloc_usable_size uses to calculate the available size of any used chunk. The only issue here was, if a buffer exists in the middle of an allocated chunk (defined in the middle of a struct for example), malloc_usable_size does not return the size of the chunk, but a value <= 0. A known fact is that all of the instances of heap allocated memory are on even byte boundaries. So, if a buffer exists in the middle of a chunk, I call malloc_usable_size repeatedly, decrementing the input pointer by 4 bytes each time until it outputs a value >0, at which point I can be sure that this is the value of the total size of this chunk. In this way, and with some minor calculations, I can find how much space is available after the pointer to the buffer and before the chunk ends. I compare this available size to the size of the src string (assumed to be a proper c string and null terminated) and if the src string is too big, I stop program execution and print a buffer overrun warning to stdout. If its not too big, I call the original strcpy/strcat function and allow the program to execute normally. A slight tradeoff I have made here is that technically, the buffer IS allowed to overflow, up til the chunk boundary. So, the memory next to the buffer in the chunk might get corrupted, but I prevent any overflows into the malloc metadata, which is the real security issue here.

(ii) Stack: The first step in checking for buffer overflows on the stack, is to know whether the pointer passed to strcpy/strcat resides on the stack in the first place. One way of doing this is to use the gcc built-in function ***__builtin_frame_address*.** This function takes in the 'stack level' and it returns the address of the stacks function frame at that level. For example, level 0 corresponds to the current stack frame. By looping through all the stack levels to the top, we can store where all the function frames are located and subsequently find the address range of the stack. In my library, the static function *isInHeap* has this functionality. It will take a pointer as input and outputs whether it resides on the stack, heap or data segment by checking if the address location of the pointer is in the stack range (stack buffer), and if not, if its between the top of the stack and start of the heap (heap buffer) and if not, it defaults to being a global buffer allocated in the data segment of the process. (On a sidenote, the heap starting location is found by preloading malloc and calling sbrk(0) to find the current break location of the heap. We save the lowest of these, which naturally corresponds to the beginning of the heap). Once a pointer is confirmed to be on the stack, the process of checking the available size of the buffer is similar to what is used on the heap. The function frame pointers returned by *__builtin_frame_address* point to the location on the stack right before the stored EIPs, so by calculating the number of bytes between the buffer location and its function frame address, we can find how many bytes can be written without 'smashing the stack'.

Some assumptions I made, and tradeoffs to this approach are :

-All strings MUST be null terminated. The programs behaviour is undefined for non null terminated strings.

-This library will only work for 32-bit binaries

-It is also dependent on the current version of malloc (ptmalloc2)

-ASLR should be disabled

-gets() is a very dangerous function because the size of the string read from stdin cannot be known at compile time and so, our library disallows its use entirely.

**Usage:**

The library must built as a shared library with gcc with the following options : -m32 -fPIC -Wall -shared -ldl. After this, we can add the shared library (.so file) to the LD_PRELOAD environment variable so that the libraries strcpy/strcat definitions are loaded instead of libc's. This library can be used with any 32-bit binary executable. (Dont forget to unset LD_PRELOAD after using the library!)

**Future Improvements:**

In the future Id like to improve the efficiency of my method. There is an overhead of tracing the call stack every time functions strcpy/strcat are called. I would also like to improve on finding the exact buffer sizes so that even memory next to the buffer cannot be corrupter. Currently, derailing the programs control flow using buffer overflows is not possible, but corrupting nearby memory on the stack is.