Security testing

Nicolò Fornari

February 19, 2016

1 Buffer Overflow

1.1 Stack and Heap

The *stack* is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some book keeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack; freeing a block from the stack is nothing more than adjusting one pointer.

The *heap* is memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time; there are many custom heap allocators available to tune heap performance for different usage patterns.

Each thread gets a stack, while there's typically only one heap for the application (although it isn't uncommon to have multiple heaps for different types of allocation).

The OS allocates the stack for each system-level thread when the thread is created. Typically the OS is called by the language runtime to allocate the heap for the application.

Scope: the stack is attached to a thread, so when the thread exits the stack is reclaimed. The heap is typically allocated at application startup by the runtime, and is reclaimed when the application (technically process) exits.

Size: the size of the stack is set when a thread is created. The size of the heap is set on application startup, but can grow as space is needed (the allocator requests more memory from the operating system).

Speed: the stack is faster because the access pattern makes it trivial to allocate and deallocate memory from it (a pointer/integer is simply incremented or decremented), while the heap has much more complex bookkeeping involved

in an allocation or free. Also, each byte in the stack tends to be reused very frequently which means it tends to be mapped to the processor's cache, making it very fast. Another performance hit for the heap is that the heap, being mostly a global resource, typically has to be multi-threading safe, i.e. each allocation and deallocation needs to be - typically - synchronized with "all" other heap accesses in the program.

1.2 Stack smashing

The canonical method for exploiting a stack based buffer overflow is to overwrite the function return address with a pointer to attacker-controlled data (usually on the stack itself).

1.3 Fixing buffer overflow

- Used counted versions of string functions (less performance but there is a boundary check)
- Use safe string libraries, if available, or C++ strings
  - fgets instead of gets - snprintf instead of printf
- Check loop termination and array boundaries
- Use C++/STL containers instead of C arrays