CS915 Advanced Computer Security
Lab 2: Advanced Concepts

In this session we cover some more advanced concepts from the C programming language including structs, data structures, and additional work on pointers and memory allocation. We also look at some Linux and Assembly Language fundamentals which will be required to complete the coursework.

There are assessed exercises scattered throughout this lab which should be completed individually as submissions will be screened for plagiarism. Each exercise should be saved in its own C file and submitted to Tabula. Please ensure your files compile before submitting, as marks will be deducted if this is not the case. If you are struggling with any exercise please let a tutor know.

1 Basic C Structures

Structures, or *structs*, are a mechanism for defining complex data types in C. They allow a programmer to combine a group of related variables to be placed in a contiguous block of memory. The struct shares a superficial similarity with Java's objects in that they allow a programmer to model a complex structure like a bank account as a single entity. That said, C structs are not objects as they cannot contain functions, have no mechanism for inheritance or polymorphism and do not support encapsulation.

The following code demonstrates the use of a statically allocated struct. The *struct account* type models a simple bank account. Data members are accessed with dot notation: `struct_name.member_name`.

```
#include <stdio.h>

struct account {
  char *account_holder;
  double balance;
  double interest_rate;
};

int main(int argc, char* argv[]) {
  // Structs have a special initialization syntax:
  struct account my_account = {"John Smith", 1000.0, 0.03};
  printf("Account holder: %s\n", my_account.account_holder);
  printf("Account balance: %f\n", my_account.balance);
  // Withdraw some funds:
  my_account.balance = my_account.balance - 35.0;
  printf("New account balance: %f\n", my_account.balance);
}
```

2 Memory Allocation

A compiled program's memory is divided into five segments: text, data, bss, heap and stack. Each segment has a different purpose. The two segments which programmers are most concerned with are the stack and the heap. There are two ways memory can be acquired in C - statically and dynamically - which cause memory to be allocated on the stack and heap respectively.

In the previous lab we saw both static and dynamic memory allocation. When a function is called, code generated automatically by the compiler ensures enough space is reserved on the stack to store all the variables defined within a function. When a function exits, its stack frame ceases to exist and hence all statically allocated memory is automatically freed.

In dynamic memory allocation, memory is allocated in an on-demand fashion with the use of functions like `malloc` which return pointers to the newly allocated memory. Dynamic memory is allocated in the heap. Any memory allocated on the heap will stay around until it is released by calling the free function. It is important that you free any memory you allocate with `malloc()/calloc()/realloc()` otherwise your application will have a 'memory leak'.

```
void foo(double argument) {
  // Static array allocation
  int my_array[10];
  // dynamic allocation
  int *dynamic_array = malloc(10 * sizeof(int));
}

int main(int argc, char *argv) {
  foo(42.0);
  // varibles a_number and my_array have ceased to exist here.
  // The memory pointed to by dynamic_array has not been freed
  // but we have lost our pointer to it. This is a memory leak.
}
```

3 Data Structures

Structs are commonly used to implement data structures like lists and stacks in C. Because data structures such of these are expected to grow and shrink at runtime and persist across function calls they are typically allocated dynamically on the heap.

Exercise 1: The following listing demonstrates a partial linked list implementation. Note that as C lacks objects, the methods to update the list are external and are not part of the element or linked_list structs. Another key point to notice in this listing is the introduction of the crows-foot operator, ->, which is syntactic sugar to access struct members directly through a pointer.

Your task is to complete the implementation below. You should add four methods; *count* to count the number of elements in a list, *prepend_int* to add elements to the start of the list, *remove_head* to remove elements from the start of the list and *remove_tail* to remove elements from the tail of the list. **Note:** Be careful to avoid memory leaks.

```c
#include <stdio.h>
#include <stdlib.h>
struct element {
  struct element * next;
  int data;
};
struct linked_list {
  struct element * head;
};

void append_int(struct linked_list * list, int val) {
  struct element * elem = malloc(sizeof(struct element));
  elem->data = val;
  elem->next = NULL; // Really important to explicitly set this to
      null. Malloc does not zero memory
  if (list->head == NULL) {
    // Empty list, we need to append to head
    list->head = elem;
  } else {
    // List has some elements, find the end and append to that
    struct element * tail = list->head;
    while (tail->next != NULL) {
      tail = tail->next;
    }
    tail->next = elem;
  }
}
```

4 LD_PRELOAD

Shared libraries allow software components to be shared between different applications. The C standard library is one example which contains many frequently used functions (`printf`, `fopen`, `malloc`, etc). Code which wishes to use these functions must include the library with `#include <stdlib.h>`.

Shared libraries are loaded into memory when a program starts. Sometimes it can be useful to override library functions such as those found in the C standard library. In order to intercept calls to the standard library we must write our own library and ensure that it is loaded before the standard library. If you set the `LD_PRELOAD` environment variable to the path of a shared object, that file will be loaded before any other library (including the C runtime, `libc.so`). When the target application calls a library function it will look in our library first. We can use this fact to override the default implementations of standard library functions.

```c
#include <stdio.h>
#include <stdint.h>
#include <dlfcn.h>

void * malloc(size_t size) {
  // Static variables keep their values between function calls
  static void* (*wrapped_malloc)(size_t) = NULL;
  if (wrapped_malloc == NULL) {
    // This will be called the first time our malloc is run
    // dlsym returns a function pointer to the "real" libc malloc
    wrapped_malloc = dlsym(RTLD_NEXT, "malloc");
  }
```

```
  // Pass our call through to the real malloc
  void *retval = wrapped_malloc(size);
  // Do extra logging
  printf("malloc(%d) = %p\n", size, retval);
  return retval;
}
```

The listing above demonstrates how we can wrap the `malloc` function to augment it with additional logging. This library will be loaded into the memory address space of another process to work, which means we must compile and run it a little differently than how we are used to. The article at [1] is worth reading for more examples and a more comprehensive description of how `LD_PRELOAD` functions. The following listing shows how we can compile our library and use it to monitor how programs allocate memory.

```
$ gcc  -D_GNU_SOURCE -fPIC -shared -ldl -o mymalloc.so mymalloc.c
$ LD_PRELOAD=./mymalloc.so date
malloc(5) = 0x730040
malloc(120) = 0x730060
....
malloc(20) = 0x7310c0
malloc(21) = 0x7310e0
Sun Jan 25 23:25:38 GMT 2015
```

Exercise 2: Double free errors occur when `free` is called on a memory address which has already been freed. When this happens, the program's memory management data structures become corrupted which can allow a malicious user to write values in arbitrary memory spaces. This corruption can cause the program to crash or even allow an attacker to execute arbitrary code.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  void *some_memory = malloc(100);
  free(some_memory);
  free(some_memory);
}
```

```
$ gcc -o double_free double_free.c
$ ./double_free
*** glibc detected *** ./double_free: double free or corruption
(fasttop): 0x0000000001b67010 ***
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x76a16)[0x7fd53d44aa16]
...
```

For this exercise you should use the `LD_PRELOAD` trick to hook the `free` function with a version of your own. Your function should maintain a list of previously freed memory locations. When it is called it should check to see whether it has already freed the memory location it was called with. If not,

your function should call the standard free function and add the new address to the freed list. If it has, it should simply return. **Note:** This is not an ideal solution to double frees. Can you think why?

5 Buffer Overflows

Buffer overflows are one of the oldest and most prevalent classes of security vulnerability. C is particularly vulnerable as it places responsibility for data integrity in the hands of the programmer. This means that when data is written to a variable there are no inbuilt safe-guards to ensure that what is written fits into the buffer it is written to. The seminal paper for buffer overflows was written by a hacker known by the alias Aleph One [2].

Exercise 3: Buffer overflows are the first explicitly security-related task of this course, and will feature prominently in next week's coursework. In order to become familiar with this type of vulnerability you should inspect the following code and fix any vulnerabilities present. **Hint:** a good place to start would be experimenting with passwords of different lengths while running gdb.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
  int authorized = 0;
  char password_buffer[16];
  strcpy(password_buffer, password);
  if (strcmp(password_buffer, "admin123") == 0) {
    authorized = 1;
  }
  return authorized;
}

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Usage: %s <password>\n", argv[0]);
  } else {
    if (check_authentication(argv[1])) {
      printf("Access Granted!\n");
    } else {
      printf("Access Denied!\n");
    }
  }
}
```

References

[1] Rafał Cieslak, Dynamic linker tricks: Using LD_PRELOAD to cheat, inject features and investigate programs. `https://rafalcieslak.wordpress.com/2013/04/02/dynamic-linker-tricks-using-ld_preload-to-cheat-inject-features-and-investigate-programs`

[2] Aleph One, *Smashing the Stack for Fun and Profit*, Phrack Magazine, Vol 7