overrun lab instructions and lab report template

This document includes both the lab instructions and empty boxes for you to embed your screenshots and answers to lab questions. Please use this document for your lab report.

Overview

This exercise illustrates overrunning the intended bounds of data structures in a C program.

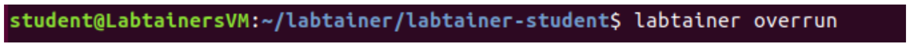
No coding is required in this lab, but it will help if the student can understand a simple C or C++ program. The GDB program is used to explore the executing program, including viewing a bit of its disassembly, however no assembly language background is necessary to perform the lab.

Lab Environment

This lab uses the Labtainer Linux-based virtual machine.

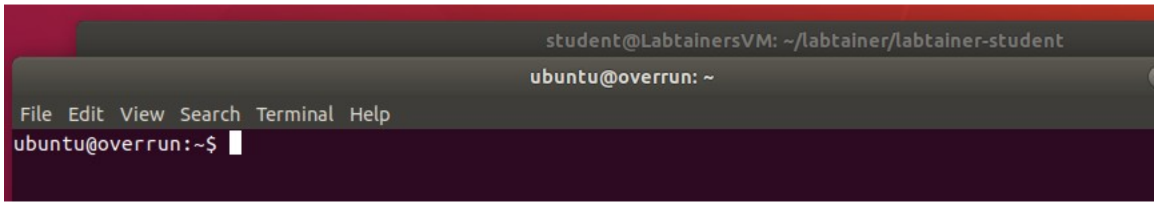
- Start the Labtainer VM. Once the VM is running, from the `labtainer-student` terminal window, start the lab using the command:

  `labtainer overrun`



```
student@LabtainersVM:~/labtainer/labtainer-student$ labtainer overrun
```

- Once the lab completes the startup process, a ***separate*** terminal window for `ubuntu@overrun` will open for all the lab tasks.



student@LabtainersVM: ~/labtainer/labtainer-student
ubuntu@overrun: ~
File Edit View Search Terminal Help

```
ubuntu@overrun:~$
```

- At the end of the lab tasks, you will return to the `labtainer-student` terminal to stop the lab.

Lab Tasks

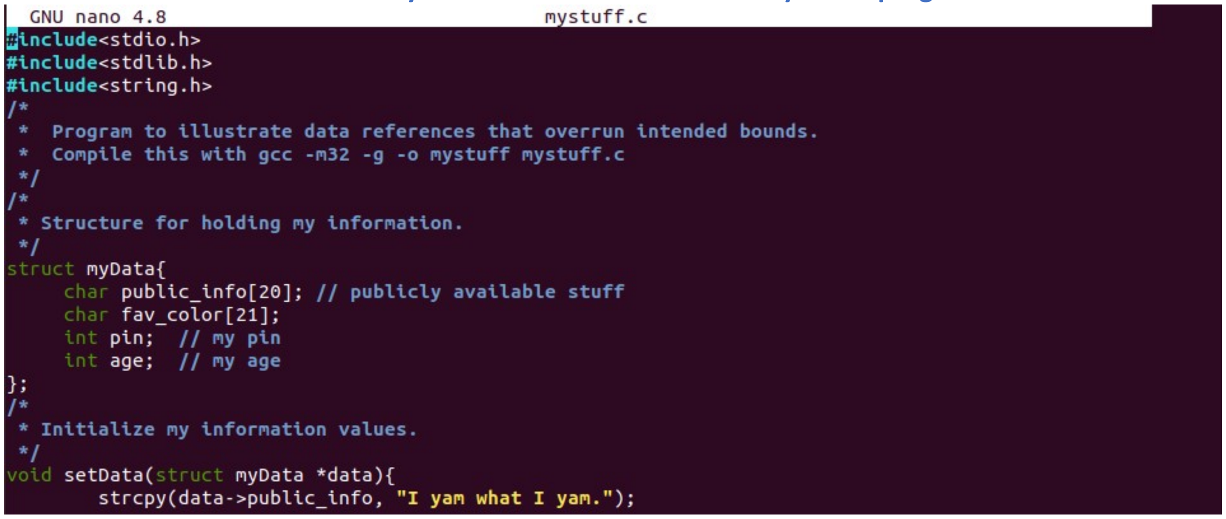
1) Review the mystuff.c program

At the ubuntu@overrun terminal, view the mystuff.c program. Use either the command line editors `vi` or `nano`, or just type `less mystuff.c` to view the code in the terminal window. (Type `q` to exit the `less` command.)

The myData structure

Look at the myData structure. In the program, the variable my_data is declared to be a myData struct. Note the public_info character array has 20 elements. As with any array, we can refer to elements of the array using an index. For example, my_data.public_info[4] refers to the fifth character in the array, and my_data.public_info[19] refers to the very last character in the array. If 19 is the very last character in the array, what would my_data.public_info[20] refer to?

INSERT YOUR SCREENSHOT OF myDATA STRUCTURE from the mystuff.c program.



```
  GNU nano 4.8                             mystuff.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
/*
 *  Program to illustrate data references that overrun intended bounds.
 *  Compile this with gcc -m32 -g -o mystuff mystuff.c
 */
/*
 * Structure for holding my information.
 */
struct myData{
     char public_info[20]; // publicly available stuff
     char fav_color[21];
     int pin;  // my pin
     int age;  // my age
};
/*
 * Initialize my information values.
 */
void setData(struct myData *data){
        strcpy(data->public_info, "I yam what I yam.");
```

Addresses of fields

After the program initializes the my data structure, it displays the addresses of the start of the public data field, and the pin field, and it displays the memory values of those fields.

Memory content

The program then enters a loop in which it allows the user to display hex values of individual characters within the public_info field. It is this loop that will let you explore the question asked earlier, namely: what would my_data.public_info[20] refer to?

ADD YOUR ANSWER: In the context of the mystuff.c program, what does my_data.public_info[20] refer to?

It refers to the memory location outside correct index range of the array, which means that it is considered an undefined behavior in a lot of programming languages.

2) Compile and run the program

Use this command to compile the program:

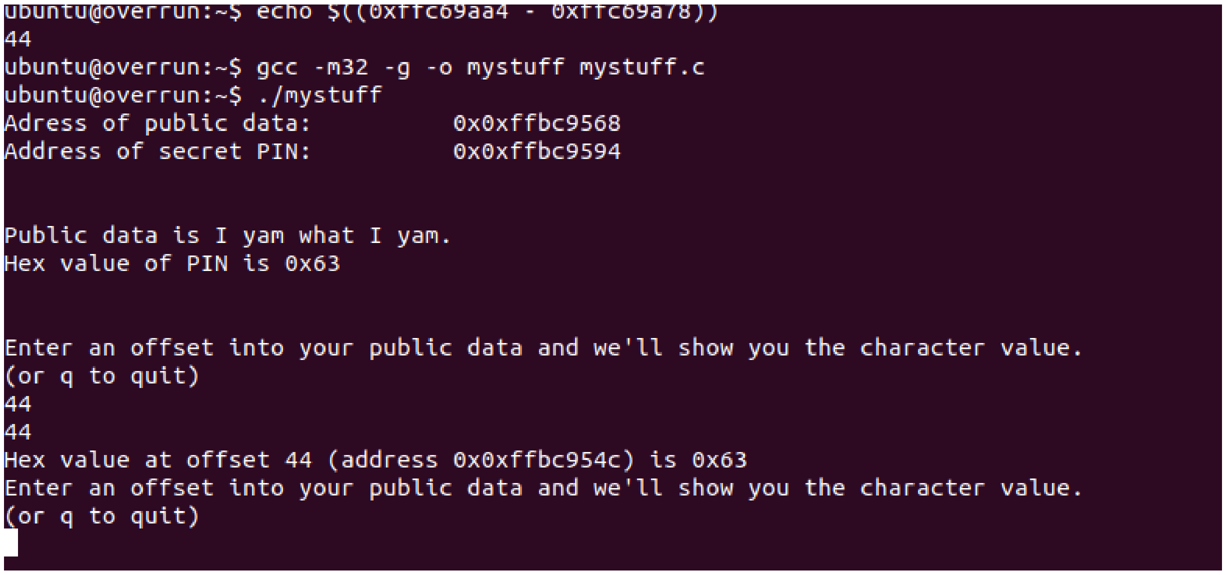
```
gcc -m32 -g -o mystuff mystuff.c
```

Note the -m32 switch creates a 32-bit binary and the -g switch includes symbols in the binary that will let us explore the program's execution using gdb.

Run the program with this command:
```
./mystuff
```

Explore the values displayed at different offsets within (and beyond) the public info field. Note the displayed address of the public info field and the address of the pin field. ***How many bytes separate the two fields?*** Use the program to display the value of the pin field. Note that if your `fav_color` buffer size is odd, the compiler will pad the buffer so that the next variable starts on 4-byte word boundary.

INSERT YOUR SCREENSHOT OF YOUR EXPLORATION OF DIFFERENT OFFSETS and the VALUE OF THE PIN FIELD.



```
ubuntu@overrun:~$ echo $((0xffc69aa4 - 0xffc69a78))
44
ubuntu@overrun:~$ gcc -m32 -g -o mystuff mystuff.c
ubuntu@overrun:~$ ./mystuff
Adress of public data:          0x0xffbc9568
Address of secret PIN:          0x0xffbc9594


Public data is I yam what I yam.
Hex value of PIN is 0x63


Enter an offset into your public data and we'll show you the character value.
(or q to quit)
44
44
Hex value at offset 44 (address 0x0xffbc954c) is 0x63
Enter an offset into your public data and we'll show you the character value.
(or q to quit)
```

ADD YOUR ANSWER: How many bytes separate the public_info and pin fields?
44 bytes

3) Explore with gdb

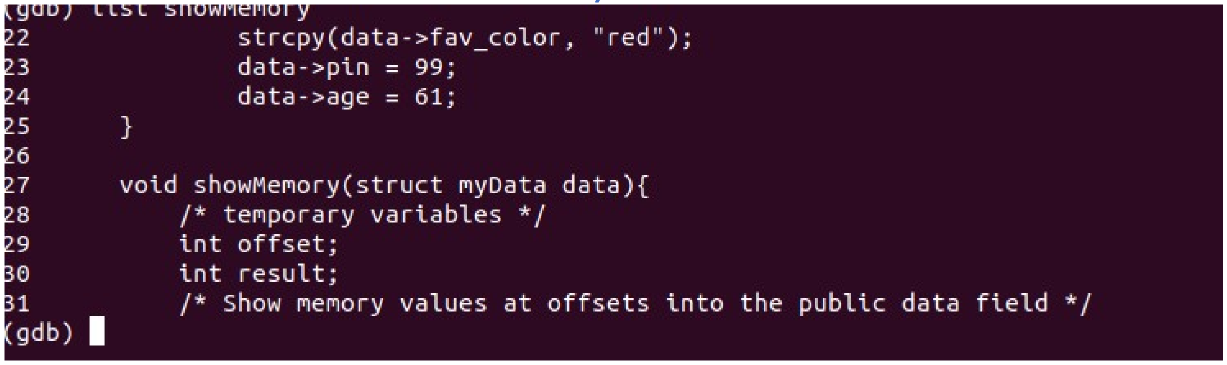
Run the program under the GDB debugger:

```
gdb mystuff
```

Use the `list` command to view the source code. Set a breakpoint in the `showMemory` function on the line where it will print the value at the given offset. (Use `list showMemory` to view the source code for that function.) And then run the program from within `gdb` (Replace `<line number>` with an actual line number for the breakpoint.):

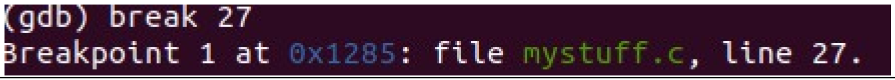
```
break <line number>

run
```

INSERT YOUR SCREENSHOT OF THE showMemory FUNCTION



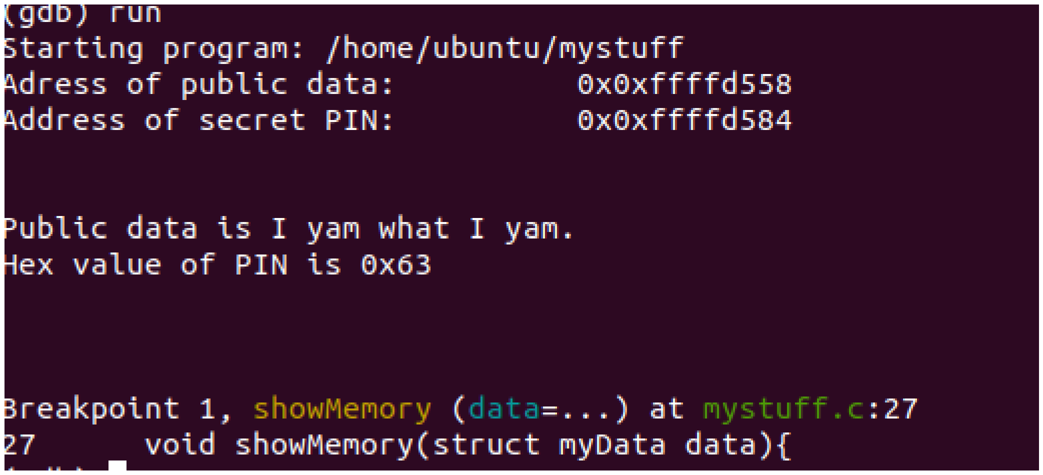
```
(gdb) list showMemory
22              strcpy(data->fav_color, "red");
23              data->pin = 99;
24              data->age = 61;
25      }
26
27      void showMemory(struct myData data){
28          /* temporary variables */
29          int offset;
30          int result;
31          /* Show memory values at offsets into the public data field */
(gdb) 
```

INSERT YOUR SCREENSHOT OF THE SETTING OF THE BREAKPOINT



```
(gdb) break 27
Breakpoint 1 at 0x1285: file mystuff.c, line 27.
```

INSERT YOUR SCREENSHOT OF THE PROGRAM RUN WITHIN GDB

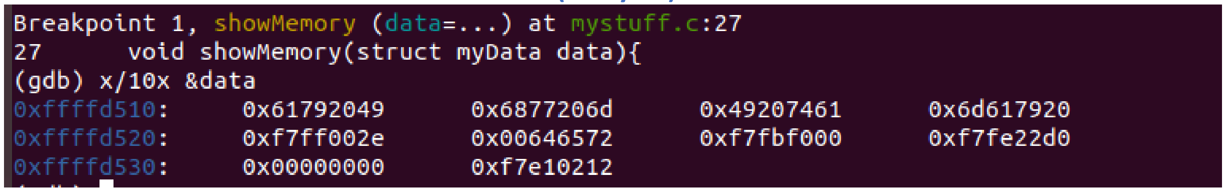


```
(gdb) run
Starting program: /home/ubuntu/mystuff
Adress of public data:          0x0xffffd558
Address of secret PIN:          0x0xffffd584


Public data is I yam what I yam.
Hex value of PIN is 0x63



Breakpoint 1, showMemory (data=...) at mystuff.c:27
27      void showMemory(struct myData data){
```

When the program hits the breakpoint, display 10 words (40 bytes) of system memory as hex values starting at the data structure:

```
x/10x &data
```

INSERT YOUR SCREENSHOT OF THE 10-words (40 bytes) DISPLAY



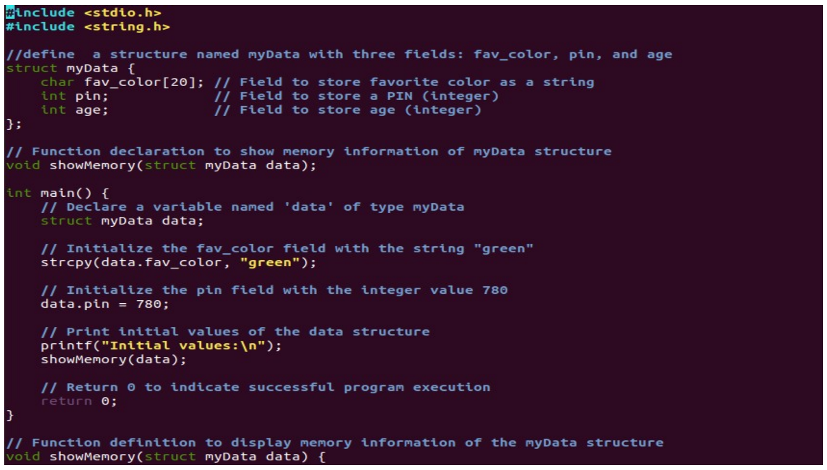
```
Breakpoint 1, showMemory (data=...) at mystuff.c:27
27      void showMemory(struct myData data){
(gdb) x/10x &data
0xffffd510:     0x61792049      0x6877206d      0x49207461      0x6d617920
0xffffd520:     0xf7ff002e      0x00646572      0xf7fbf000      0xf7fe22d0
0xffffd530:     0x00000000      0xf7e10212
```

Does the memory content correspond to what you observed while running the program?

ADD YOUR ANSWER: Does the memory content correspond to what you observed while running the program? The memory content does correspond to what I observed while running the program. The first part contains the ASCII implementation of “I yam what” and the other part includes a hexadecimal value of ‘0x00646572’ which may represent the color red in a specific format. The last two words are zeros which means null. This shows that the program was expecting initialization and knowing that there was a string within the program: “I yam what I yam.”

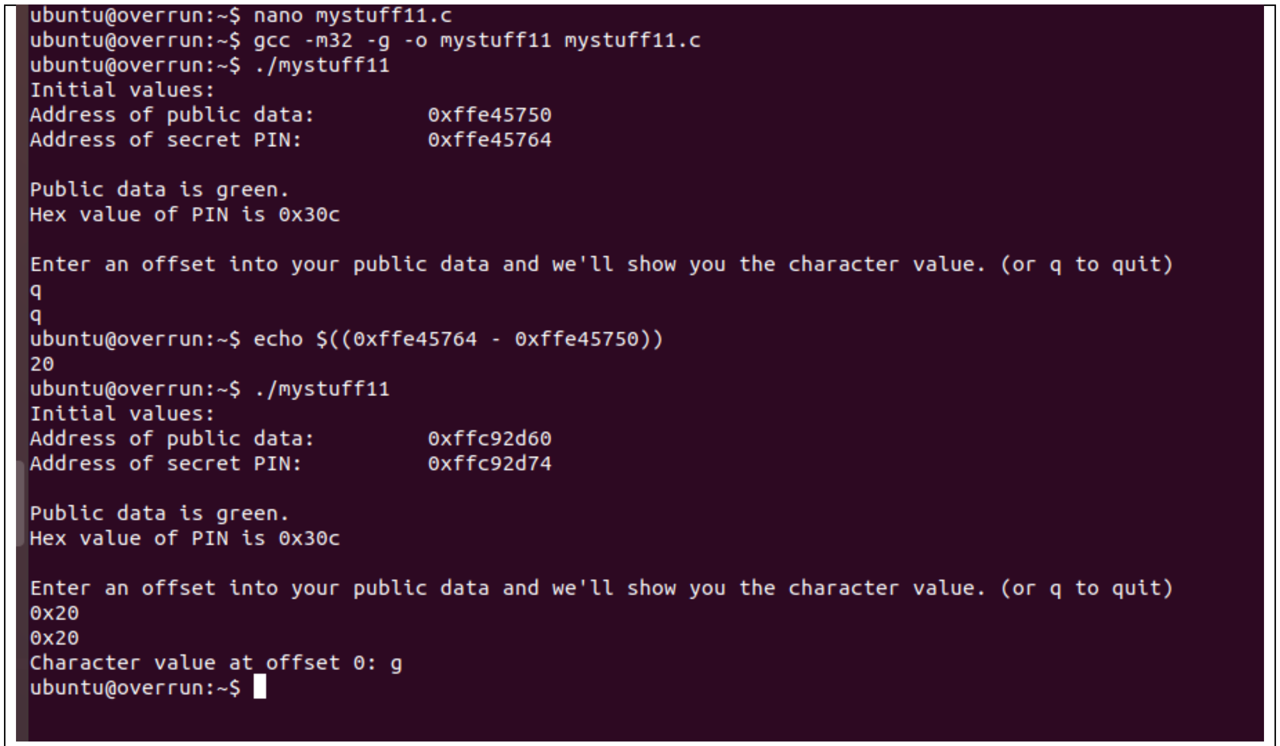
4) Add your own experiment to the lab that is not one of the tasks previously given but is relevant to the lab topics.

ADD YOUR EXPERIMENT: Describe what you did and add screenshots to document the steps and results.



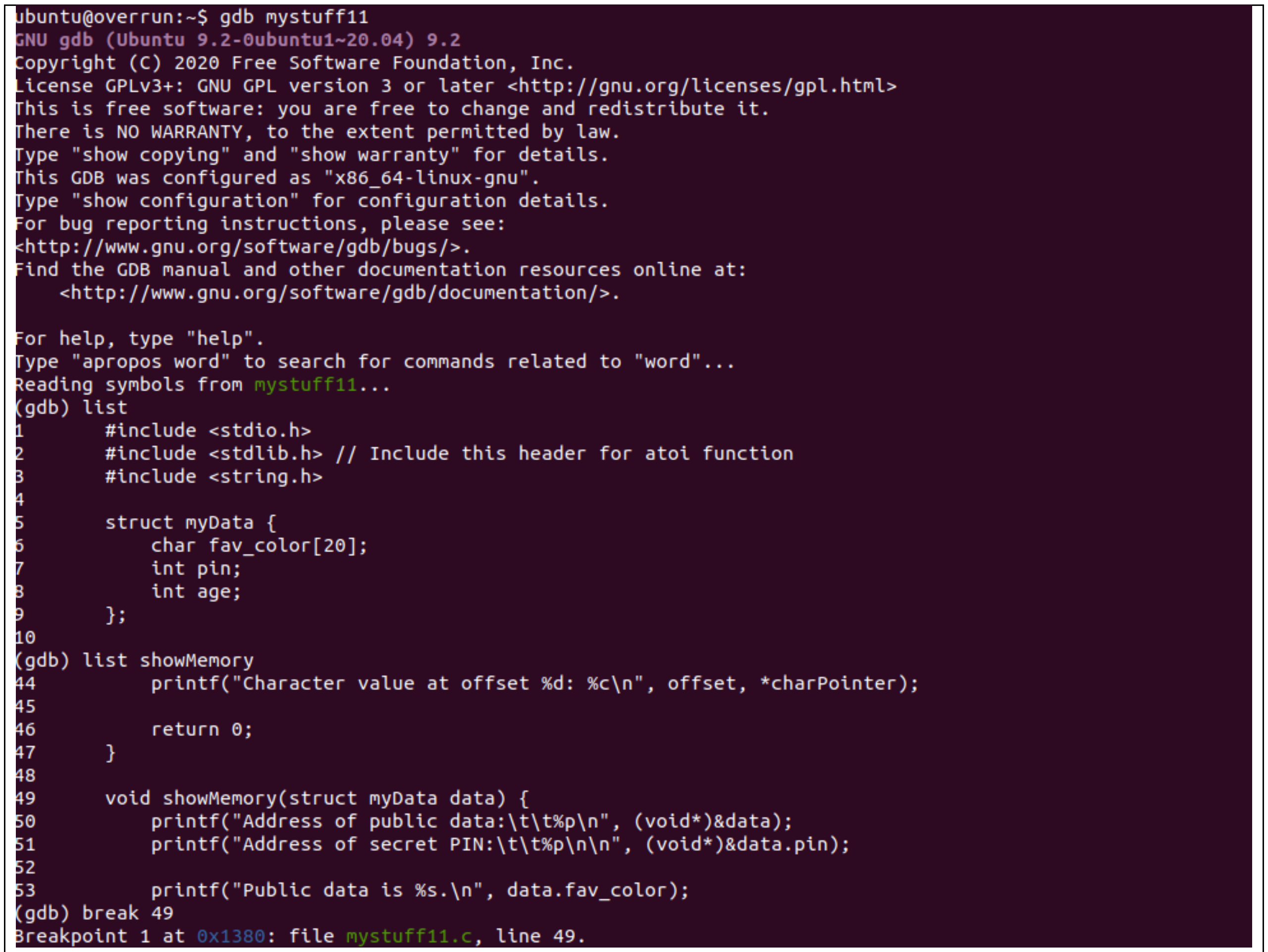
```c
#include <stdio.h>
#include <string.h>

//define  a structure named myData with three fields: fav_color, pin, and age
struct myData {
    char fav_color[20]; // Field to store favorite color as a string
    int pin;            // Field to store a PIN (integer)
    int age;            // Field to store age (integer)
};

// Function declaration to show memory information of myData structure
void showMemory(struct myData data);

int main() {
    // Declare a variable named 'data' of type myData
    struct myData data;

    // Initialize the fav_color field with the string "green"
    strcpy(data.fav_color, "green");

    // Initialize the pin field with the integer value 780
    data.pin = 780;

    // Print initial values of the data structure
    printf("Initial values:\n");
    showMemory(data);

    // Return 0 to indicate successful program execution
    return 0;
}

// Function definition to display memory information of the myData structure
void showMemory(struct myData data) {
```

1. I implemented a C program which is similar to the c file that was written for the lab.

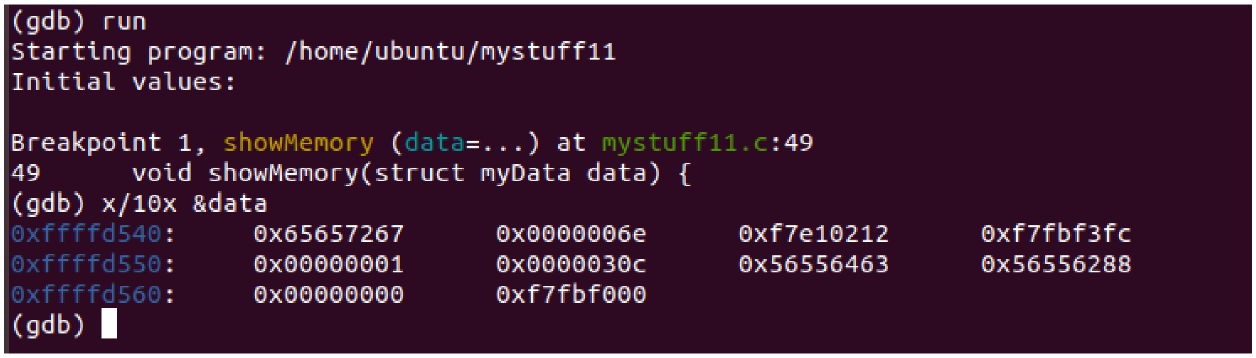


```
ubuntu@overrun:~$ nano mystuff11.c
ubuntu@overrun:~$ gcc -m32 -g -o mystuff11 mystuff11.c
ubuntu@overrun:~$ ./mystuff11
Initial values:
Address of public data:         0xffe45750
Address of secret PIN:          0xffe45764

Public data is green.
Hex value of PIN is 0x30c

Enter an offset into your public data and we'll show you the character value. (or q to quit)
q
q
ubuntu@overrun:~$ echo $((0xffe45764 - 0xffe45750))
20
ubuntu@overrun:~$ ./mystuff11
Initial values:
Address of public data:         0xffc92d60
Address of secret PIN:          0xffc92d74

Public data is green.
Hex value of PIN is 0x30c

Enter an offset into your public data and we'll show you the character value. (or q to quit)
0x20
0x20
Character value at offset 0: g
ubuntu@overrun:~$ 
```

2. Using the command "gcc -m32 -g -o mystuff11 mystuff11.c" to compile and to run the program "./mystuff11" to determine the offsets and bytes the program has. I calculated the hexadecimals and got the answer of 20 bytes. Entering an offset, I put 0x20 and it gave me a character value of 'g.'



```
ubuntu@overrun:~$ gdb mystuff11
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from mystuff11...
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h> // Include this header for atoi function
3       #include <string.h>
4
5       struct myData {
6           char fav_color[20];
7           int pin;
8           int age;
9       };
10
(gdb) list showMemory
44          printf("Character value at offset %d: %c\n", offset, *charPointer);
45
46          return 0;
47      }
48
49      void showMemory(struct myData data) {
50          printf("Address of public data:\t\t%p\n", (void*)&data);
51          printf("Address of secret PIN:\t\t%p\n\n", (void*)&data.pin);
52
53          printf("Public data is %s.\n", data.fav_color);
(gdb) break 49
Breakpoint 1 at 0x1380: file mystuff11.c, line 49.
```



```
(gdb) run
Starting program: /home/ubuntu/mystuff11
Initial values:

Breakpoint 1, showMemory (data=...) at mystuff11.c:49
49          void showMemory(struct myData data) {
(gdb) x/10x &data
0xffffd540:     0x65657267      0x0000006e      0xf7e10212      0xf7fbf3fc
0xffffd550:     0x00000001      0x0000030c      0x56556463      0x56556288
0xffffd560:     0x00000000      0xf7fbf000
(gdb) 
```

3. Just like the 3rd part of the lab I implemented the gdb command for the file I created for the c program. The output `0xffffd540: 0x65657267 0x0000006e 0xf7e10212 0xf7fbf3fc` represents the memory content at the address of the data structure. The first four bytes form the ASCII string "gerg," followed by a null-terminated string with an 'n'. The subsequent values are likely addresses or data. The following four bytes represent the integer `1`, and the next four indicate the value `780`. The remaining values are likely addresses or data. This output reveals the initialized values and potential addresses stored in the memory at the data structure's address.

Complete lab

After finishing, go to the terminal window that was used to start the lab and type:

```
      stoplab
```