ECE 264: Advanced C Programming

Lecture Notes 6/12/19

1 Topics

1. GDB basic usage using gdbdemo.c with demo
2. Why we need to look at assembly code? - Buffer Overflow Attack with demo
3. Data types and Strings - defining them, memory representation, initializing, literals, identifying their segment location.

2 GDB basic usage

We saw with the help of the file gdbdemo.c, how a program's execution can be controlled. In particular, we saw:

- how to enable a program for debugging
- how to set and manage breakpoints
- how to start, pause and resume a debug session.
- how to step through each statement, step-in and step-out of a function.
- how to print variable values, memory contents, and register contents.
- how to modify variables, memory contents.

Below is the source code of gdbdemo.c

```
1  #include<stdio.h>
2  int foo(int a, int b)
3  {
4      int x = a + 1;
5      int y = b + 2;
6      int sum = x + y;
7
8      return x * y + sum;
9  }
10 int main(int argc, char*argv[])
11 {
12    int ret = foo(10, 20);
13    printf("value returned from foo: %d\n",ret);
14    return 0;
15 }
```

3 GDB Advanced Usage

Below is the source code of wrongindex1.c:

```c
// wrongindex1.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char * * argv)
{
  int x = -2;
  int arr[] = {0, 1, 2, 3, 4};
  int y = 15;
  printf("Before: x = %d, y = %d\n", x, y);
  arr[7] = -353;
  printf("After: x = %d, y = %d\n", x, y);
  return EXIT_SUCCESS;
}
```

We tried to answer why we need to look at assembly code sometimes with the help of wrongindex1.c. wrongindex1.c accessed invalid array indices and this lead to modification of some local variables.

As the demo showed, without explicitly modifying the variable `y`, `y`'s value was changed. We did this through accessing `y`'s memory location indirectly and then overwriting the contents of that memory location. The integer array `arr` provided a convenient handle to access the memory location of `y`.

Essentially, we accessed contents beyond the space reserved for a fixed-length buffer. Buffer overflow attacks are fancier versions of this idea:

3.1 Buffer overflow attack

In simple terms, an attacker gives input too big for a fixed-length buffer. When this has the effect of overwriting the return address, normal program execution is highjacked. When the return address overwritten leads to a block of malicious code that encrypts or deletes files on a local file system, victim suffers.

We saw Buffer overflow attack in action with a demo using wrongindex3.c. Below is the source code of wrongindex.c

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

void dummy()
{
  int x;
  printf("Printing inside the dummy function\n");
  return;
}

void f1(void)
{
  int f1_top = 0xAAAAAAAA; // used as references
  char name[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
  int f1_bottom = 0xBBBBBBBB; // used as references
  name[40] = 0x6b; //These values may have to change
  //name[41] = 0x05; //These values may have to change
  dummy();
}

void f2(void)
{
  int f2_top = 0xCCCCCCCC; // used as references
  char message[] = "REALLY BAD IF YOU SEE THIS\n";
```

```
25    printf(message);
26    int f2_btm = 0xDDDDDDDD; // used as references
27  }
28
29  int main(int argc, char * * argv)
30  {
31    int main_top = 0xEEEEEEEE; // used as references
32    int main_top2 = 0xEEEEEEEE; // used as references
33    f1();
34    int main_btm = 0xFFFFFFFF; // used as references
35    return EXIT_SUCCESS;
36  }
```

Executing this program caused function `f2` getting executed even though the function was not called anywhere in the program. We saw using GDB how we could hijack function `f1`'s execution to execute function `f2` before function `f1` called function `dummy`.

The gdb demo followed similar steps as mentioned here:

https://engineering.purdue.edu/ milind/ece264/2017spring/assignments/pa02/

4 Data types and Strings

We looked at basic data types, compound types, and type modifiers in C. We also looked at the storage space requirements of those types and the values that a variable of a given type in C could take. We then looked at the string type in C, how a string is represented in memory, and how strings are initialized in C. We saw that strings could be initialized using *literals* or constants that are stored in the read-only area of data segment. We concluded the lecture with an exercise. This exercise identifed the segment where different strings appearing in a partial code would be stored.